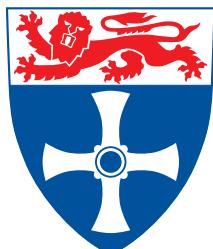

UNIVERSITY OF
NEWCASTLE UPON TYNE



Proceedings of the Second UK Embedded Forum

A. Koelmans, A. Bystrov, M. Pont, R. Ong, A. Brown (Editors)

Foreword

This volume contains the set of papers presented at the second UK Embedded Forum, a satellite event to the Embedded Systems Show (a yearly exhibition of embedded software and hardware products, organised by EDA Exhibitions Ltd). The aim of the Embedded Forum is to bring academic researchers in this economically important discipline together. The Forum aims to give PhD students from the UK the chance to present their work to their peers, and receive valuable feedback in return. A total of eighteen papers were presented, all of which are included in this volume. Their topics span an impressive range of research activities, from System-on-Chip design issues to software scheduling techniques.

The event took place on October 20th, 2005, at the National Exhibition Centre, Birmingham. The event was well attended, and produced some lively discussion throughout the day. We are deeply indebted to EDA Exhibitions Ltd, especially Jeremy Kenyon and Andrew Porter, for the local arrangements and general support.

The organisers intend to make this a yearly event for the foreseeable future.

Newcastle, Leicester, Southampton
November 2005
The Editors

© Copyright is held by the authors of the papers in this volume.
All rights reserved.
Published 2005, University of Newcastle upon Tyne.
ISBN 0-7017-0191-9.

Contents

- 4 Using XOR operations to reduce variations in the transmission time of CAN messages:
A pilot study
M. Nahas, M.J. Pont, University of Leicester
- 18 Using "planned pre-emption" to reduce levels of task jitter in a time-triggered hybrid scheduler
A. Maaita, M.J. Pont, University of Leicester
- 36 Building reliable embedded systems using Abstract Patterns, Patterns, and Pattern Implementation Examples
S. Kurian, M.J. Pont, University of Leicester
- 60 A 'Hardware-in-the Loop' testbed representing the operation of a cruise-control system in a passenger car
D. Ayavoo, M.J. Pont, J. Fang, M. Short, S. Parker, University of Leicester, Pi Technology
- 91 The use of ASIPs and customised Co-Processors in an Embedded Real-time System
J. Whitham, N. Audsley, University of York
- 111 On-line IDDQ testing of security circuits
J. Murphy, A. Bystrov, University of Newcastle
- 119 Design for low-power and high-security based on timing diversity
D. Sokolov, A. Bystrov, A. Yakovlev, University of Newcastle
- 137 The Use Of Scenarios To Improve The Flexibility In Embedded Systems
I. Bate, P. Emberson, University of York
- 157 Reconfigurable Hardware Network Scanning Packet Structure
K. Cheng, M. Fleury, University of Essex
- 171 How to Manage Determinism and Caches in Embedded Systems
R. Scottow, K.D. McDonald-Maier, University of Kent
- 177 Developing reliable embedded systems using a pattern-based code generation tool:
A case study
C. Mwelwa, M.J. Pont, D. Ward, University of Leicester, MIRA Ltd
- 194 Mining for Pattern Implementation Examples
S. Kurian, M.J. Pont, University of Leicester
- 202 On-chip Sub-Picosecond Phase Alignment
C. D'Alessandro, K. Gardiner, D.J. Kinniment, A. Yakovlev, University of Newcastle
- 209 Automatic conversion from 'single processor' to 'multi-processor' software architectures for embedded control systems
P.J. Vidler, M.J. Pont, University of Leicester
- 224 The PH Processor: A soft embedded core for use in university research and teaching
Z. Hughes, M.J. Pont, R. Ong, University of Leicester
- 246 Two novel shared-clock scheduling algorithms for use with CAN-based distributed systems
D. Ayavoo, M.J. Pont, M. Short, S. Parker, University of Leicester, Pi Technology
- 262 Comparing the performance and resource requirements of "PID" and "LQR" algorithms when used in a practical embedded control system: A pilot study
R. Bautista, M.J. Pont, T. Edwards, University of Leicester
- 290 A comparison of synchronous and asynchronous network architectures for use in embedded control systems with duplicate processor nodes
T. Edwards, M.J. Pont, M. Short, P. Scotson, S. Crumpler, University of Leicester, TRW Conekt

Using XOR operations to reduce variations in the transmission time of CAN messages: A pilot study

Mouaaz Nahas and Michael J. Pont

*Embedded Systems Laboratory,
University of Leicester, University Road, Leicester LE1 7RH, UK*

Abstract

Nolte and colleagues (e.g. Nolte *et al.* Proc. of the 8th IEEE Real-Time and Embedded Technology and Applications Symposium, San Jose, California. 2002) have described an approach which aims to reduce the impact of bit stuffing in networks employing the Controller Area Network (CAN) protocol. In this paper, we explore the impact of the Nolte technique on a set of general - random – CAN data and show that, as expected, the reduction in the level of bit stuffing is minimal. We go on to describe techniques for selectively applying the Nolte approach on a frame-by-frame or byte-by-byte basis to these random data. Using a case study, we then illustrate that a selective (byte-based) application of the Nolte approach has the potential to significantly reduce the level of bit stuffing in practical applications without requiring a large increase in CPU or memory requirements.

Acknowledgements

This project is support by the UK Government (EPSRC-DTA award). The work described in this paper was carried out while MJP was in study leave from the University of Leicester.

1. Introduction

The Controller Area Network (CAN) protocol is widely used in low-cost embedded systems (Farsi and Barbosa, 2000; Fredriksson, 1994; Thomesse, 1998; Sevillano *et al.*, 1998). The CAN protocol was introduced by Robert Bosch GmbH in the 1980s (Bosch, 1991). Although originally designed for automotive applications, CAN is now being used in process control and many other industrial areas (Farsi and Barbosa, 2000; Fredriksson, 1994; Thomesse, 1998; Sevillano *et al.*, 1998; Pazul, 1999; Zuberi and Shin, 1995; Misbahuddin and Al-Holou, 2003). As a consequence of its popularity and widespread use, most modern microcontroller families now include one or more members with on-chip hardware support for this protocol (e.g. Siemens, 1997; Infineon, 2000; Philips, 2004).

The bit representation used by CAN is "Non Return to Zero" (NRZ) coding. If (for example) two CAN controllers are linked, then their clocks are synchronised by means of the data edges. As a consequence, a CAN message contained a long sequence of identical bits, the clock in the CAN receiver may drift, with the consequent risk of message corruption (Farsi *et al.*, 2000). To eliminate the possibility of such a scenario, CAN incorporates a "bit stuffing" mechanism – at the physical layer - which operates as follows: when five identical bits (e.g. five 0s) have been transmitted on the bus, a bit of the opposite polarity is "stuffed" (transmitted) by the sender controller afterwards. The receiver controller follows the reverse protocol and removes the stuffed bit, thereby restoring the original data stream.

Whilst providing an effective mechanism for clock synchronization in the CAN hardware, the bit-stuffing mechanism causes the frame length to become (in part) a complex function of the data contents. It is useful to understand the level of bit stuffing that this process may induce. When using (for example) 8-byte data and standard CAN identifiers, the minimum message length will be 111 bits (without bit stuffing) and the maximum message length will be 135 bits (with the worst-case level of bit stuffing): see Nolte (2001) for details. At the maximum CAN baud rate (1 Mbit/sec), this translates to a possible variation in message lengths of 24 μ s.

Once transmission starts, a CAN message cannot be interrupted, and the variation in transmission times therefore has the potential to have a significant impact on the real-time behaviour of systems employing this protocol. One key impact can be on system jitter behaviour: for example, Matheson (2004) suggests that jitter values between 1 μ s and 120 μ s can be expected for CAN-based networks. The presence of significant jitter can have a detrimental impact on the performance of

many distributed embedded systems. For example, Cottet and David (1999) show that – during data acquisition tasks – jitter rates of 10% or more can introduce errors which are so significant that any subsequent interpretation of the sampled signal may be rendered meaningless. Similarly Jerri (1977) has discussed the serious impact of jitter on applications such as spectrum analysis and filtering.

Various studies have been carried out to explore ways of synchronising clocks in CAN-based networks. For example, Rodrigues *et al* (1998) describe a fault-tolerant clock-synchronisation algorithm for CAN. The algorithm is based on *a posteriori agreement* (Verissimo and Rodrigues, 1992) and can offer a precision of 100 μ s if the clock is synchronised once every 45 sec. An alternative approach is described by Nolte *et al.* (2001 and 2002). For example, Nolte *et al.* (2002) present a technique – based on encoding of the data transmitted on the CAN bus - in which the worst-case number of stuff-bits in a CAN message is reduced from 17 to 4, with a corresponding reduction in message-length variations.

In this paper, we begin by considering the results from the Nolte studies (Nolte *et al* 2001 and Nolte *et al* 2002). We seek to demonstrate that – while providing a significant jitter reduction for the data set considered in Nolte’s papers – the jitter reduction obtained is much less significant when a completely general (random) data set is considered. We go on to describe techniques for selectively applying the Nolte approach on a frame-by-frame or byte-by-byte basis. Using a case study, we seek to demonstrate that a selective (byte-based) application of the Nolte approach has the potential to significantly reduce the level of bit stuffing in practical applications without requiring a large increase in CPU or memory requirements.

We begin the main body of this paper by describing the Nolte approach.

2. The Nolte approach

By analyzing 25000 CAN frames from an automotive system, Nolte *et al.* (2001) found that the probability of having a bit value of 1 (or 0) in the data section was not 50%. More specifically, they observed that the probability of having consecutive bits of the same polarity was high, and that - therefore - the number of stuff bits is higher than would be expected with random data.

To reduce the number of stuff-bits inserted by the CAN protocol, Nolte suggested a simple encoding scheme based on an XOR operation. In this scheme, the data section of each CAN frame is XOR-ed with the bit-pattern 101010... (See Nolte *et al.*, 2001 and Nolte *et al.*, 2002 for more

details). At the receiving end, the same bit operation is applied again, to extract the original data (see Figure 1).

Original frame:	00000011110011000000111 ...
XOR with bit-mask:	101010101010101010101010 ...
Transmitted frame:	101010010100110010101101 ...

Figure 1: Encoding process for Nolte method.

3. Impact of the Nolte method on a random CAN traffic

As noted in Section 2, Nolte proposed the application of an XOR transform to reduce levels of bit stuffing in frames which were – in his application area – found to contain long sequences of identical bits.

In a more general case, the data transmitted may not have the same characteristics. Indeed, if we model a completely general CAN message using random data, then we would not expect to see a significant reduction in the level of bit-stuffing if the Nolte (XOR) transform is applied. To illustrate this, we created 10 million pseudo-random data frames, each with eight data bytes. The results from a simple analysis of these data are presented in Table 1.

Table 1: Results from Nolte technique applied to random CAN frames.

No. of frames exposed to CAN bit stuffing	Maximum number of stuff-bit	Average number of stuff-bit
8,932,166	10	2.27

Table 1 shows that – of the 10,000,000 frames - a total of 8,932,166 (around 89%) would be subject to CAN bit stuffing. In this data set, the maximum number of stuff bits (for any frame) was 10 and the average number of stuff bits (across all frames) was 2.27.

Table 2 then illustrates what happens if we apply the Nolte approach to all of the frames in the data set: for ease of later reference, we will refer to this approach as “Nolte A”.

Table 2: Results from Nolte technique applied to random CAN frames (Nolte A)

Bit-stuffed frames	Maximum. stuff bits	Average stuff bits	Reduction in frames	Reduction in max bits	Reduction in average bits
8,931,642	10	2.27	0.006%	0%	0%

In Table 2, “Reduction in frames” shows the reduction in the number of frames which are subject to bit stuffing after Nolte A is applied: in this case, the result is small (0.006%). Similarly, the reductions in the maximum number of stuff bits (0%) and the average number of stuff bits (0%) are also small. Overall, we can conclude that Nolte A is having a minimal impact on the level of bit stuffing for the random data.

Table 3 shows the results obtained in response to an alternative application of the Nolte approach (which we will call “Nolte B”). This table uses the same data set used in Table 2. This time, however, the frames are tested individually before Nolte is applied: in situations where – for the whole frame – bit stuffing will not occur, the frame is transmitted unaltered. Only where bit-stuffing will be applied (to the “raw” frame) is the frame subject to an XOR transform.

Table 3: Results from Nolte technique applied to random CAN frames (Nolte B)

Bit-stuffed frames	Maximum. stuff bits	Average stuff bits	Reduction in frames	Reduction in max bits	Reduction in average bits
7,927,015	10	2.22	11.25%	0%	2.2%

In this case, we note that “Reduction in frames” and reduction in the average number of stuff bits are larger after Nolte B is applied. However, no reduction is obtained in the maximum number of stuff bits.

Table 4 shows the results obtained in response to a third implementation of the Nolte approach (which we will call “Nolte C”). This table again uses the same data set. This time, however, each byte of data in each frame is tested individually before Nolte is applied: in situations where – for the byte – bit stuffing will not occur, the byte is transmitted unaltered. Only where bit-stuffing will be applied is the byte subject to an XOR transform.

Table 4: Results from Nolte technique applied to random CAN frames (Nolte C)

Bit-stuffed frames	Maximum. stuff bits	Average stuff bits	Reduction in frames	Reduction in max bits	Reduction in average bits
5,638,654	6	1.39	36.87%	40%	38.77%

In this case, we note that, against all the measures made here, there has been a reduction in the level of bit stuffing. In this case, the “Reduction in frames” is approximately 37%. The reductions in the maximum and average number of stuff bits are at similar levels.

4. Case study: Practical application of the selective Nolte methods

From the small study described in Section 3, we note that XOR transform suggested by Nolte has – as expected - little impact on the random data set. However, by applying Nolte’s transform selectively (when required) we can significantly reduce the level of bit stuffing.

Of course, the study outlined in Section 2 was highly artificial, and took no account of – for example – the need to transmit information about the encoding process to the receiver, to allow successful decoding of the data stream. In this section, we present a small case study in which the selective Nolte methods (outlined in Section 3) are incorporated in an existing network protocol.

4.1 Shared-clock (S-C) protocol

The “shared-clock” (S-C) architecture (Pont, 2001) operates as follows (see Figure 2). On the Master node, a conventional (co-operative or hybrid) scheduler executes and the scheduler is driven by periodic interrupts generated from an on-chip timer. On the Slave nodes, a very similar scheduler operates. However, on the Slaves, no timer is used: instead, the Slave scheduler is driven by interrupts generated through the arrival of messages sent from the Master node.

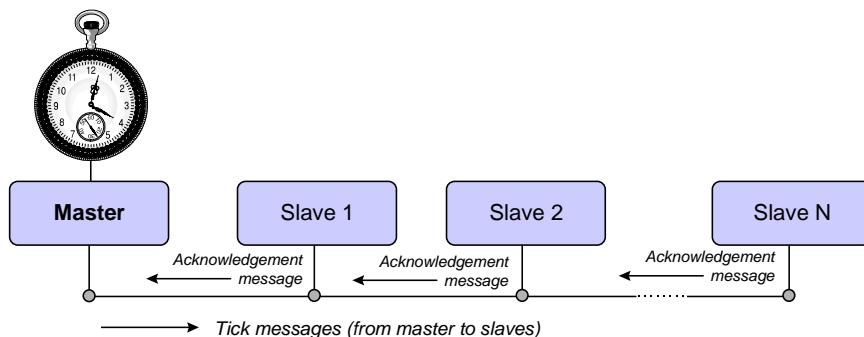


Figure 2: Simple architecture of CAN-based shared-clock (S-C) scheduler.

The S-C architecture provides an effective platform to try out the selective Nolte transforms because – when using such an architecture - there will be jitter in the timing of tasks on the Slave nodes if there is any variation in the time taken to send “tick” messages between the Master and the Slaves: such a variation in transmission times can arise due to CAN-based bit stuffing. Figure 3 below shows the impact on the Slave tick timing.

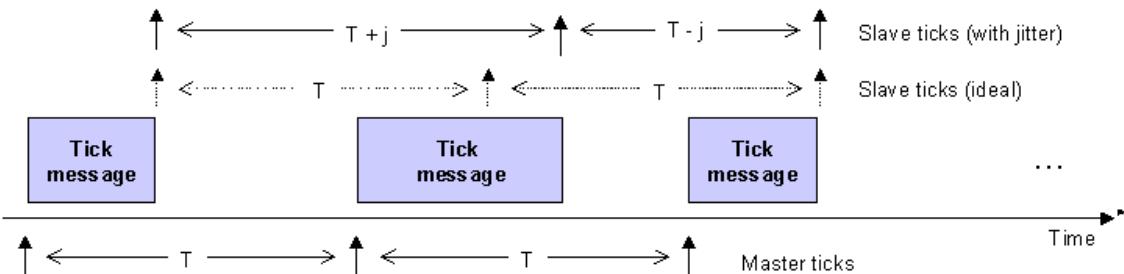


Figure 3: Impact of frame length on the Slave tick in the S-C system.

4.2 Applying the Nolte transforms in S-C architectures

We describe how the selective Nolte methods were incorporated in the S-C architecture in this section.

a) Network architecture

The test platform used in this study had two nodes: one Master and one Slave. Each node was based on a Phytec board, supporting a (16-bit) C167 microcontroller. The oscillator frequency was 20 MHz. For this implementation, we ported the 8051 design from (Pont, 2001) to the C16x family. The Keil C166 compiler was used (Keil Software, 1998). The network nodes were connected using a twisted-pair CAN link. The CAN baudrate was 1 Mbit/sec.

Note that, while 8-byte “Tick” messages were used, one byte was reserved for the Slave ID (see Pont, 2001): thus, in the studies considered here, up to 7 bytes of real data were transmitted.

b) Nolte A

In this method we XOR-ed every byte of the CAN data message (apart from the Slave ID) with the bit pattern 10101010. In this method the maximum data bandwidth of eight bytes can be used.

c) Nolte B

In this method, we check each CAN frame and – if a sequence of five identical bits is detected - we XOR the whole frame with the bit mask (10101010 ...).

To allow decoding, we need only one bit to indicate whether the frame is masked or not. To make best use of the available bandwidth, we used one bit in Byte 1 (which otherwise contains the Slave ID) to store the masking information. Appropriate coding schemes were used to ensure that the bit stuffing was not introduced in the Slave ID byte (see Figure 4).

Bit 1	Bit 2	Bit 3	Bit 4	Bit 5	Bit 6	Bit 7	Bit 8
Slave ID	Slave ID	Masking info	Slave ID				
0	0	1 or 0	0	0	0	1	0

Figure 4: Layout for Byte 1 in Nolte B

d) Nolte C

In this method, we check the CAN frame on byte-by-byte basis, and once a byte contains a sequence of five identical bits is detected, this particular byte will be masked using Nolte bit-mask.

To hold the masking information, we require 1 bit per byte of data. In this case, we chose to use 6 bytes for data, and therefore needed six bits. However, it was necessary to ensure that these 6 bits did not themselves introduce bit stuffing. We therefore (as with Nolte B) stored one bit of the Slave-ID byte to store decoding information, along with 5 bits (and appropriate padding) in the last CAN data byte (see Figure 5 - Figure 7).

Byte 1	Byte 2	-----	Byte 7	Byte 8
Slave ID + one bit for masking info		Actual data		Masking info

Figure 5: Layout for data field in Nolte C

Bit 1	Bit 2	Bit 3	Bit 4	Bit 5	Bit 6	Bit 7	Bit 8
Adjustable based on the last bit	Relevant to data byte 2	Relevant to data byte 3	Relevant to data byte 4	Adjustable based on the last bit	Relevant to data byte 6	Relevant to data byte 7	Adjustable based on the last bit

Figure 6: Layout for Byte 8 in Nolte C

Bit 1	Bit 2	Bit 3	Bit 4	Bit 5	Bit 6	Bit 7	Bit 8
Slave ID	Slave ID	Relevant to data byte 5	Slave ID				

Figure 7: Layout for Byte 1 in Nolte C

For example, if Byte 8 equals to: 01010110 and Byte 1 equals to: 00100010, the receiving node will know that bytes 2, 4, 5, 6 and 7 were masked (Slave ID was again selected with extra care to avoid exposure to bit-stuffing: see Figure 4).

4.3 Jitter measurement methodology

In this study, we wished to measure the differences between the start time of “Task A” on the Master (the only task running on this node), and the start time of “Task B” on the Slave (the first of ten tasks running on this node)*.

To make these measurements, a pin on the Master node was set “high” (for a short period) at the start of the Task A. Similarly, a pin on the Slave was set low at the start of Task B. The signals from these two pins were then AND-ed (using an 74LS08N chip: Texas Instruments, 2004), to give a pulse stream illustrated in Figure 8.

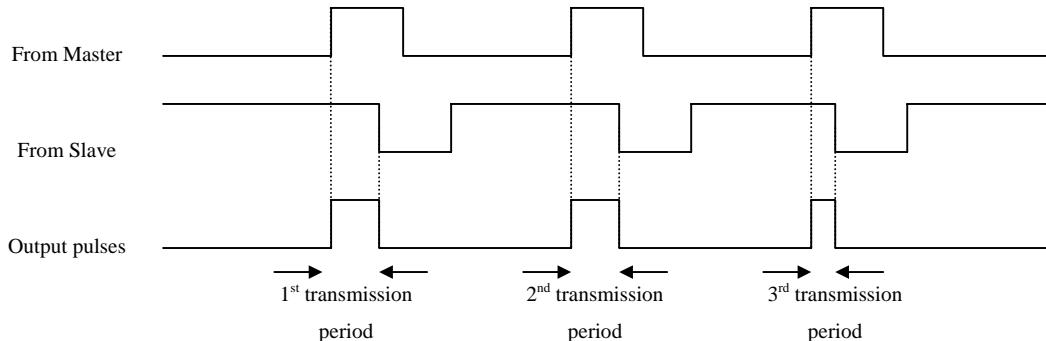


Figure 8: Method used to measure the transmission time in the CAN system.

In each set of results presented in this paper, 50,000 pulse widths (that is, the durations of 50,000 message transmissions) were measured using the data set described in Section 4.4. In each case, the width of the pulses was measured using a National Instruments data acquisition card ‘NI PCI-6035E’ (National Instruments, 2004), used in conjunction with appropriate LabVIEW software (v7.1: LabVIEW, 2004). The maximum, minimum and average message transmission times are reported.

* The measured jitter in this case represents the jitter introduced by both the node-to-node communications *and* the scheduler. However, the scheduler we used here had “reduced jitter” (see Nahas *et al.*, 2004).

To assess the jitter levels, we also report the average jitter and the difference jitter. The difference jitter was obtained by subtracting the best-case (minimum) transmission time from the worst-case (maximum) transmission time from the measurements in the sample set. The average jitter is represented by the standard deviation in the measure of average message transmission time.

4.4 Data set

These CAN messages used in this study were created from a total of 40,000 pseudo-random bytes of data which were produced using a desktop C program. Each of the studies discussed in this paper used an identical data set as the basis of its CAN messages. To support a larger (on going) study, the complete data set was split into five sets, each containing 8,000 bytes.

In each experiment described here, one of the five data sets was stored in an array in the Master node, allowing the generation of up to 1,000 “random” CAN frames. We sent each of these frames ten times and measured the results (that is, we sent 10,000 messages). We repeated each of these studies five times, using a different data set in each case. In total, therefore, 50,000 messages were transmitted (and measured) in each study.

4.5 Task jitter from the three methods of Nolte on a S-C design

To provide a meaningful comparison, we measured the jitter for the following protocols:

1. Original data without masking (with 8 data bytes[†] in CAN message)
2. Original data without masking (7 data bytes)
3. Nolte A (8 data bytes)
4. Nolte A (7 data bytes)
5. Nolte B (8 data bytes)
6. Nolte B (7 data bytes)
7. Nolte C (7 data bytes[‡])

The results from all methods are shown in Table 5 and Table 6, and then compared in Figure 9.

Note that all values are in microseconds and presented at the maximum CAN baudrate (1 Mbit/s).

[†] In each case, the “data bytes” include the Slave ID (byte) used in the S-C protocol. Thus, when we – for example – transmit “8 data bytes” we are sending 7 bytes of “real” user data plus 1 byte of Slave ID: see Section 4.2 and Pont (2001).

[‡] In the case of Nolte C, we transmit 8 bytes of CAN data. However, 1 byte is required to store the decoding information so that only 6 bytes of “real” user data are transmitted, plus 1 byte for the Slave ID. For comparison with the other methods, we view this as a “7-byte” method.

Table 5: Results from 8-byte methods.

	Original	Nolte A	Nolte B	Nolte C
Min transmission time	167.4	167.4	167.9	----
Max transmission time	177.6	177.5	177.9	----
Average transmission time	171.7	171.6	171.8	----
Difference jitter	10.2	10.1	10	----
Average jitter	1.5	1.4	1.5	----

Table 6: Results from 7-byte methods.

	Original	Nolte A	Nolte B	Nolte C
Min transmission time	157.5	157.4	157.8	165
Max transmission time	166.6	166.4	166.9	172.1
Average transmission time	160.4	160.6	161	167.1
Difference jitter	9.1	9	9.1	7.1
Average jitter	1.4	1.3	1.4	1.2

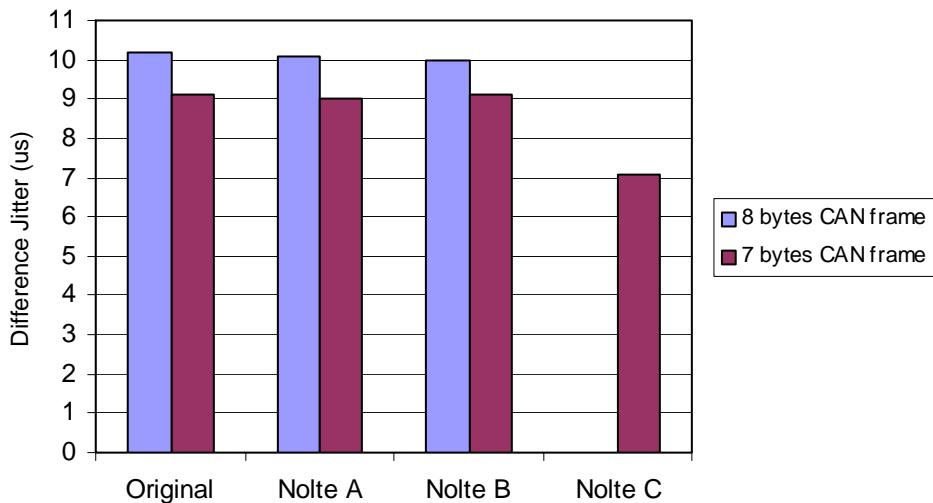


Figure 9: Jitter levels from all methods described above.

From the tables and graph, we can see that the jitter levels were not improved by applying the Nolte A or Nolte B methods on the random CAN data. However, the Nolte C method did help to substantially reduce the impact of CAN bit stuffing. The minimum level of (difference) jitter measured here was approximately 7 μ s, compared to a figure of 9-10 μ s with the original (raw) data.

4.6 Memory and CPU requirements

In many embedded designs, memory and CPU resources are limited. It is therefore important to understand the cost (in terms of resource requirements) of implementing these methods.

For each method, the average duration of the encoding and decoding processes was measured in hardware using LabVIEW measurement tools. For each case, 1000 samples were recorded and then averaged to give the values presented in Table 7.

Table 7: CPU load imposed by the three methods of Nolte technique.

	Encoding process (ms)	Decoding process (ms)
Nolte A	0.0158	0.0157
Nolte B	0.2457	0.0144
Nolte C	0.2896	0.0218

From the values shown in the table, the Nolte C encoding scheme required a CPU overhead of approximately 0.3 ms on this hardware platform.

The data and code memory requirements for all methods considered here are shown in Table 8.

Table 8: Memory requirements for all methods described here.

	Master			Slave	
	Data (Bytes)	ROM (Bytes)	RAM (Bytes)	ROM (Bytes)	RAM (Bytes)
Original	8	1610	32	1590	108
	7	1582	31	1574	106
Nolte A	8	1632	32	1616	108
	7	1608	31	1596	106
Nolte B	8	1986	40	1632	108
	7	1962	39	1612	106
Nolte C	7	1854	35	1650	108

As a summary, to implement Nolte C on a C167 microcontroller, we required an extra 4 data bytes (RAM) and 272 code bytes (ROM) for the encoding process. The decoding process required no extra data bytes and 76 extra code bytes. To put these figures in context, each C167 processor used in this study has 2 kbytes of on-chip RAM and 32 kbytes of on-chip ROM (Infineon, 2000).

5. Discussion and conclusion

In this paper, we have explored the impact of the techniques proposed by Nolte *et al.* on a set of general - random – CAN data and shown that, as expected, the reduction in the level of bit stuffing is minimal. We then described techniques for selectively applying the Nolte approach on a frame-by-frame (“Nolte B”) or byte-by-byte (“Nolte C”) basis to these random data. In a case study – using 7 bytes of random data – we showed that the overall reduction in jitter (arising from bit stuffing) was approximately 22% when Nolte C was applied.

The CPU and memory load imposed by all methods used in this study were also considered. Examining the results presented in Section 4.6, we can see that the modifications to the original Nolte method have resulted in a comparatively small increase in CPU and memory requirements. However, a CAN bandwidth reduction of 12.5% (as a result of the need to transmit encoding information with each message) must also be taken into account when deciding whether to employ these techniques.

Finally, we note that other approaches can also be used to reduce variations in the transmission times of CAN messages: for example, in a recent study, we explored the effectiveness of "software" bit stuffing as an alternative solution (see Nahas et al., 2005, for further details).

References

- Bosch (1991), Robert Bosch GmbH “CAN Specification Version 2.0”.
- Cottet, F. and David, L. (1999) “A solution to the time jitter removal in deadline based scheduling of real-time applications”, 5th IEEE Real-Time Technology and Applications Symposium - WIP, Vancouver, Canada, pp. 33-38
- Farsi, M. and Barbosa, M. (2000) “CANopen Implementation, applications to industrial networks”, Research Studies Press Ltd, England.
- Fredriksson, L.B. (1994) “Controller Area Networks and the protocol CAN for machine control systems”, Mechatronics Vol.4 No.2, pp. 159-192.
- Infineon (2000) “C167CR Derivatives 16-Bit Single-Chip Microcontroller”, Infineon Technologies.
- Jerri, A.J. (1977) “The Shannon sampling theorem: its various extensions and applications a tutorial review”, Proc. of the IEEE, vol. 65, n° 11, p. 1565-1596.
- LabVIEW 6.1 user guide: WWW webpage: <http://www.ni.com/pdf/manuals/322884a.pdf> [accessed May 2005]
- Matheson, M. (2004) “High performance motion control based on Ethernet” Proceedings of IEE Computing & Control, 2004.
- Misbahuddin, S.; Al-Holou, N. (2003) “Efficient data communication techniques for controller area network (CAN) protocol”, Computer Systems and Applications, 2003. Book of Abstracts. ACS/IEEE International Conference on, Pages:22.

- Nahas, M., Pont, M.J. and Jain, A. (2004) "Reducing task jitter in shared-clock embedded systems using CAN". In: Koelmans, A., Bystrov, A. and Pont, M.J. (Eds.) Proceedings of the UK Embedded Forum 2004 (Birmingham, UK, October 2004), pp.184-194. Published by University of Newcastle upon Tyne [ISBN: 0-7017-0180-3].
- Nahas, M., Short, M. and Pont, M. J. (2005) "The impact of bit stuffing on the real-time performance of a distributed control system", Proceeding of the 10th International CAN conference (Rome, Italy, March 2005), pp. 10-1 to 10-7.
- National Instruments; PCI-6035E data sheet and specs; WWW webpage:
http://www.ni.com/pdf/products/us/4daqsc202-204_ETCx2_212_213.pdf [accessed May 2005]
- Nolte, T.; Hansson, H.; Norström, C. and Punnekkat, S. (2001) "Using Bit-stuffing Distributions in CAN Analysis", IEEE/IEE Real-Time Embedded Systems Workshop (Satellite of the IEEE Real-Time Systems Symposium) London
- Nolte, T., Hansson, H. A., Norström, C. (2002) "Minimizing CAN response-time jitter by message manipulation", The 8th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2002), San Jose, California. 2002
- Pazul, K. (1999) "Controller Area Network (CAN) Basics", Microchip Technology Inc. Preliminary DS00713A-page 1 AN713.
- Philips (2004) "LPC2119/2129/2194/2292/2294 microcontrollers user manual", Philips Semiconductor.
- Pont, M.J. (2001) "Patterns for time-triggered embedded systems: Building reliable applications with the 8051 family of microcontrollers", ACM Press / Addison-Wesley. ISBN: 0-201-331381.
- Rodrigues, L., Guimarães, M. and Rufino, J. (1998) "Fault-Tolerant Clock Synchronization in CAN" Proceedings of the 19th IEEE Real-Time Systems Symposium, Madrid, Spain, December, 1998.
- Sevillano J L, Pascual A, Jiménez G and Civit-Balcells A (1998) "Analysis of channel utilization for controller area networks" Computer Communications, Volume 21, Issue 16, Pages 1446-1451
- Siemens (1997) "C515C 8-bit CMOS microcontroller, user's manual", Siemens.
- Texas Instruments: 74LS08 Datasheet, WWW webpage:
<http://www.cs.amherst.edu/~sfkaplan/courses/spring-2002/cs14/74LS08-datasheet.pdf> [accessed May 2004]
- Thomesse, J. P. (1998) "A review of the fieldbuses" Annual Reviews in Control, Volume 22, Pages 35-45
- Verissimo, P. and Rodrigues, L. (1992) "A posteriori agreement for fault-tolerant clock synchronization on broadcast networks" In Digest of Papers, The 22nd International Symposium on Fault-Tolerant Computing (IEEE), Boston, USA, 1992.
- Zuberi, K. M. and Shin, K. G. (1995) "Non-Preemptive Scheduling of Messages on Controller Area Network for Real-Time Control Applications", in Proc. Real-Time Technology and Applications Symposium, pp. 240-249.

Using “planned pre-emption” to reduce levels of task jitter in a time-triggered hybrid scheduler

Adi Maaita and Michael J. Pont

*Embedded Systems Laboratory,
University of Leicester, University Road, Leicester LE1 7RH, UK*

Abstract

This paper is concerned with the production of embedded systems with minimal resource requirements and highly predictable patterns of behaviour. Our particular concern is with levels of jitter in the start times of the pre-emptive task in systems implemented using a “time-triggered hybrid” (TTH) software architecture. We introduce a technique - “planned pre-emption” – which is intended to reduce this jitter level. We go on to present results from an initial study which suggest that the use of planned pre-emption can reduce jitter in the pre-emptive task by a factor of 5 (or more), when compared to the original TTH architecture.

Acknowledgements

Work on this paper is supported by an award from Al-Isra Private University of Jordan to AM. The work described in this paper was carried out while MJP was on Study Leave from the University of Leicester.

1. Introduction

This paper is concerned with the scheduling of tasks in low-cost embedded systems where predictable behaviour is a key design consideration. In such systems, the possible scheduling options may be distinguished by considering the manner in which the execution of tasks is triggered. For example, if all the tasks are invoked by aperiodic events (typically implemented as hardware interrupts) the system may be described as ‘event triggered’ (Nissanke, 1997). Alternatively, if all the tasks are invoked periodically (say every 10 ms), under the control of a timer, then the system may be described as ‘time triggered’ (Kopetz, 1997). The nature of the tasks themselves is also significant. If the tasks, once invoked, can pre-empt (or interrupt) other tasks, then the system is said to be ‘pre-emptive’; if tasks cannot be interrupted, the system is said to be co-operative (or “non-preemptive”).

When compared to pre-emptive schedulers, co-operative schedulers have a number of desirable features, particularly for use in safety-related systems (Allworth, 1981; Ward, 1991; Nissanke, 1997; Bate, 2000). For example, Bate (2000) identifies the following four advantages of co-operative scheduling: [1] The scheduler is simpler; [2] The overheads are reduced; [3] Testing is easier; [4] Certification authorities tend to support this form of scheduling.

One of the simplest implementations of a co-operative scheduler is a cyclic executive (e.g. see Baker and Shaw, 1989; Locke, 1992): this is one form of time triggered, co-operative (TTC), architecture. Provided that an appropriate implementation is used TTC architectures are a good match for a wide range of applications. For example, we have previously described in detail how these techniques can be applied in various automotive applications (e.g. Short and Pont, 2005; Ayavoo et al., 2004), a wireless (ECG) monitoring system (Phatrapornnant and Pont, in press), various control applications (e.g. Bautista et al., 2005; Edwards et al., 2004; Key et al., 2004), and in data acquisition systems, washing-machine control and monitoring of liquid flow rates (Pont, 2002).

Despite having many excellent characteristics, a TTC solution will not always be appropriate. The main problem with this architecture is that long tasks will have an impact on the responsiveness of the system. This concern is succinctly summarised by Allworth: “[*The*] *main drawback with this [co-operative] approach is that while the current process is running, the system is not responsive to changes in the environment. Therefore, system*

processes must be extremely brief if the real-time response [of the] system is not to be impaired." (Allworth, 1981). We can express this concern slightly more formally by noting that if the system must execute one or more tasks of (worst-case) execution time e and also respond within an interval t to external events then, in situations where $t < e$, a pure co-operative scheduler will not generally be suitable.¹

When there is a mismatch between task execution times and system response times, it is tempting to opt immediately for a full pre-emptive design: indeed, some studies seem to suggest that this is the only alternative. For example, Locke (1992) - in a widely cited publication - suggests that "*traditionally, there have been two basic approaches to the overall design of application systems exhibiting hard real-time deadlines: the cyclic executive ... and the fixed priority [pre-emptive] architecture.*" (p.37). More recently Bate (1998) compared cyclic executives and fixed-priority pre-emptive schedulers (exploring, in greater depth, Locke's study from a few years earlier). Other researchers have also decided not to consider cyclic executives, but for different reasons. For example, according to Liu and Ha, (1995): "*[An] objective of reengineering is the adoption of commercial off-the-shelf and standard operating systems. Because they do not support cyclic scheduling, the adoption of these operating systems makes it necessary for us to abandon this traditional approach to scheduling.*"

However, there are other design options available. For example, we have previously described ways in which support for a single, time-triggered, pre-emptive task can be added to a TTC architecture, to give we have called a "time-triggered hybrid" (TTH) scheduler (Pont, 2001). This architecture is illustrated in Figure 1. This figure shows the situation where a short pre-emptive task is executed every millisecond, while a co-operative task (with a duration greater than 1 ms) is "simultaneously" executed every 3 milliseconds.

¹ Please note that in this paper we are concerned with situations in which it can be determined at the time of system design that $t < e$. In some situations, a system error (e.g. sensor failure) may cause a task to overrun at run time. This problem (which may apply with both TTC and TTH schedulers, and others) can be addressed with an appropriate form of "task guardian". Task guardians are not considered in the present paper: please refer to Hughes and Pont (2004) for further information about this topic.

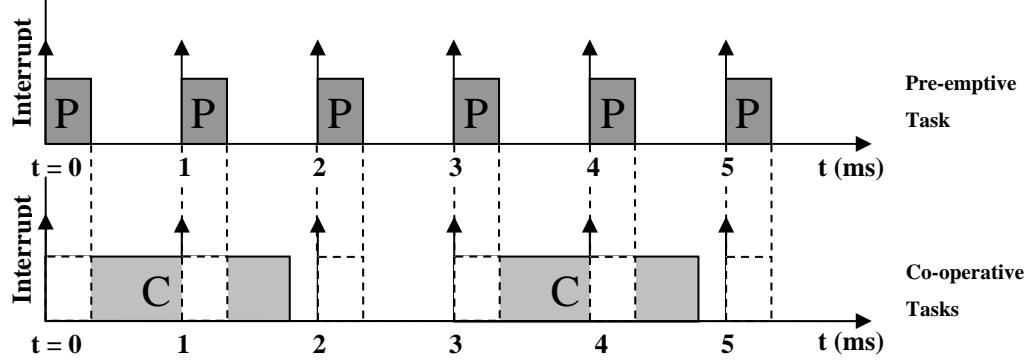


Figure 1: Illustrating the operation of a TTH scheduler. See text for details.

Overall, as we have previously discussed at length (Pont, 2001) a TTH scheduler combines the features of pre-emptive and co-operative schedulers, without requiring complex mechanisms for dealing with shared resources. As such, this simple but flexible architecture has the potential to “bridge the gap” between pure TTC implementations (such as cyclic executives) and full pre-emptive schedulers.

However, while a TTH architecture may serve as a cost-effective replacement for a full pre-emptive design in some circumstances, both solutions are susceptible to jitter. Jitter can have a very serious impact on systems in which a TTH design might otherwise be highly appropriate. For example, Cottet and David (1999) show that – during data acquisition tasks – jitter rates of 10% or more can introduce errors which are so significant that any subsequent interpretation of the sampled signal may be rendered meaningless. Similarly Jerri (1977) discusses the detrimental impact of jitter on applications such as spectrum analysis and filtering. Also, in control systems, jitter can greatly degrade the performance by varying the sampling period (Torgren, 1998; Mart et al., 2001).

In this paper, we discuss a solution to the problem of jitter in TTH designs using an approach which we refer to as “planned pre-emption”: such an approach is only possible with time-triggered architectures.

This paper is organised as follows. In Section 2 we discuss two possible ways in which a time triggered hybrid scheduler can be implemented. In Section 3 we compare the performance of the different types of hybrid schedulers, considering the jitter produced by each. In Section 4, we propose a technique for jitter reduction, and we explain how such an approach can be implemented in two practical TTH schedulers. In Section 5, we demonstrate

the effect of the new technique on the jitter levels observed in the two scheduler implementations. Our conclusions are presented in Section 6.

2. Implementing a TTH scheduler

In this section we will discuss two possible ways in which a time triggered hybrid scheduler can be implemented.

a) A TTH-SL scheduler

The first implementation we will consider here is a hybrid “super loop” scheduler (TTH-SL scheduler).

During normal operation of this system, the first function to be run (after any startup code) is `main()`. Function `main()` will start an endless `while` loop (see Listing 1), in which the co-operative task(s) will be executed in a sequential manner.

```
while (1)
{
    C_Task();
    // Execute other co-operative tasks (as required)
}

void P_Dispatch_ISR(void)
{
    P_Task();
}
```

Listing 1: A simple implementation of a “TTH-SL” scheduler

In “parallel” with this activity, a timer-based interrupt occurs every millisecond (in typical implementations) and invokes the ISR `P_Dispatch_ISR()`. `P_Dispatch_ISR()` then directly calls the pre-emptive task, which we will assume is called `P_Task()`: see Listing 1. Once the pre-emptive task is complete it returns control to the co-operative task which has been interrupted, and that task continues its execution from the point of interruption.

An overview of the architecture of this scheduler is shown in Figure 2.

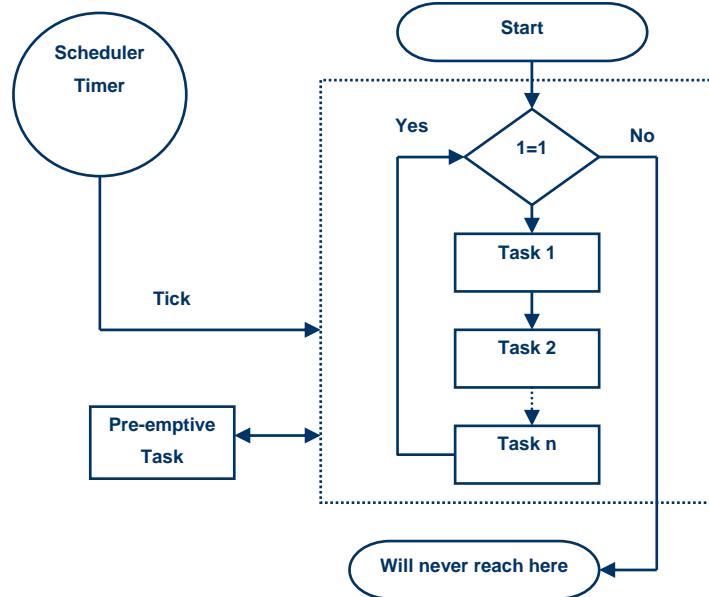


Figure 2: TTH-SL Scheduler Flowchart

In this architecture, the processor time is used to its maximum capacity as the co-operative tasks will be executed endlessly.

b) A more complete TTH Scheduler

We refer to the second (more complete) scheduler implementation considered in this paper simply as a “TTH Scheduler”.

During normal operation of the systems using the TTH Scheduler architecture, function `main()` runs an endless `while` loop (see Listing 2) from which the function `C_Dispatch()` is called: this in turn launches the co-operative task(s) currently scheduled to execute. Once these tasks are completed, `C_Dispatch()` calls `Sleep()`, placing the processor into a suitable “idle” mode.

```

while (1)
{
    C_Dispatch();           // Dispatch Co-op tasks
}

void C_Dispatch (void)
{
    // Go through the task array
    // Execute "C_Task()" when due to run
    // Execute other co-operative tasks (as required)

    // The scheduler may enter idle mode at this point
    Sleep();
}

void P_Dispatch_ISR(void) // ISR
{
    P_Task();               // Dispatch pre-emptive task
}

```

Listing 2: TTH Scheduler

In parallel with this activity, a timer-based interrupt occurs every millisecond (in typical implementations) which invokes the ISR `P_Dispatch_ISR()`. `P_Dispatch_ISR()` then calls the pre-emptive task: see Listing 2.

An overview of the architecture of this scheduler is shown in Figure 3.

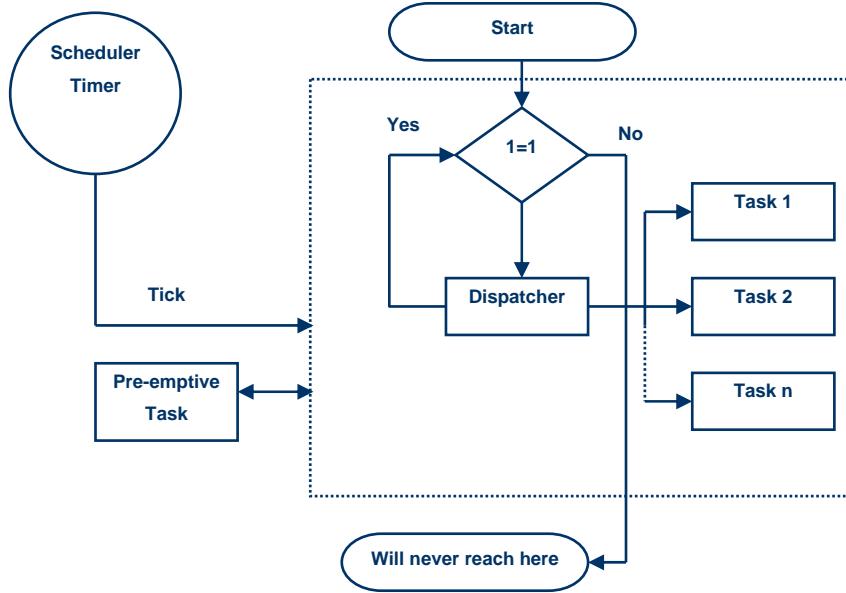


Figure 3: TTH Scheduler Flowchart

3. Jitter assessment for the TTH-SL Scheduler and TTH Scheduler

In this section we will consider and compare the performance of the TTH-SL and the TTH architectures, in terms of the jitter produced.

b) Platform

For the studies described in this paper, we used an ARM7 platform: the specific processor used was the Philips LPC2129. The LPC2129 is a 32-bit microcontroller with an ARM7-core which can run – under control of an on-chip phase-locked loop (PLL) – at frequencies from 10 MHz to 60 MHz (Philips, 2004). For the purposes of this paper, the LPC2129 was mounted on a Keil MCB2129 evaluation board.

The compiler used was the ARM GCC 3.4.1 operating in Windows by means of Cygwin (a Linux emulator). The IDE used was the Keil ARM development kit.

c) Methodology

To compare the jitter produced by the TTH-SL and the TTH architectures, we conducted a series of tests using the platform described in the previous section. In this initial study, we used dummy tasks to represent both the pre-emptive and the co-operative tasks: these consisted of simple “for loop” delays.

We wanted to measure the differences between the start time of one execution of the dummy task, and the start time of the next execution of the same task. To make these measurements,

a pin on the ARM7 microcontroller was set “high” (for a short period) at the start of the dummy task, and then set “low” before the next execution of the task. The widths of the resulting pulses was measured using a National Instruments data acquisition card ‘NI PCI-6035E’ (National Instruments, 2004), used in conjunction with appropriate software (LabVIEW, 2004). The resolution of the timing measurements was $0.1 \mu\text{s}$.

In each study, 50000 consecutive pulse widths were measured to give the results presented in this paper.

d) Results

The measurements described in the previous section produced the results shown in Table 1.

	TTH-SL Scheduler	TTH Scheduler
Max time (s)	0.0010002	0.0010004
Min time (s)	0.0009997	0.0009996
Avg time (s)	0.0001000	0.0001000
Std dev (s)	0.0000002	0.0000001
Jitter (s)	0.0000005	0.0000008

Table 1: Jitter results using dummy tasks.

The results in Table 1 show that there are variations in the pulse widths measured: these variations correspond to jitter in the task timings. In this study, it was found that both the TTH-SL architecture and the TTH architecture produced measurable levels of jitter.

d) Discussion

When a timer interrupt occurs, the processor must complete the current instruction before servicing the interrupt. In the TTH-SL architecture, the jitter occurs because different instructions take different periods of time to complete and – therefore – the time taken to respond to the timer interrupt is variable. In addition, when using the TTH scheduler, the processor will – sometimes – be in “idle” mode when the timer interrupt occurs: leaving idle mode is expected to take longer than any “conventional instruction”, and we see evidence of this in the results obtained.

3. The TTHj architecture

In this section, we consider a simple “planned pre-emption” mechanism which is expected to reduce the level of jitter observed in TTH designs.

a) Overview of the approach

As we noted in Section 3c, jitter in TTH designs is inevitable, since the time taken to respond to interrupts varies depending on the state of the processor at the time of the interrupt. In the TTC architecture (Pont, 2001) such jitter is not observed, because the processor is – in normal circumstances - always in the same (idle) state when the timer interrupt occurs.

Because the TTH architecture is time triggered, we know when the next pre-emptive task is due to execute: this allows us to implement a “planned pre-emption” scheme. Specifically, in this case, we aim to reduce the jitter level by ensuring that the processor is always in idle mode when the pre-emptive task is due to be dispatched. We do this (in the present implementation) by using a second timer to trigger a “Sleep” ISR, putting the processor into idle mode just before the timer overflow (linked to the dispatch of the pre-emptive task) is triggered.

b) A TTHj-SL Scheduler

We first consider how we can incorporate the second timer in the TTH-SL Scheduler, resulting in what we will refer to here as a “TTHj-SL Scheduler”.

During the operation of the system using the TTHj-SL architecture, the function `main()` starts an endless `while` loop (see Listing 3), in which the co-operative tasks are executed in a sequential manner. In parallel with this execution, a timer-based interrupt occurs every millisecond (in typical implementations) which invokes `P_Dispatch_ISR()`.

`P_Dispatch_ISR()` starts the “idle timer” which will overflow after around 90%² of the interval between “pre-emptive ticks” has elapsed. After this, the pre-emptive task is executed.

After the pre-emptive task completes its execution, control returns to the co-operative task list, where it will remain until the idle timer overflows: the ISR associated with this overflow

² The precise timing will depend on the scheduler overhead, which will – in turn – depend on the CPU performance of the processor used.

will place the system in idle mode, where it will remain until the timer for the pre-emptive task overflows, and `P_Dispatch_ISR()` is called once again.

```

while (1)
{
    C_Task();
    // Execute other co-operative tasks (as required)
}

void P_Dispatch_ISR(void)
{
    ITimer();           // Start idle timer
    P_Task();          // Dispatch pre-emptive task
}

void Idle_Timer_ISR(void)
{
    Sleep();           // Enter idle mode
}

```

Listing 3: TTHj-SL scheduler

An overview of the architecture of this scheduler is shown in Figure 4.

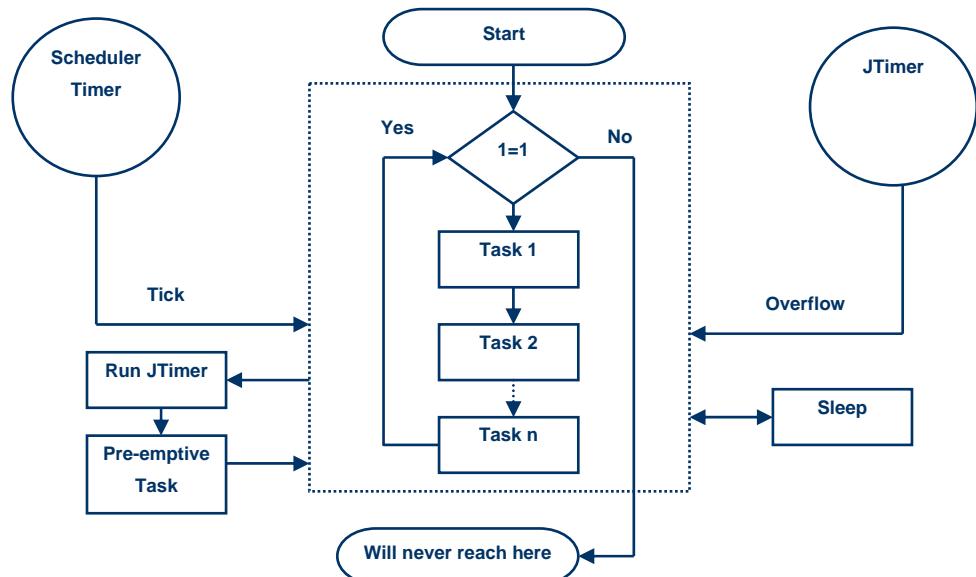


Figure 4: TTHj-SL scheduler Flowchart

c) The TTHj Scheduler

We next consider how we can incorporate the second timer in the TTH Scheduler architecture, resulting in what we will refer to here as a “TTHj Scheduler”.

During the operation of the system using the TTHj Scheduler architecture, the function `main()` starts an endless `while` loop (see Listing 4), in which the function `C_Dispatch()` executes the co-operative task(s) scheduled to be run.

```
while (1)
{
    C_Dispatch();
}

void C_Dispatch (void)
{
    // Go through the task array
    // Execute "C_Task()" when due to run
    // Execute other co-operative tasks (as required)

    // The scheduler may enter idle mode at this point
    Sleep();
}

void P_Dispatch_ISR(void)
{
    ITimer();           // Start idle timer
    P_Task();          // Dispatch pre-emptive task
}

void Idle_Timer_ISR(void)
{
    Sleep();           // Enter idle mode
}
```

Listing 4: TTHj Scheduler

A timer-based interrupt occurs every millisecond (in typical implementations) and invokes `P_Dispatch_ISR()`. `P_Dispatch_ISR()` starts the “idle timer” which will – as with the TTHj-SL Scheduler - overflow after around 90% of the interval between “pre-emptive ticks” has elapsed. After this, `P_Task()` is executed.

After the pre-emptive task completes its execution, control returns to the co-operative task list, where it will remain until the idle timer overflows, as with the TTHj-SL architecture.

An overview of the architecture of this scheduler is shown in Figure 5.

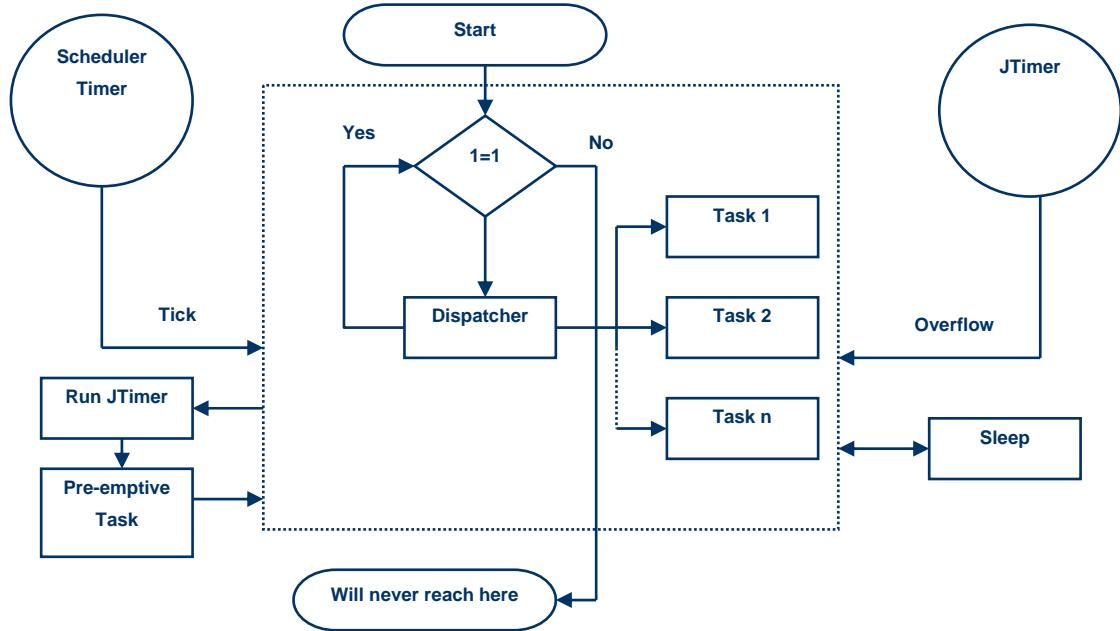


Figure 5: TTHj scheduler Flowchart

5. Comparing the jitter performance of TTH and TTHj architectures

In this section we will discuss and compare the performance of the TTH and the TTHj architectures, in terms of the jitter produced.

d) Platform

The platform used was as described in Section 3.a.

e) Methodology

The methodology used was as described in Section 3.b.

f) Results

The measurements described in the previous section produced the results shown in Table 2:

	TTH-SL Scheduler	TTHj-SL Scheduler	TTH Scheduler	TTHj Scheduler
Max time (s)	0.0010002	0.001000	0.0010004	0.001000
Min time (s)	0.0009997	0.0009999	0.0009996	0.0009999
Avg time (s)	0.0001000	0.0001000	0.0001000	0.0001000
Std dev (s)	0.0000002	0.0000000	0.0000001	0.0000000
Jitter (s)	0.0000005	0.0000001	0.0000008	0.0000001

Table 2: Jitter results using dummy tasks.

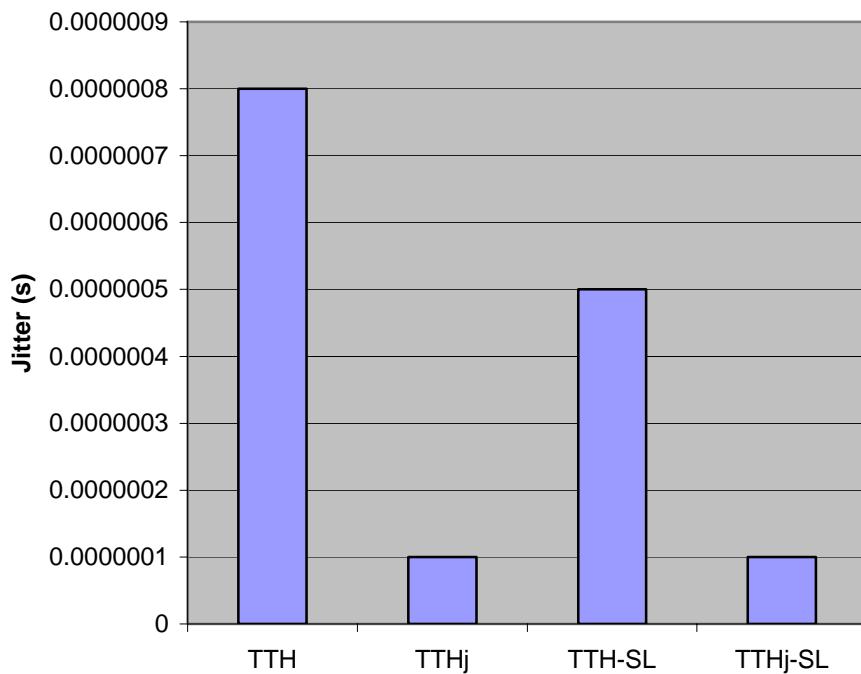


Figure 6: Jitter produced for dummy tasks

g) Discussion

The results above show that using planned pre-emption significantly reduces jitter observed in the start times of pre-emptive tasks (when compared with the original TTH scheduler implementations).

This improvement in performance requires a small increase in the system (source code) size.

4. Conclusions

This paper has been concerned with the production of embedded systems with minimal resource requirements and highly predictable patterns of behaviour. Our particular focus was on the levels of jitter in the start times of the pre-emptive task in systems implemented using a “time-triggered hybrid” (TTH) software architecture. We have introduced a technique - “planned pre-emption” – which is intended to reduce this jitter level, and demonstrated the effectiveness of this approach in a simple example. Overall, based on the results obtained here, the use of planned pre-emption can reduce jitter in the pre-emptive task by a factor of 5 (or more), when compared to the original TTH architecture.

References

- Allworth, S.T. (1981) “*An Introduction to Real-Time Software Design*”, Macmillan, London.
- Ayavoo, D., Pont, M.J. and Parker, S. (2004) “Using simulation to support the design of distributed embedded control systems: A case study”. In: Koelmans, A., Bystrov, A. and Pont, M.J. (Eds.) *Proceedings of the UK Embedded Forum 2004* (Birmingham, UK, October 2004), pp.54-65. Published by University of Newcastle upon Tyne [ISBN: 0-7017-0180-3].
- Baker, T.P. and Shaw, A. (1989) “The cyclic executive model and Ada”, *Real-Time Systems*, 1(1): 7-25.
- Bate, I.J. (1998) “Scheduling and timing analysis for safety critical real-time systems”, PhD thesis, University of York, UK.
- Bate, I.J. (2000) “Introduction to scheduling and timing analysis”, in “*The Use of Ada in Real-Time System*” (6 April, 2000). IEE Conference Publication 00/034.
- Bautista, R., Pont, M.J. and Edwards, T. (2005) “Comparing the performance and resource requirements of ‘PID’ and ‘LQR’ algorithms when used in a practical embedded control system: A pilot study”, paper presented at the 2nd UK Embedded Forum (Birmingham, UK, October 2005).
- Cottet, F. and David, L. (1999) “A solution to the time jitter removal in deadline based scheduling of real-time applications”, 5th IEEE Real-Time Technology and Applications Symposium - WIP, Vancouver, Canada, pp. 33-38.
- Edwards, T., Pont, M.J., Scotson, P. and Crumpler, S. (2004) “A test-bed for evaluating and comparing designs for embedded control systems”. In: Koelmans, A., Bystrov, A. and

Pont, M.J. (Eds.) Proceedings of the UK Embedded Forum 2004 (Birmingham, UK, October 2004), pp.106-126. Published by University of Newcastle upon Tyne [ISBN: 0-7017-0180-3].

Jerri, A.J. (1977) "The Shannon sampling theorem: its various extensions and applications a tutorial review", Proc. of the IEEE, vol. 65, n° 11, p. 1565-1596.

Key, S. and Pont, M.J. (2004) "Implementing PID control systems using resource-limited embedded processors". In: Koelmans, A., Bystrov, A. and Pont, M.J. (Eds.) Proceedings of the UK Embedded Forum 2004 (Birmingham, UK, October 2004), pp.76-92. Published by University of Newcastle upon Tyne [ISBN: 0-7017-0180-3].

Kopetz, H. (1997) "Real-time systems: Design principles for distributed embedded applications", Kluwer Academic.

LabVIEW 6.1 user guide: WWW webpage: <http://www.ni.com/pdf/manuals/322884a.pdf> [accessed May 2004]

Liu, J.W.S. and Ha, R. (1995) "Methods for validating real-time constraints", *Journal of Systems and Software*.

Locke, C.D. (1992) "Software architecture for hard real-time systems: Cyclic executives vs. Fixed priority executives", *The Journal of Real-Time Systems*, 4: 37-53.

Mart, P., Fuertes, J. M., Villt, R. and Fohler, G. (2001), "On Real-Time Control Tasks Schedulability", European Control Conference (ECC01), Porto, Portugal, pp. 2227-2232.

National Instruments; PCI-6035E data sheet and specs; WWW webpage:
http://www.ni.com/pdf/products/us/4daqsc202-204_ETCx2_212_213.pdf [accessed May 2004]

Nissanke, N. (1997) "*Realtime Systems*", Prentice-Hall.

Phatrapornnant, T. and Pont, M.J. (in press) "Reducing jitter in embedded systems employing a time-triggered software architecture and dynamic voltage scaling" to appear in: IEEE Transactions on Computers (Special Issue on Design and Test of Systems-On-a-Chip), May 2006.

Philips Semiconductors (2004), "LPC2119/2129/2194/2292/2294; Single-chip 32-bit microcontrollers user manual",

- Pont, M.J. (2001) "Patterns for Time-Triggered Embedded Systems: Building Reliable Applications with the 8051 Family of Microcontrollers", Addison-Wesley / ACM Press. ISBN: 0-201-331381.
- Pont, M.J. (2002) "Embedded C", Addison-Wesley. ISBN: 0-201-79523-X.
- Short, M. and Pont, M.J. (2005) "Hardware in the loop simulation of embedded automotive control systems", in Proceedings of the 8th IEEE International Conference on Intelligent Transportation Systems (IEEE ITSC 2005) held in Vienna, Austria, 13-16 September 2005, pp. 226-231.
- Torngren, M. (1998) "Fundamentals of implementing real-time control applications in distributed computer systems", Real-Time Systems, vol.14, pp.219-250.
- Ward, N. J. (1991) "The static analysis of a safety-critical avionics control system", in Corbyn, D.E. and Bray, N. P. (Eds.) "*Air Transport Safety: Proceedings of the Safety and Reliability Society Spring Conference, 1991*" Published by SaRS, Ltd.

Building reliable embedded systems using Abstract Patterns, Patterns, and Pattern Implementation Examples

Susan Kurian and Michael J. Pont

Embedded Systems Laboratory, University of Leicester, University Road, LEICESTER LE1 7RH, UK

Abstract

We have previously described a “language” consisting of more than seventy patterns. This language is intended to support the development of reliable embedded systems: the particular focus of the collection is on systems with a time triggered co-operative (TTC) system architecture. We have been assembling this collection for almost a decade. As our experience with the collection has grown, we have begun to add a number of new patterns and revised some of the existing ones. As we have worked with this collection, we have felt that there were ways in which the overall architecture could be improved in order to make the collection easier to use, and to reduce the impact of future changes. This paper briefly describes the approach that we have taken in order to re-factor and refine our original pattern collection. It goes on to describe some of the new and revised patterns that have resulted from this process.

Acknowledgements

This work is supported by an ORS award to Susan Kurian from the UK Government (Department for Education and Skills) and by the University of Leicester. An early version of this paper was published as: Pont, M.J., Kurian, S., Maaita, A. and Ong, R. (2005) “Restructuring a pattern language for reliable embedded systems”, ESL Technical Report 2005-01. Work on this paper was carried out while Michael J. Pont was on Study Leave from the University of Leicester.

1. Introduction

We have previously described a “language” consisting of more than seventy patterns, which will be referred to here as the “PTTES Collection” (see Pont, 2001). This language is intended to support the development of reliable embedded systems: the particular focus of the collection is on systems with a time triggered, co-operatively scheduled (TTCS) system architecture. Work began on these patterns in 1996, and they have since been used in a range of industrial systems, numerous university research projects, as well as in undergraduate and postgraduate teaching on many university courses (e.g. see Pont, 2003; Pont and Banner, 2004).

As our experience with the collection has grown, we have begun to add a number of new patterns and revised some of the existing ones (e.g. see Pont and Ong, 2003; Pont *et al.*, 2004; Key *et al.*, 2004). Inevitably, by definition, a pattern language consists of an inter-related set of components: as a result, it is unlikely that it will ever be possible to refine or extend such a system without causing some side effects. However, as we have worked with this collection, we have felt that there were ways in which the overall architecture could be improved in order to make the collection easier to use, and to reduce the impact of future changes.

This paper briefly describes the approach that we have taken in order to re-factor and refine our original pattern collection. It goes on to describe some of the new and revised patterns that have resulted from this process.

2. The new structure

In the original PTTES collection, we labelled all parts of the collection as “patterns”. We now believe it is more appropriate to divide the collection as follows:

- Abstract patterns
- Patterns, and,
- Pattern implementation examples

In this new structure, the “abstract patterns” are intended to address common design decisions faced by developers of embedded systems. Such patterns do not – directly – tell the user how to construct a piece of software or hardware: instead they are intended to help a developer

decide whether use of a particular design solution (perhaps a hardware component, a software algorithm, or some combination of the two) would be an appropriate way of solving a particular design challenge. Note that the problem statements for these patterns typically begin with the phrase “Should you use a ...” (or something similar).

For example, in this report, we present the abstract pattern TTC PLATFORM. This pattern describes what a time-triggered co-operative (TTC) scheduler is, and discusses situations when it would be appropriate to use such an architecture in a reliable embedded system. If you decide to use a TTC architecture, then you have a number of different implementation options available: these different options have varying resource requirements and performance figures. The patterns TTC-SL SCHEDULER, TTC- ISR SCHEDULER and TTC SCHEDULER describe some of the ways in which a TTC PLATFORM can be implemented. While documenting each of these “full” patterns, we refer back to the abstract pattern for background information.

We take this layered approach one stage further with what we call “pattern implementation examples” (PIEs). As the name might suggest, PIEs are intended to illustrate how a particular pattern can be implemented. This is important (in our field) because there are great differences in system environments, caused by variations in the hardware platform (e.g. 8-bit, 16-bit, 32-bit, 64-bit), and programming language (e.g. assembly language, C, C++). The possible implementations are not sufficiently different to be classified as distinct patterns: however, they do contain useful information.

Note that, as an alternative to the use of PIEs, we could simply extend each pattern with a large numbers of examples. However, this would make the pattern bulky, and difficult to use. In addition, new devices appear with great frequency in the embedded sector. By having distinct PIEs, we can add new implementation descriptions when these are useful, without revising the entire pattern each time we do so.

3. Overview of this paper

This paper presents the abstract pattern TTC PLATFORM. This is followed by a pattern (TTC-SL SCHEDULER). We also present a pattern implementation example (TTC-SL SCHEDULER [C, C167]).

Figure 1 provides a schematic representation of the different pattern layers discussed in this paper.

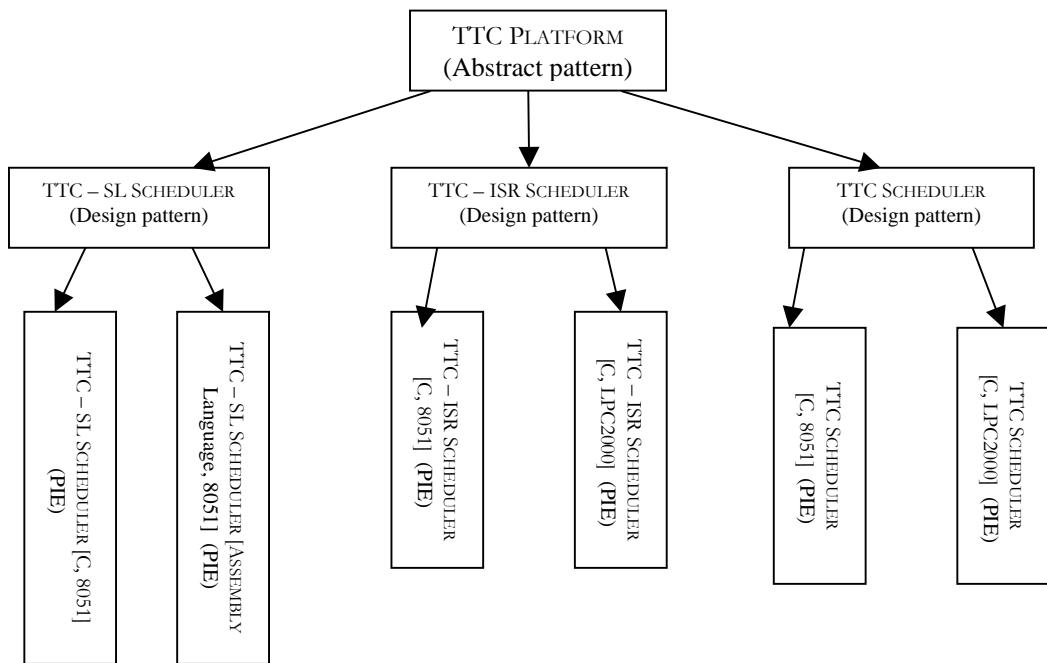


Figure 1: The three different types of pattern discussed in this paper

Context

- You are developing an embedded system.
- Reliability is a key design requirement.

Problem

Should you use a time-triggered co-operative (TTC) scheduler as the basis of your embedded system?

Background

This pattern is concerned with systems which have at their heart a time-triggered co-operative (TTC) scheduler. We will be concerned both with “pure” TTC designs - sometimes referred to as “cyclic executives” (e.g. Baker and Shaw, 1989; Locke, 1992; Shaw, 2001) – as well as “hybrid” TTC designs (e.g. Pont, 2001; Pont, 2004), which include a single pre-emptive task.

We provide some essential background material and definitions in this section.

Tasks

Tasks are the building blocks of embedded systems. A task is simply a labeled segment of program code: in the systems we will be concerned with in this pattern a task will generally be implemented using a C function¹.

Most embedded systems will be assembled from collections of tasks. When developing systems, it is often helpful to divide these tasks into two broad categories:

- *Periodic* tasks will be implemented as functions which are called – for example – every millisecond or every 100 milliseconds during some or all of the time that the system is active.
- *Aperiodic* tasks will be implemented as functions which may be activated if a particular event takes place. For example, an aperiodic task might be activated when a switch is pressed, or a character is received over a serial connection.

Please note that the distinction between calling a periodic task and activating an aperiodic task is significant, because of the different ways in which events may be handled. For

example, we might design the system in such a way that the arrival of a character via a serial (e.g. RS-232) interface will generate an interrupt, and thereby call an interrupt service routine (an “ISR task”). Alternatively, we might choose to design the system in such a way that a hardware flag is set when the character arrives, and use a periodic task “wrapper” to check (or poll) this flag: if the flag is found to be set, we can then call an appropriate task

Basic timing constraints

For both types of tasks, timing constraints are often a key concern. We will use the following loose definitions in this pattern:

- A task will be considered to have **soft timing (ST) constraints** if its execution ≥ 1 second late (or early) may cause a significant change in system behaviour.
- A task will be considered to have **firm timing (FT) constraints** if its execution ≥ 1 millisecond late (or early) may cause a significant change in system behaviour.
- A task will be considered to have **hard timing (ST) constraints** if its execution ≥ 1 microsecond late (or early) may cause a significant change in system behaviour.

Thus, for example, we might have a FT periodic task that is due to execute at times $t = \{0 \text{ ms}, 1000 \text{ ms}, 2000 \text{ ms}, 3000 \text{ ms}, \dots\}$. If the task executes at times $t = \{0 \text{ ms}, 1003 \text{ ms}, 2000 \text{ ms}, 2998 \text{ ms}, \dots\}$ then the system behaviour will be considered unacceptable.

Jitter

For some periodic tasks, the absolute deadline is less important than variations in the timing of activities. For example, suppose that we intend that some activity should occurs at times:

$$t = \{1.0 \text{ ms}, 2.0 \text{ ms}, 3.0 \text{ ms}, 4.0 \text{ ms}, 5.0 \text{ ms}, 6.0 \text{ ms}, 7.0 \text{ ms}, \dots\}.$$

Suppose, instead, that the activity occurs at times:

$$t = \{11.0 \text{ ms}, 12.0 \text{ ms}, 13.0 \text{ ms}, 14.0 \text{ ms}, 15.0 \text{ ms}, 16.0 \text{ ms}, 17.0 \text{ ms}, \dots\}.$$

In this case, the activity has been delayed (by 10 ms). **For some applications – such as data, speech or music playback, for example – this delay may make no measurable difference to the user of the system.**

¹ A task implemented in this way does not need to be a “leaf” function: that is, a task may call (other) functions.

However, suppose that – for a data playback system - same activities were to occur as follows:

$$t = \{1.0 \text{ ms}, 2.1 \text{ ms}, 3.0 \text{ ms}, 3.9 \text{ ms}, 5.0 \text{ ms}, 6.1 \text{ ms}, 7.0 \text{ ms}, \dots\}.$$

In this case, there is a variation (or jitter) in the task timings. Jitter can have a very detrimental impact on the performance of many applications, particularly those involving period sampling and / or data generation (such as data acquisition, data playback and control systems: see Torngren, 1998).

For example, Cottet and David (1999) show that – during data acquisition tasks – jitter rates of 10% or more can introduce errors which are so significant that any subsequent interpretation of the sampled signal may be rendered meaningless. Similarly Jerri (1977) discuss the serious impact of jitter on applications such as spectrum analysis and filtering. Also, in control systems, jitter can greatly degrade the performance by varying the sampling period (Torgren, 1998; Mart et al., 2001).

Transactions

Most systems will consist of several tasks (a large system may have hundreds of tasks, possibly distributed across a number of CPUs). Whatever the system, tasks are rarely independent: for example, we often need to exchange data between tasks. In addition, more than one task (on the same processor) may need to access shared components such as ports, serial interfaces, digital-to-analogue converters, and so forth. The implication of this type of link between tasks varies depending on the method of scheduling that is employed: we discuss this further shortly.

Another important consideration is that tasks are often linked in what are sometimes called *transactions*. Transactions are sequences of tasks which must be invoked in a specific order. For example, we might have a task that records data from a sensor (`TASK_Get_Data()`), and a second task that compresses the data (`TASK_Compress_Data()`), and a third task that stores the data on a Flash disk (`TASK_Store_Data()`). Clearly we cannot compress the data before we have acquired it, and we cannot store the data before we have compressed it: we must therefore always call the tasks in the same order:

```
TASK_Get_Data()  
TASK_Compress_Data()  
TASK_Store_Data()
```

When a task is included in a transaction it will often inherit timing requirements. For example, in the case of our data storage system, we might have a requirement that the data are acquired every 10 ms. This requirement will be inherited by the other tasks in the transaction, so that all three tasks must complete within 10 ms.

Scheduling tasks

As we have noted, most systems involve more than one task. For many projects, a key challenge is to work out how to schedule these tasks so as to meet all of the timing constraints.

The scheduler we use can take two forms: co-operative and pre-emptive. The difference between these two forms is - superficially – rather small but has very large implications for our discussions in this pattern. We will therefore look closely at the differences between co-operative and pre-emptive scheduling.

To illustrate this distinction, suppose that – over a particular period of time – we wish to execute four tasks (Task A, Task B, Task C, Task D) as illustrated in Figure 2.

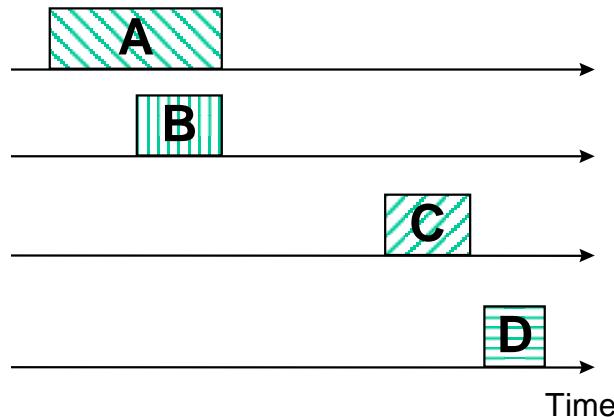


Figure 2: A schematic representation of four tasks (Task A, Task B, Task C, Task D) which we wish to schedule for execution in an embedded system with a single CPU.

We assume that we have a single processor. As a result, what we are attempting to achieve is shown in Figure 3.



Figure 3: Attempting the impossible: Task A and Task B are scheduled to run simultaneously.

In this case, we can run Task C and Task D as required. However, Task B is due to execute before Task A is complete. Since we cannot run more than one task on our single CPU, one of the tasks has to relinquish control of the CPU at this time.

In the simplest solution, we schedule Task A and Task B *co-operatively*. In these circumstances we (implicitly) assign a high priority to any task which is currently using the CPU: any other task must therefore wait until this task relinquishes control before it can execute. In this case, Task A will complete and then Task B will be executed (Figure 4).

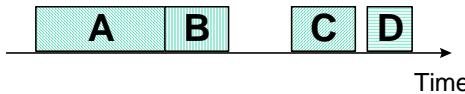


Figure 4: Scheduling Task A and Task B co-operatively.

Alternatively, we may choose a *pre-emptive* solution. For example, we may wish to assign a higher priority to Task B with the consequence that – when Task B is due to run – Task A will be interrupted, Task B will run, and Task A will then resume and complete (Figure 5).

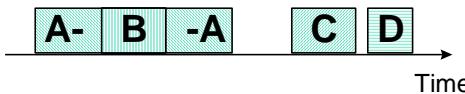


Figure 5: Assigning a high priority to Task B and scheduling the two tasks pre-emptively.

A closer look at co-operative vs. pre-emptive architectures

When compared to pre-emptive schedulers, co-operative schedulers have a number of desirable features, particularly for use in safety-related systems (Allworth, 1981; Ward, 1991; Nissanke, 1997; Bate, 2000). For example, Nissanke (1997, p.237) notes: “[Pre-emptive] schedules carry greater runtime overheads because of the need for context switching - storage and retrieval of partially computed results. [Co-operative] algorithms do not incur such overheads. Other advantages of [co-operative] algorithms include their better understandability, greater predictability, ease of testing and their inherent capability for

guaranteeing exclusive access to any shared resource or data.”. Allworth (1981, p.53-54) notes: “*Significant advantages are obtained when using this [co-operative] technique. Since the processes are not interruptable, poor synchronisation does not give rise to the problem of shared data. Shared subroutines can be implemented without producing re-entrant code or implementing lock and unlock mechanisms*”. Also, Bate (2000) identifies the following four advantages of co-operative scheduling, compared to pre-emptive alternatives: [1] The scheduler is simpler; [2] The overheads are reduced; [3] Testing is easier; [4] Certification authorities tend to support this form of scheduling.

This matter has also been discussed in the field of distributed systems, where a range of different network protocols have been developed to meet the needs of high-reliability systems (e.g. see Kopetz, 2001; Hartwich *et al.*, 2002). More generally, Fohler has observed that: “Time triggered real-time systems have been shown to be appropriate for a variety of critical applications. They provide verifiable timing behavior and allow distribution, complex application structures, and general requirements.” (Fohler, 1999).

Solution

This pattern is intended to help answer the question: “Should you use a time-triggered co-operative (TTC) scheduler as the basis for your reliable embedded system?”

In this section, we will argue that the short answer to this question is “yes”. More specifically, we will explain how you can determine whether a TTC architecture is appropriate for your application, and – for situations where such an architecture is inappropriate – we will describe ways in which you can extend the simple TTC architecture to introduce limited degrees of pre-emption into the design.

Overall, our argument will be that – to maximise the reliability of your design – you should use the simplest “appropriate architecture”, and only employ the level of pre-emption that is essential to the needs of your application.

When is it appropriate (and not appropriate) to use a pure TTC architecture?

Pure TTC architectures are a good match for a wide range of applications. For example, we have previously described in detail how these techniques can be in – for example - data acquisition systems, washing-machine control and monitoring of liquid flow rates (Pont, 2002), in various automotive applications (e.g. Ayavoo *et al.*, 2004), a wireless (ECG)

monitoring system (Phatrapornnant and Pont, 2004), and various control applications (e.g. Edwards et al., 2004; Key et al., 2004).

Of course, this architecture not always appropriate. The main problem is that long tasks will have an impact on the responsiveness of the system. This concern is succinctly summarised by Allworth: “[*The*] main drawback with this [*co-operative*] approach is that while the current process is running, the system is not responsive to changes in the environment. Therefore, system processes must be extremely brief if the real-time response [of the] system is not to be impaired.” (Allworth, 1981).

We can express this concern slightly more formally by noting that if the system must execute one or more tasks of duration X and also respond within an interval T to external events (where $T < X$), a pure co-operative scheduler will not generally be suitable.

In practice, it is sometimes assumed that TTC architecture is inappropriate because some simple design options have been overlooked. We will use two examples to try and illustrate how – with appropriate design choices – we can meet some of the challenges of TTC development.

Example: Multi-stage tasks

Suppose we wish to transfer data to a PC at a standard 9600 baud; that is, 9600 bits per second. Transmitting each byte of data, plus stop and start bits, involves the transmission of 10 bits of information (assuming a single stop bit is used). As a result, each byte takes approximately 1 ms to transmit.

Now, suppose we wish to send this information to the PC:

```
Current core temperature is 36.678 degrees
```

If we use a standard function (such as some form of `printf()`) - the task sending these 42 characters will take more than 40 milliseconds to complete. If this time is greater than the system tick interval (often 1 ms, rarely greater than 10 ms) then this is likely to present a problem (Figure 6).

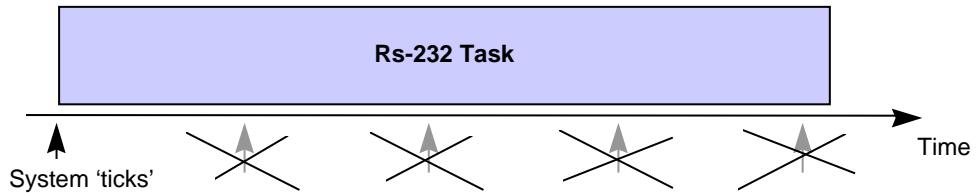


Figure 6: A schematic representation of the problems caused by sending a long character string on an embedded system with a simple operating system. In this case, sending the message takes 42 ms while the OS tick interval is 10 ms.

Perhaps the most obvious way of addressing this issue is to increase the baud rate; however, this is not always possible, and - even with very high baud rates - long messages or irregular bursts of data can still cause difficulties.

A complete solution involves a change in the system architecture. Rather than sending all of the data at once, we store the data we want to send to the PC in a buffer (Figure 7). Every ten milliseconds (say) we check the buffer and send the next character (if there is one ready to send). In this way, all of the required 43 characters of data will be sent to the PC within 0.5 seconds. This is often (more than) adequate. However, if necessary, we can reduce this time by checking the buffer more frequently. Note that because we do not have to wait for each character to be sent, the process of sending data from the buffer will be very fast (typically a fraction of a millisecond).

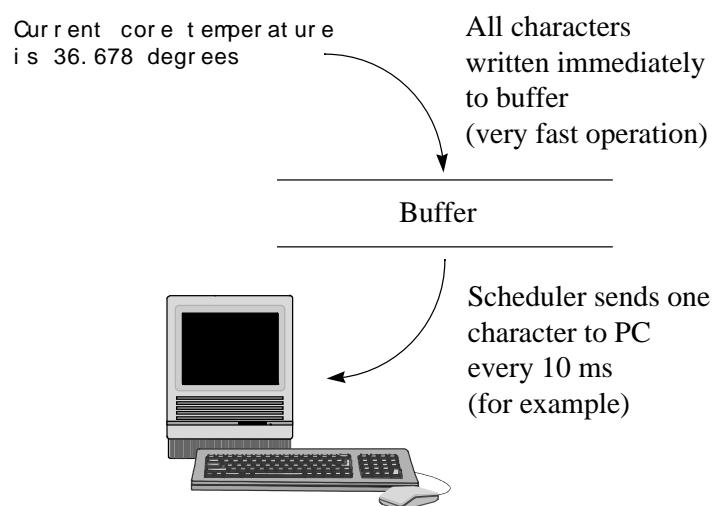


Figure 7: A schematic representation of the software architecture used in the RS-232 library.

This is an example of an effective solution to a widespread problem. The problem is discussed in more detail in the pattern MULTI-STAGE TASK [Pont, 2001].

Example: Rapid data acquisition

The previous example involved sending data to the outside world. To solve the design problem, we opted to send data at a rate of one character every millisecond. In many cases, this type of solution can be effective.

Consider another problem (again taken from a real design). This time suppose we need to receive data from an external source over a serial (RS-232) link. Further suppose that these data are to be transmitted as a packet, 100 ms long, at a baud rate of 115 kbaud. One packet will be sent every second for processing by our embedded system.

At this baud rate, data will arrive approximately every 87 μ s. To avoid losing data, we would – if we used the architecture outlined in the previous example – need to have a system tick interval of around 40 μ s. This is a short tick interval, and would only produce a practical TTC architecture if a powerful processor was used.

However, a pure TTC architecture may still be possible, as follows. First, we set up an ISR, set to trigger on receipt of UART interrupts:

```
void UART_ISR(void)
{
    // Get first char

    // Collect data for 100 ms (with timeout)
}
```

These interrupts will be received roughly once per second, and the ISR will run for 100 ms.

When the ISR ends, processing continues in the main loop:

```
void main(void)
{
    ...

    while(1)
    {
        Process_UART_Data();
        Go_To_Sleep();
    }
}
```

Here we have up to 0.9 seconds to process the UART data, before the next tick.

What should you do if a pure TTC architecture cannot meet your application needs?
In the previous two examples, we could produce a clean TTC system with appropriate design.

This is – of course – not always possible. For example, consider a wireless electrocardiogram (ECG) system (Figure 8).

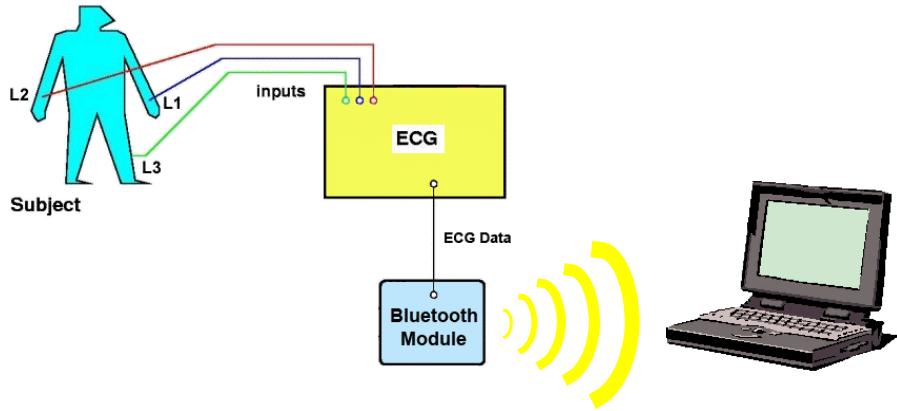


Figure 8: A schematic representation of a system for ECG monitoring.
See Phatrapornnant and Pont (2004) for details.

An ECG is an electrical recording of the heart that is used for investigating heart disease. In a hospital environment, ECGs normally have 12 leads (standard leads, augmented limb leads and precordial leads) and can plot 250 sample-points per second (at minimum). In the portable ECG system considered here, three standard leads (Lead I, Lead II, and Lead III) were recorded at 500 Hz. The electrical signal were sampled using a (12-bit) ADC and – after compression – the data were passed to a “Bluetooth” module for transmission to a notebook PC, for analysis by a clinician (see Phatrapornnant and Pont, 2004)

In one version of this system, we are required to perform the following tasks:

- Sample the data continuously at a rate of 500 Hz. Sampling takes less than 0.1 ms.
- When we have 10 samples (that is, every 20 ms), compress and transmit the data, a process which takes a total of 6.7 ms.

In this case, we will assume that the compression task cannot be neatly decomposed into a sequence of shorter tasks, and we therefore cannot employ a pure TTC architecture.

In such circumstances, it is tempting to opt immediately for a full pre-emptive design. Indeed, many studies seem to suggest that this is the only alternative. For example, Locke (1992) - in a widely cited publication - suggests that *“traditionally, there have been two basic approaches to the overall design of application systems exhibiting hard real-time deadlines:*

the cyclic executive ... and the fixed priority [pre-emptive] architecture.” (p.37). Similarly, Bennett (1994, p.205) states: “*If we consider the scheduling of time allocation on a single CPU there are two basic alternatives: [1] cyclic, [2] pre-emptive.*” More recently Bate (1998) compared cyclic executives and fixed-priority pre-emptive schedulers (exploring, in greater depth, Locke’s study from a few years earlier).

However, even if you cannot – cleanly - solve the long task / short response time problem, then you can maintain the core co-operative scheduler, and add only the limited degree of pre-emption that is required to meet the needs of your application.

For example, in the case of our ECG system, we can use a time-triggered hybrid architecture.

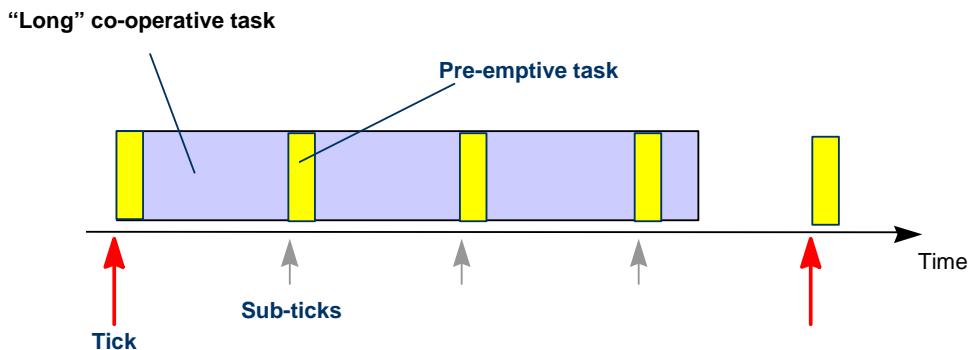


Figure 9: A “hybrid” software architecture. See text for details.

In this case, we allow a single pre-emptive task to operate: in our ECG system, this task will be used for data acquisition. This is a time-triggered task, and such tasks will generally be implemented as a function call from the timer ISR which is used to drive the core TTC scheduler. As we have discussed in detail elsewhere (Pont, 2001: Chapter 17) this architecture is extremely easy to implement, and can operate with very high reliability. As such it is one of a number of architectures, based on a TTC scheduler, which are co-operatively based, but also provide a controlled degree of pre-emption.

As we have noted, most discussions of scheduling tend to overlook these “hybrid” architectures in favour of fully pre-emptive alternatives. When considering this issue, it cannot be ignored that the use of (fully) pre-emptive environments can be seen to have clear commercial advantages for some companies. For example, a co-operative scheduler may be easily constructed, entirely in a high-level programming language, in around 300 lines of ‘C’

code. The code is highly portable, easy to understand and to use and is, in effect, freely available. By contrast, the increased complexity of a pre-emptive operating environment results in a much larger code framework (some ten times the size, even in a simple implementation: Labrosse 1992). The size and complexity of this code makes it unsuitable for ‘in house’ construction in most situations, and therefore provides the basis for commercial ‘RTOS’ products to be sold, generally at high prices and often with expensive run-time royalties to be paid. The continued promotion and sale of such environments has, in turn, prompted further academic interest in this area. For example, according to Liu and Ha, (1995): “[An] objective of reengineering is the adoption of commercial off-the-shelf and standard operating systems. Because they do not support cyclic scheduling, the adoption of these operating systems makes it necessary for us to abandon this traditional approach to scheduling.”

Related patterns and alternative solutions

We highlight some related patterns and alternative solutions in this section.

Implementing a TTC Scheduler

The following patterns describe different ways of implementing a TTC PLATFORM:

- TTC – SL SCHEDULER
- TTC-ISR SCHEDULER
- TTC SCHEDULER

Alternatives to TTC scheduling

If you are determined to implement a fully pre-emptive design, then Jean Labrosse (1999) and Anthony Massa (2003) discuss – in detail – the construction of such systems.

Reliability and safety implications

For reasons discussed in detail in the previous sections of this pattern, co-operative schedulers are generally considered to be a highly appropriate platform on which to construct a reliable (and safe) embedded system.

Overall strengths and weaknesses

- ☺ Tends to result in a system with highly predictable patterns of behaviour.
- ☹ Inappropriate system design using this approach can result in applications which have a comparatively slow response to external events.

Further reading

- Allworth, S.T. (1981) "An Introduction to Real-Time Software Design", Macmillan, London.
- Ayavoo, D., Pont, M.J. and Parker, S. (2004) "Using simulation to support the design of distributed embedded control systems: A case study". In: Koelmans, A., Bystrov, A. and Pont, M.J. (Eds.) Proceedings of the UK Embedded Forum 2004 (Birmingham, UK, October 2004). Published by University of Newcastle.
- Baker, T.P. and Shaw, A. (1989) "The cyclic executive model and Ada", Real-Time Systems, 1(1): 7-25.
- Bate, I.J. (1998) "Scheduling and timing analysis for safety critical real-time systems", PhD thesis, University of York, UK.
- Bate, I.J. (2000) "Introduction to scheduling and timing analysis", in "The Use of Ada in Real-Time System" (6 April, 2000). IEE Conference Publication 00/034.
- Bennett, S. (1994) "Real-Time Computer Control" (Second Edition) Prentice-Hall.
- Cottet, F. and David, L. (1999) "A solution to the time jitter removal in deadline based scheduling of real-time applications", 5th IEEE Real-Time Technology and Applications Symposium - WIP, Vancouver, Canada, pp. 33-38.
- Edwards, T., Pont, M.J., Scotson, P. and Crumpler, S. (2004) "A test-bed for evaluating and comparing designs for embedded control systems". In: Koelmans, A., Bystrov, A. and Pont, M.J. (Eds.) Proceedings of the UK Embedded Forum 2004 (Birmingham, UK, October 2004). Published by University of Newcastle.
- Fohler, G. (1999) "Time Triggered vs. Event Triggered - Towards Predictably Flexible Real-Time Systems", Keynote Address, Brazilian Workshop on Real-Time Systems, May 1999.
- Hartwich F., Muller B., Fuhrer T., Hugel R., Bosh R. GmbH, (2002), Timing in the TTCAN Network, Proceedings 8th International CAN Conference.
- Jerri, A.J. (1977) "The Shannon sampling theorem: its various extensions and applications a tutorial review", Proc. of the IEEE, vol. 65, n° 11, p. 1565-1596.
- Key, S. and Pont, M.J. (2004) "Implementing PID control systems using resource-limited embedded processors". In: Koelmans, A., Bystrov, A. and Pont, M.J. (Eds.) Proceedings of the UK Embedded Forum 2004 (Birmingham, UK, October 2004). Published by University of Newcastle.
- Kopetz, H. (1997) "Real-time systems: Design principles for distributed embedded applications", Kluwer Academic.
- Labrosse, J. (1999) "MicroC/OS-II: The real-time kernel", CMP books. ISBN: 0-87930-543-6.
- Liu, J.W.S. and Ha, R. (1995) "Methods for validating real-time constraints", *Journal of Systems and Software*.
- Locke, C.D. (1992) "Software architecture for hard real-time systems: Cyclic executives vs. Fixed priority executives", *The Journal of Real-Time Systems*, 4: 37-53.
- Mart, P., Fuertes, J. M., Villt, R. and Fohler, G. (2001), "On Real-Time Control Tasks Schedulability", European Control Conference (ECC01), Porto, Portugal, pp. 2227-2232.

- Massa, A.J. (2003) "Embedded Software Development with eCOS", Prentice Hall. ISBN: 0-13-035473-2.
- Nissanke, N. (1997) "*Realtime Systems*", Prentice-Hall.
- Phatrapornnant, T. and Pont, M.J. (2004) "The application of dynamic voltage scaling in embedded systems employing a TTCS software architecture: A case study", Proceedings of the IEE / ACM Postgraduate Seminar on "System-On-Chip Design, Test and Technology", Loughborough, UK, 15 September 2004. Published by IEE. ISBN: 0 86341 460 5 (ISSN: 0537-9989)
- Pont, M.J. (2001) "Patterns for Time-Triggered Embedded Systems: Building Reliable Applications with the 8051 Family of Microcontrollers", Addison-Wesley / ACM Press. ISBN: 0-201-331381.
- Pont, M.J. (2002) "Embedded C", Addison-Wesley. ISBN: 0-201-79523-X.
- Pont, M.J. (2004) "A "Co-operative First" approach to software development for reliable embedded systems", invited presentation at the UK Embedded Systems Show, 13-14 October, 2004. Presentation available here: www.le.ac.uk/eg/embedded
- Proctor, F. M. and Shackleford, W. P. (2001), "Real-time Operating System Timing Jitter and its Impact on Motor Control", proceedings of the 2001 SPIE Conference on Sensors and Controls for Intelligent Manufacturing II, Vol. 4563-02.
- Shaw, A.C. (2001) "Real-time systems and software" John Wiley, New York.
[ISBN 0-471-35490-2]
- Torngren, M. (1998) "Fundamentals of implementing real-time control applications in distributed computer systems", Real-Time Systems, vol.14, pp.219-250.
- Ward, N. J. (1991) "The static analysis of a safety-critical avionics control system", in Corbyn, D.E. and Bray, N. P. (Eds.) "*Air Transport Safety: Proceedings of the Safety and Reliability Society Spring Conference, 1991*" Published by SaRS, Ltd.

Context

- You have decided that a TTC PLATFORM will provide an appropriate basis for your embedded system.

and

- Your application will have a single periodic task (or a single transaction).
- Your task / transaction has soft or firm constraints.
- There is no risk of task overruns (or occasional overruns can be tolerated).
- You need to use a minimum of CPU and memory resources.

Problem

How can you implement a TTC PLATFORM which meets the above requirements?

Background

See TTC PLATFORM for relevant background information.

Solution

A TTC-SL Scheduler allows us to schedule a single periodic task. To implement such a scheduler, we need to do the following:

1. Determine the task period (that is, the interval between task executions).
2. Determine the worst case execution time (WCET) of the task.
3. The required delay value is task period – WCET.
4. Choose an appropriate delay function (e.g. SOFTWARE DELAY or HARDWARE DELAY: Pont, 2001) that meets the delay requirements.
5. Implement a suitable SUPER LOOP (Pont, 2001) containing a task call and a delay call.

For example, suppose that we wish to flash an LED on and off at a frequency of 0.5 Hz (that is, on for one second, off for one second, etc). Further suppose that we have a function - `LED_Flash_Update()` – that changes the LED state every time it is called.

`LED_Flash_Update()` is the task we wish to schedule. It has a WCET of approximately 0, so we require a delay of 1000 ms. Listing 1 shows a TTC-SL Scheduler framework which will allow us to schedule this task as required.

```

#include "Main.h"
#include "Loop_Del.h"
#include "LED_Flas.h"

void main(void)
{
    LED_Flash_Init();

    while (1)
    {
        LED_Flash_Update();
        Loop_Delay(1000);      // Delay 1000 ms
    }
}

```

Listing 1: Implementation of a TTC-SL SCHEDULER

Related patterns and alternative solutions

We highlight some related patterns and alternative solutions in this section.

- SOFTWARE DELAY
- HARDWARE DELAY
- TTC-ISR SCHEDULER
- TTC SCHEDULER

Reliability and safety implications

In this section we consider some of the key reliability and safety implications resulting from the use of this pattern.

Running multiple tasks

TTC-SL SCHEDULERS can be used to run multiple tasks with soft timing requirements. It is important that the worst-case execution time of each task is known before hand to set up the appropriate delay values.

Use of Idle mode and task jitter

The processor does not benefit from using idle mode. There is considerable jitter in scheduling tasks when using a SUPERLOOP in conjunction with a SOFTWARE DELAY.

What happens if a task overruns?

Task overruns are undesirable and can upset the proper functioning of the system.

Overall strengths and weaknesses

- 😊 Simple design, easy to implement
- 😊 Very small resource requirements
- 😢 Not sufficiently reliable for precise timing

- (⌚) Low energy efficiency, due to inefficient use of idle mode

Further reading

- Pont, M.J. (2001) "Patterns for Time-Triggered Embedded Systems: Building Reliable Applications with the 8051 Family of Microcontrollers", Addison-Wesley / ACM Press. ISBN: 0-201-331381.
- Pont, M.J. (2002) "Embedded C", Addison-Wesley. ISBN: 0-201-79523-X.
- Pont, M.J. (2004) "A "Co-operative First" approach to software development for reliable embedded systems", invited presentation at the UK Embedded Systems Show, 13-14 October, 2004. Presentation available here: www.le.ac.uk/eg/embedded

TTC-SL SCHEDULER [C, C167]

{pattern implementation example}

Context

- You wish to implement a TTC-SL SCHEDULER [this paper]
- Your chosen implementation language is C².
- Your chosen implementation platform is the C167 family of microcontrollers.

Problem

How can you implement a TTC-SL SCHEDULER for the C167 family of microcontrollers?

Background

-

Solution

Listing 2 shows a complete implementation of a “flashing LED” scheduler, based on the example in TTC-SL Scheduler (Solution section).

² The examples in the pattern were created using the Keil C compiler, hosted in a Keil uVision 3 IDE.

```

/*-----*
LED_167.C (7 November, 2001)

-----
Simple 'Flash LED' test function for C167 scheduler.

-*-----*/
#include "Main.h"
#include "Port.h"
#include "LED_167.h"

// ----- SFRs -----
sfr PICON = 0xF1C4;

// ----- Private variable definitions -----
static bit LED_state_G;

/*-----*
LED_Flash_Init()

- See below.
-*-----*/
void LED_Flash_Init(void)
{
    LED_state_G = 0;

    PICON = 0x0000;

    P2    = 0xFFFF; // set port data register
    ODP2 = 0x0000; // set port open drain control register
    DP2  = 0xFFFF; // set port direction register
}

/*-----*
LED_Flash_Update()

Flashes an LED (or pulses a buzzer, etc) on a specified port pin.

Must schedule at twice the required flash rate: thus, for 1 Hz
flash (on for 0.5 seconds, off for 0.5 seconds) must schedule
at 2 Hz.

-*-----*/
void LED_Flash_Update(void)
{
    // Change the LED from OFF to ON (or vice versa)
    if (LED_state_G == 1)
    {
        LED_state_G = 0;
        LED_pin0 = 0;
    }
    else
    {
        LED_state_G = 1;
        LED_pin0 = 1;
    }
}

```

Listing 2: Implementation of a simple task – LED_Flash_Update on C167 platform

```

//-----
// File: Delay.C (v1.00)
// Author: M.J.Pont
// Date: 05/12/2002
// Description: Simple hardware delays for C167.
// -----
#include "hardware_delay_167.h"

//-----
// Hardware_Delay()
//
// Function to generate N millisecond delay (approx).
// Uses Timer 2 in GPT1.
// -----
void Hardware_Delay(const tWord N)
{
    tWord ms;

    // Using GPT1 for hardware delay (Timer 2)
    T2CON = 0x0000;
    T3CON = 0x0000;

    // Delay value is *approximately* 1 ms per loop
    for (ms = 0; ms < N; ms++)
    {
        // 20 MHz, prescalar of 8
        T2 = 0xF63C; // Load timer 2 register

        T2IR = 0; // Clear overflow flag
        T2R = 1; // Start timer

        while (T2IR == 0); // Wait until timer overflows

        T2R = 0; // Stop timer
    }
}

```

Listing 3: Hardware Delay implemented on C167 platform.

```

void main(void)
{
    // Prepare for the 'Flash_LED' task
    LED_Flash_Init();

    while(1)
    {
        LED_Flash_Update();
        Hardware_Delay(1000);
    }
}

```

Listing 4: Using a simple SUPERLOOP architecture to schedule an LED_Flash_Update task.
The LED continuously flashes on for 1s and off for 1s

Further Reading

A ‘Hardware-in-the Loop’ testbed representing the operation of a cruise-control system in a passenger car

Devaraj Ayavoo¹, Michael J. Pont¹, Jianzhong Fang¹, Michael Short¹ and Stephen Parker²

¹*Embedded Systems Laboratory,
University of Leicester, University Road, Leicester LE1 7RH, UK*

²*Pi Technology,
Milton Hall, Ely Road, Milton, Cambridge CB4 6WZ, UK*

Abstract

The developer of a modern embedded system faces a bewildering range of design options. One way in which the impact of different design choices can be explored in a rigorous and controlled manner is through the use of appropriate hardware-in-the loop (HIL) simulator. HIL simulators – unlike software-only equivalents - allow studies to be carried out in real time, with real signals being measured. In this paper, we describe a HIL testbed that represents an automotive cruise-control system (CCS). A case study is used to illustrate how this testbed may be used to compare the different implementation options for single-processor and multi-processor system designs.

Acknowledgements

This work is supported by an ORS award (to DA) from the UK Government (Department for Education and Skills), by Pi Technology, by the Leverhulme Trust and by the University of Leicester. Work on this paper was completed while MJP was on Study Leave from the University of Leicester.

1. Introduction

The developer of a modern embedded system faces a bewildering range of design options. For example, the designer of a modern passenger car may need to choose between the use of one (or more) network protocols based on CAN (Rajnak and Ramnerfors, 2002), TTCAN (Hartwich et al., 2002), LIN (Specks and Rajnak, 2002), FlexRay or TTP/C (Kopetz, 2001). The resulting network may be connected in, for example, a bus or star topology (Tanenbaum, 1995). The individual processor nodes in the network may use event-triggered (Nissanke, 1997) or time-triggered (Kopetz, 1997) software architectures, or some combination of the two. The clocks associated with these processors may be linked using, for example, shared-clock techniques (Pont, 2001) or synchronisation messages (Hartwich et al., 2000). These individual processors may, for example, be C167 (Siemens, 1996), ARM (ARM, 2001), MPC555 (Bannatyne, 2003) or 8051 (Pont, 2001).

One way in which we can explore the impact of different design choices in a rigorous and controlled manner is through the use of appropriate hardware-in-the loop (HIL) simulators (see for example Hanselmann, 1996; Dynasim, 2003). HIL simulators – unlike software-only equivalents – allow studies to be carried out in real time, with real signals being measured.

The basic setup for the HIL simulator is illustrated in Figure 1, where the HIL simulation and the embedded system interconnect with each other by exchanging information through the necessary I/Os, such as digital, analogue and serial ports.

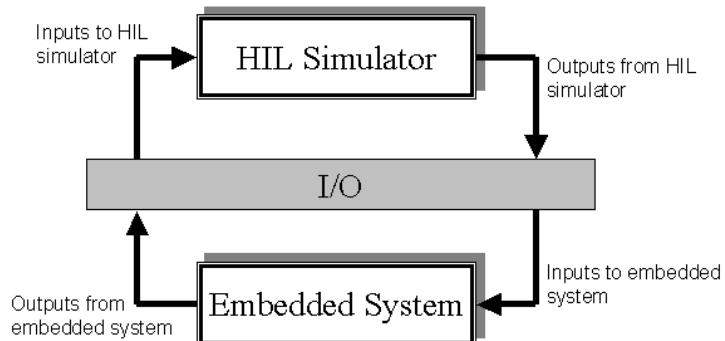


Figure 1: The HIL approach

We have recently described a detailed HIL simulation of an adaptive cruise-control system (ACCS) for a passenger car (Short and Pont, in press; Short et al., 2004a; Short et al., 2004b; Short et al., 2004c), and shown how this can be employed to assess new network protocols (Nahas et al., 2005).

The complexity of the full ACCS simulation ensures accurate results at the cost of system complexity. In some cases, it can be useful to be able to eliminate inappropriate design options more quickly through this use of a less detailed HIL simulation. To this end a simple (non-adaptive) cruise-control design was developed. The design, implementation and evaluation of this simple simulator is described in detail in this paper.

The paper is organised as follows. Section 2 to Section 7 introduce the cruise-control testbed and describe the model and implementation details. A case study that employs the testbed is then presented in Section 8. Our conclusions are presented in Section 9.

2. An overview of the CCS testbed

An automotive cruise-control system (CCS) is intended to provide the driver with an option of maintaining the vehicle at a desired speed without further intervention, by controlling the throttle (accelerator) setting. Such a driver assistance system can reduce the strain on the driver especially while travelling on long journeys.

Such a CCS will typically have the following features:

- i) An ON / OFF button to enable / disable the system.
- ii) An interface through which the driver can change the set speed while cruising.
- iii) Switches on the accelerator and brake pedals that can be used to disengage the CCS and return control to the driver.

For the purpose of our study, the specification of the CCS was simplified such that the vehicle was assumed to be always in “cruise” mode. While in cruise mode, a “speed dial” was available to allow the driver to dynamically change the car speed. The control process ensures that the vehicle would travel at the desired set speed.

3. The design of the car environment

As with any HIL systems, a simulation model (sometimes known as plant within the control engineering community) is required in order to represent the system to be controlled. In this case, a computational model was used to represent the car environment in which the CCS would operate (based on a model described in Pont, 2001). The core of the car environment is a simplified physical model based on Newton’s law of motion (see Figure 2). Please note that in this case, it is assumed that the vehicle is under the influence of only two forces, the torque exerted on the car engine and the frictional force that acts in the opposite direction to the motion.

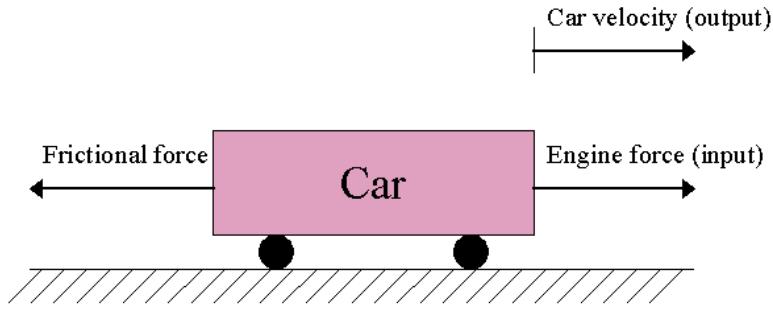


Figure 2: The car environment for the CCS

To model this mathematically, the summation of all the forces acting on the car is calculated, beginning with Newton's Second Law of Motion. The terms listed below are used in the equations that follow in this section:

A	Acceleration
v_i	Initial speed
v_f	Final speed
M	Mass of car
Δx	Displacement
θ	Throttle setting
Fr	Frictional coefficient
T	Engine torque

The frictional force and the engine force of the car are incorporated into Equation 1.

$$EngineForce - FrictionalForce = ma \quad (1)$$

The frictional force is a function of the velocity of the car, whereas the engine force is a product of the throttle setting and the engine torque. The engine torque is assumed to be constant over the speed range. The following model (Equation 2) is thus produced.

$$\theta\tau - v_i Fr = ma \quad (2)$$

This model is then used to determine the output of the car environment, which is the final velocity of the car (v_f). Solving for a , the instantaneous acceleration of the vehicle is first calculated (Equation 3).

$$a = (\theta\tau - v_i Fr) / m \quad (3)$$

Once this acceleration has been obtained, the distance travelled by the car (Δx) is solved using Equation 4.

$$\Delta x = v_i t + \frac{1}{2} a t^2 \quad (4)$$

The final speed of the car (v_f) is then determined using Equation 5.

$$v_f^2 = v_i^2 + 2a\Delta x \quad (5)$$

4. The implementation of the car environment

The car environment was implemented using a time-triggered co-operative scheduling architecture on a basic desktop PC (Intel Pentium II 300 MHz processor). The advantages of using PC hardware for such studies is described in detail elsewhere (see Pont et al., 2003).

Four main tasks were implemented as shown in Table 1. The source code was written and compiled using the Open Watcom C compiler¹.

Table 1: The car model task structure

Task Names	Task Description	Task Period (in ms)
Car Dynamics Update	Updates the car dynamics (speed) based on the input throttle position	5
Car Display Update	Displays the speed of the car and the throttle position on the monitor	100
PRM Update	Sends out the speed of the car as a train of pulses	1
Write To File	Records the speed and the throttle position of the car in a text file	1000

We wanted to keep the cost of this simulator as low as possible (so that it can be widely used). In order to access the PC hardware level, an operating system (OS) was required. DOS (Disk Operating System) was chosen because it offers the required flexibility at low cost². DOS in-turn control the BIOS (Basic Input Output System) of the PC to access the PC hardware level (Figure 3).

¹ The Watcom compiler can be downloaded (without charge) here: <http://www.openwatcom.org/>.

² Free versions of DOS are available. See for example <http://www.handyarchive.com/free/dos/>

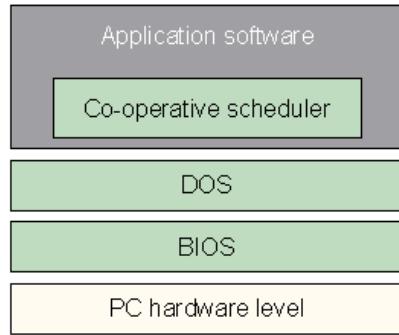


Figure 3: The operating layers for a PC hardware implementation

To interface the real world, a low-cost (but effective) option is to use the PC's parallel port. The parallel port has three registers: the data register, status register and control register (Messmer, 2002). In our version of the CCS, the port's data register (LPT1, 0x378) was used to store an 8-bit throttle position as the input signal to the car environment. The output signal from the car environment is the current speed of the car, represented as a train of pulses at the automatic line feed (ALF) pin of the port's control register (LPT1, 0x37A).

The connections are illustrated in Figure 4.

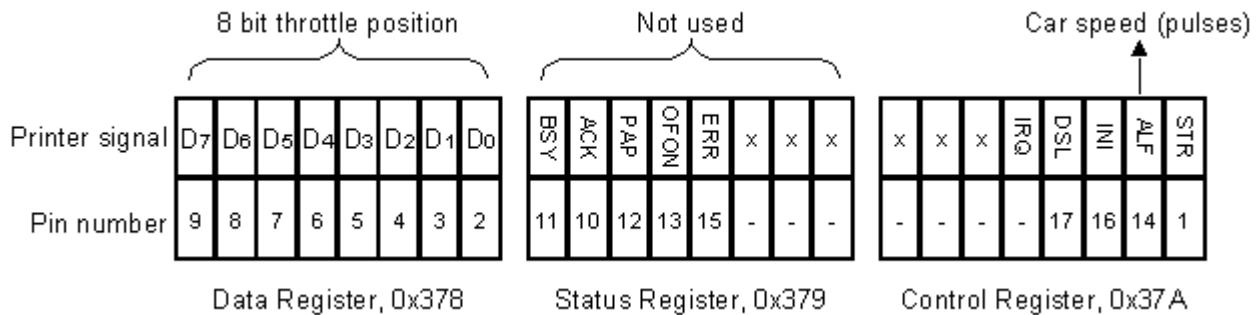


Figure 4: Connections on the PC's parallel port

Figure 5 shows a screenshot of the car environment model running on a desktop PC.

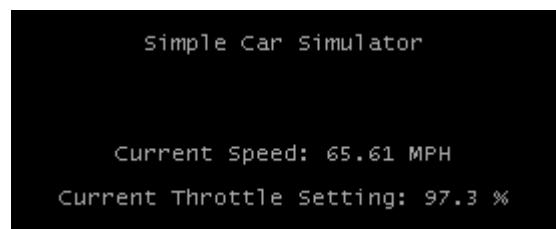


Figure 5: A screenshot of the CCS system

The source code for the car environment model is discussed in Appendix A. The complete source code is available from the Embedded Systems Laboratory website³.

5. The design of the controller

To control the velocity of the car at a set speed, a control algorithm was required. Two basic controllers can be chosen: open loop or closed loop.

In this case, a closed-loop control system was used since the output value from the car environment has an impact on the process input value to maintain the desired output value. A closed loop control system (as shown in Figure 6) sends the difference of the input signal (the desired value) and the feedback signal (actual output value) to the controller. The controller's job is to reduce this error (ideally to 0). To achieve this, a wide range of control algorithms are available (see, for example, Dorf and Bishop, 1998; Dutton et al., 1997).

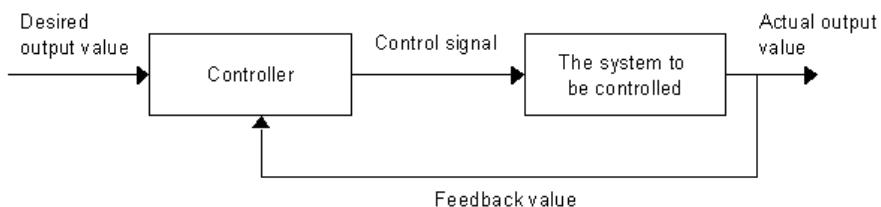


Figure 6: A closed-loop or feedback control system (Pont, 2001)

A “Proportional Integral Differential” (PID) controller was used in the CCS as it is a simple, common and effective choice (Ogata, 2002). The PID algorithm consists of three main parts: the proportional term (K_p), the integral term (K_i) and the derivative term (K_d) : please see Equation 4, where u and e represent the output signal and error signal, respectively.

$$u = K_p \times e + K_i \times \int e dt + K_d \times \frac{de}{dt} \quad (4)$$

The proportional term will have the effect of reducing the rise time. The integral term can eliminate the steady-state error, but it may make the transient response worse. The derivative term can be used to add “damping” to the response, in order to reduce the signal overshoot (Franklin et al., 1998).

³ <http://www.le.ac.uk/eg/embedded/SimpleCCS.htm>

6. Implementation of the controller

A typical control system can be divided into three main sections: sampler, control algorithm and actuator. In the first section, data are sampled from the environment model. In the second section these data are processed using an appropriate control algorithm. In the third section, an output signal is produced that will (generally) alter the system state.

In a single-processor system, all three functions will be carried out on the same node. In a distributed environment, these functions may be carried out on up to three nodes, linked by an appropriate network protocol. For example, a two-node design might carry out the sampling operations on Node 1, and the control and actuation operations on Node 2 (see, for example El-khoury and Törngren, 2001; Lonn and Axelsson, 1999).

Figure 7 and Figure 8 illustrate the implementation of a one-node and two-node CCS respectively⁴.

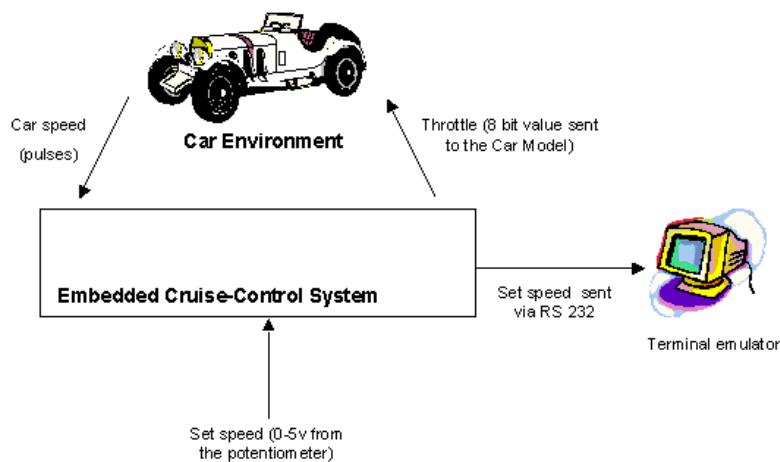


Figure 7: One-node CCS

⁴ Please note that the system could also be expanded to have more than two nodes to incorporate various design options such as bus guardians, back-up nodes and redundancy. These options are not considered here.

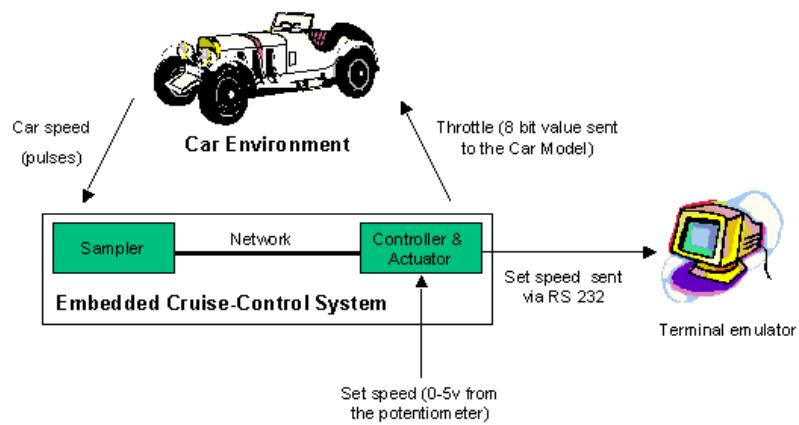


Figure 8: Two-node CCS

In both the one-node and two-node systems, the input to the CCS is the car speed (represented as a train of pulses from the car environment) and the desired set speed comes from a potentiometer. The output from the CCS is an 8-bit throttle position that is sent to the car environment and (to support the simulation) the desired set speed value that is sent via an RS232 link to a PC or similar terminal.

Figure 9 shows an example of the wiring involved for a one-node CCS on a C167 microcontroller.

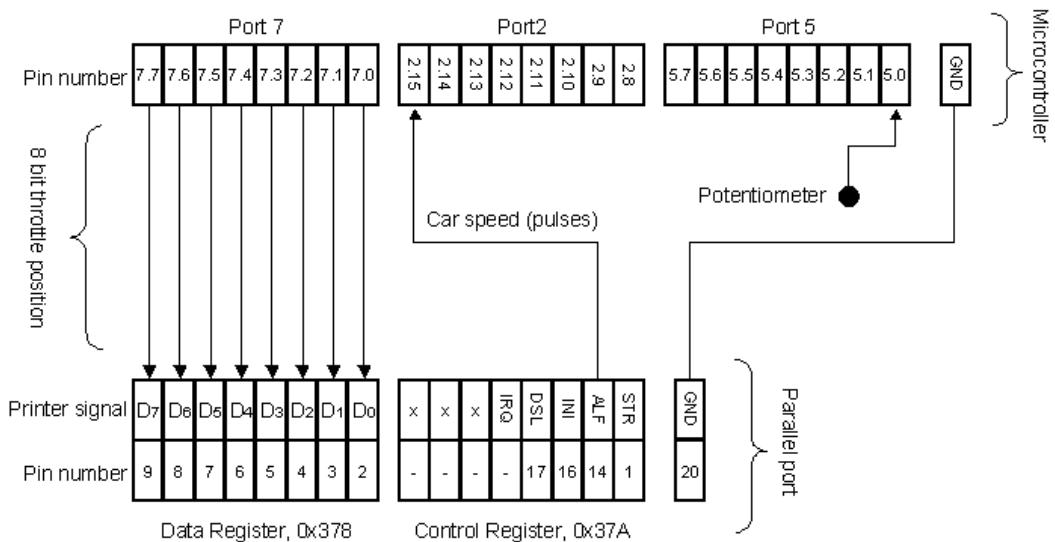


Figure 9: Wiring example for a one-node CCS system implemented using a C167 processor

7. The CCS tasks

Embedded software systems are often designed and implemented as a collection of communicating tasks (e.g. Nissanke, 1997; Shaw, 2001).

To implement the CCS, five tasks were employed (Table 2).

Table 2: The CCS task list

Task Names	Task Description	Task Period (in ms)
Compute Car Speed	Computes the car speed obtained from the car model	50
Compute Throttle	Calculates and sends the required throttle to be applied back to the car model	50
Get Ref Speed	Gets the desired speed from the driver	1000
PC Link Update	Sends a character to the serial port	10
Display Ref Speed	Updates the string that displays the desired car speed	1000

Each task is described in the subsections that follow.

a) Task: Compute Car Speed

This task acts as the signal sampler. The signal – in this case, the speed of the car – is represented as a train of pulses. To obtain the correct representation of the car speed, a hardware pulse counter is used to store the number of pulses that has arrived. This value is then filtered (using software) to remove any noise that may be present in the signal. The filtered value will then to be scaled to represent the current speed of the car.

A partial code listing for this task is show in Listing 1.

Please note that the processor chosen for this application must have at least two hardware timers – one for the periodic timer and one for the hardware counter.

```
void Sens_Compute_Speed(void)
{
    tWord raw_speed;

    raw_speed = Get_Raw_Speed();

    Scaled_speed_G = ((float)(FILTER_COEFF * Old_speed_G) + (float)
        ((1 - FILTER_COEFF) * (raw_speed * SCALING_FAC)));

    Old_speed_G = Scaled_speed_G;
}
```

Listing 1: An example of the compute car speed task

b) Task: Compute Throttle

This task functions as the controller and actuator. The PID algorithm was implemented in this function, and “anti-windup” was included⁵.

Once the necessary control value has been calculated, this value is then scaled to an 8-bit throttle position (in the range of 0-255). The partial code listing is illustrated in Listing 2.

```
void Compute_Throttle(void)
{
    unsigned char pc_throttle = 0;
    static float throttle = 0;
    float car_speed = 0;
    float set_speed = 0;
    float speed_error = 0;

    car_speed = Scaled_Speed_G;
    set_speed = Ref_Speed_G;

    speed_error = set_speed - car_speed;
    throttle = PID_Control(speed_error, throttle);
    pc_throttle = (unsigned char)(throttle * 255);
    Throttle_Port = pc_throttle;
}
```

Listing 2: An example of the compute throttle task

c) Task: Get Ref Speed

The purpose of this task is to obtain the reference or desired set speed that the driver may want the car to travel at. The reference speed of the car is obtained using a 0-5 volts potentiometer. An on-board analogue to digital converter (ADC) is used to capture the signal. The signal is then scaled to the reference speed within the required range. The code listing for the implementation on a C167 microcontroller is illustrated in Listing 3.

⁵ Anti-windup protection ensures that – when the throttle is at a maximum or minimum value – the error value does not continue to accumulate. For further details, see Åström, 2002.

```

void Act_Get_Ref_Speed(void)
{
    tWord Time_out_loop = 1;
    tWord AD_result = 0;

    ADST = 1;

    while ((ADBSY == 1) && (Time_out_loop != 0))
    {
        Time_out_loop++;
    }

    if (!Time_out_loop)
    {
        AD_result_G = 0;
    }
    else
    {
        AD_result_G = ADDAT;
    }
    Ref_Speed_G = (tByte)((AD_result_G / 1023.0f) * MAX_CAR_SPEED);
}

```

Listing 3: An example of the get ref speed task

d) Task: Display Ref Speed

This task uses the task “PC Link Update” to display the required operating speed of the car.

e) Task: PC Link Update

The purpose of this task is to display information on a terminal emulator (for example HyperTerminal running on a PC) by means of an RS232-based serial connection from the processor node on which this task is running.

8. Case study

To obtain some preliminary results from the CCS testbed, we compared the control performance of one-node and two-node CCS implementations. In each case the CCS nodes were implemented using an Infineon 16-bit microcontroller (Phytec C167CR development board): such devices are widely used in the automotive sector (Siemens, 1996).

a) Implementation of one-node CCS

The description of the one node CCS was given in Section 6 and 7. The tasks were implemented using a time-triggered co-operative scheduler (see Pont, 2001). A picture of the single node setup is shown in Figure 10. The source code for a single node implementation on the C167 is discussed in Appendix B.

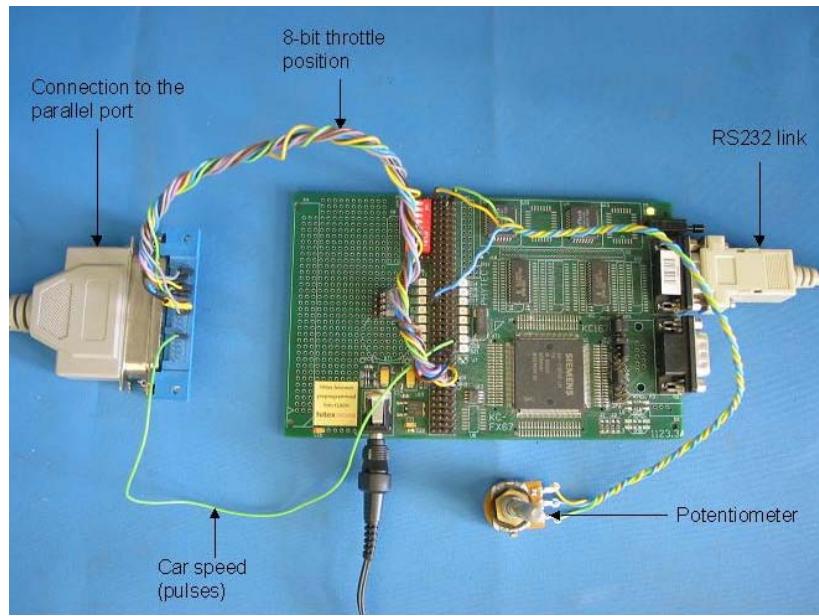


Figure 10: Implementation of a one-node CCS on the C167

b) Implementation of a two-node CCS

In the second design, the CCS was designed to operate as a distributed system using two nodes: a sampler node and a controller / actuator (CA) node (Figure 11). The sampler node was used to calculate the vehicle speed. The calculated car speed was then sent over a network to the CA node. On the CA node, the PID algorithm was used to calculate the required throttle position. The CA node was also responsible for obtaining the required “set speed” value (from the driver). The nodes were linked using a CAN bus running at 333.3 kbytes/s.

In this particular implementation, a “Time-Time-Time” system was employed. As such, the scheduling on both nodes was time-triggered and the network protocol was also time-triggered, using shared-clock scheduling (Pont, 2001). On both nodes, the tasks were scheduled to execute periodically. A “tick” message was sent from the sensor node at the beginning of every sensor node “tick”. This message was used to synchronize the CA node. The sensor status and the car speed data were also included in this message. An acknowledgement message from the CA node was then sent back to the sensor node.

The source codes for the two-node implementation on the C167 boards is discussed further in Appendix C.

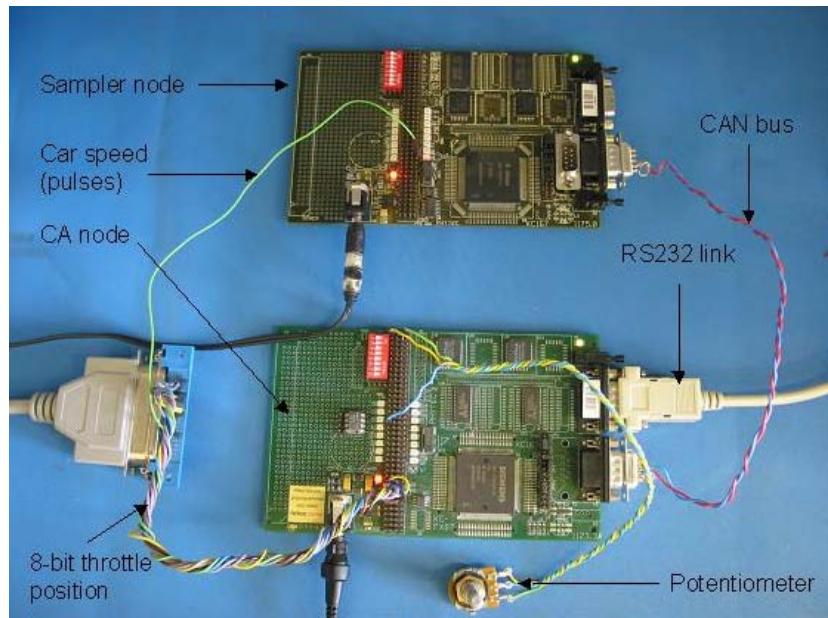


Figure 11: Implementation of a two-node CCS on the C167

c) Results

The results for the two different implementation options were compared at four different set speed values (60mph, 100mph, 170mph and 120 mph). Figure 12 shows the car speed for the two different implementations. Although both the implementation options were very similar, there was some slight differences. The results show that the single node implementation could maintain the car speed more accurately to the desired speed compared to the two-node system.

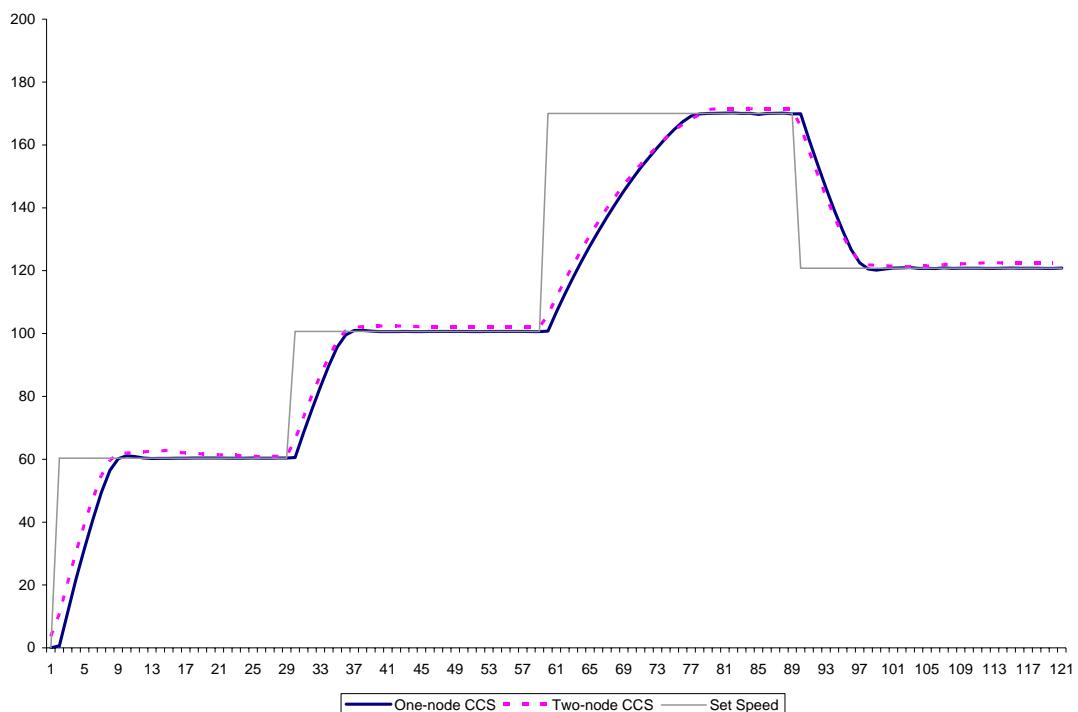


Figure 12: The performance (speed) of the car for two different implementation options

Figure 13 shows the differences in the throttle performance for the different implementations. The results indicate that the responsiveness of the controller to variations in the output signal for the single node system was – again – slightly better than the two-node implementation.

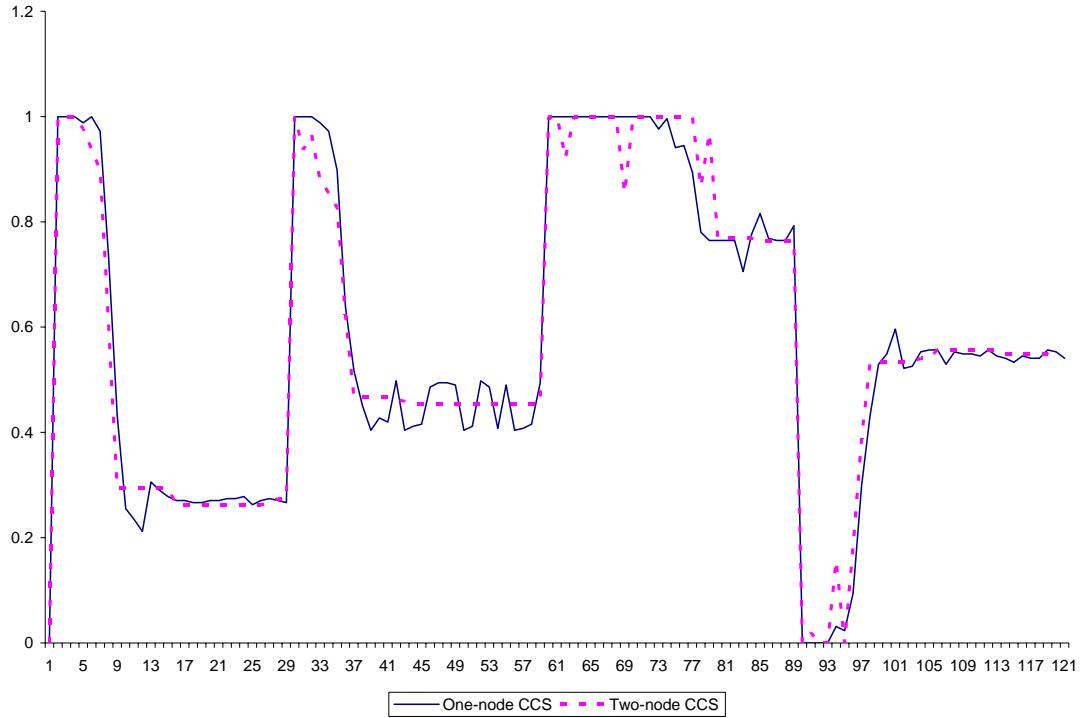


Figure 13: The performance (throttle) of the car for two different implementation options

These differences in the results can be attributed to the fact that the distributed system has an additional delay element involved in the network communication. Such delays can have an impact on the control performance of the system (Törngren, 1998).

9. Conclusions

In this paper, we have described a simple HIL testbed that can be used to quickly evaluate some design and implementation strategies for embedded automotive control systems. Although this system is incomplete, the case study shows that this simple setup can be used to compare different implementation solutions for embedded control systems.

References

- ARM, 2001. ARM7TDMI Technical Reference Manual.
- Aström, 2002. Control System Design - Lecture Notes, Department of Mechanical and Environmental Engineering, University of California, Santa Barbara.
- Bannatyne, R., 2003. Microcontrollers For Automobiles. Micro Control Journal, Transportation Systems Group(Motorola Inc).
- Dorf, D. and Bishop, R., 1998. Modern Control Systems. Addison Wesley.
- Dutton, K., Thompson, S. and Barraclough, B., 1997. The Art of Control Engineering. Addison Wesley.
- Dynasim, 2003. Hardware-in-the-Loop Simulation of Physically Based Automotive Model with Dymola, Dynasim, Lund, Sweden.
- El-khoury, J. and Törngren, M., 2001. Towards A Toolset For Architectural Design Of Distributed Real-Time Control Systems, IEEE Real-Time Symposium. IEEE, London, England.
- Franklin, G.F., Powell, J.D. and Workman, M.L., 1998. Digital Control of Dynamic Systems. Addison-Wesley.
- Hanselmann, H., 1996. Hardware-in-the-Loop Simulation Testing and its Integration into a CACSD Toolset, The IEEE International Symposium on Computer-Aided Control System Design, Michigan, USA.
- Hartwich, F., Muller, B., Fuhrer, T., Hugel, R. and GmbH, R.B., 2000. CAN Networks with Time-Triggered Communication, 7th international CAN Conference.
- Hartwich, F., Muller, B., Fuhrer, T., Hugel, R. and GmbH, R.B., 2002. Timing In The TTCAN Network, Proceedings 8th International CAN Conference.
- Kopetz, H., 1997. Real-Time Systems: Design Principles For Distributed Embedded Applications. Kluwer Academic.
- Kopetz, H., 2001. A Comparison of TTP/C and FlexRay. Research Report 10/2001.
- Lonn, H. and Axelsson, J., 1999. A Comparison Of Fixed-Priority And Static Cyclic Scheduling For Distributed Automotive Control Application, The Eleventh Euromicro Conference on Real-Time Systems, York, England.
- Messmer, H.P., 2002. The Indispensable PC Hardware Book. Addison-Wesley.
- Nahas, M., Short, M.J. and Pont, M.J., 2005. The Impact of Bit Stuffing on the Real-Time Performance of a Distributed Control System, 10th international CAN Conference, Rome, Italy.
- Nissanke, N., 1997. Realtime Systems. Prentice Hall.
- Ogata, K., 2002. Modern Control Engineering. Prentice Hall.
- Pont, M.J., 2001. Patterns For Time Triggered Embedded Systems. Addison Wesley.
- Pont, M.J., Norman, A.J., Mwelwa, C. and Edwards, T., 2003. Prototyping Time-Triggered Embedded Systems Using PC Hardware. In: K. Henney and D. Schutz (Editors), Eighth European Conference on Pattern Languages of Programs (EuroPLoP), Irsee, Germany.
- Rajnak, A. and Ramnerfors, M., 2002. The Volcano Communication Concept, International Congress on Transportation Electronics. Society of Automotive Engineers Inc.
- Shaw, A.C., 2001. Real-Time Systems and Software. John Wiley & Sons Inc.

- Short, M.J. and Pont, M.J., in press. Hardware in the loop simulation of embedded automotive control systems, IEEE International Conference on Intelligent Transportation Systems 2005, Vienna, Austria.
- Short, M.J., Pont, M.J. and Huang, Q., 2004a. Development of a Hardware-in-the-Loop Test Facility for Automotive ACC Implementations. ESL04-03, Embedded Systems Laboratory, University of Leicester.
- Short, M.J., Pont, M.J. and Huang, Q., 2004b. Simulation of Vehicle Longitudinal Dynamics. ESL 04-01, Embedded System Laboratory, University of Leicester.
- Short, M.J., Pont, M.J. and Huang, Q., 2004c. Simulation of Motorway Traffic Flows. ESL04-02, Embedded Systems Laboratory, University of Leicester.
- Siemens, 1996. C167 Derivatives - User's manual, Version 2.0.
- Specks, J.W. and Rajnak, A., 2002. LIN- Protocols, Development Tools, and Software Interfaces for Local Interconnect Networks in Vehicles, 9th International Conference on Electronic Systems for Vehicles.
- Tanenbaum, A.S., 1995. Distributed Operating Systems. Prentice Hall.
- Törngren, M., 1998. Fundamentals of Implementing Real-Time Control Applications in Distributed Computer Systems. Journal of Real-Time Systems, 14: 219-250.

Appendix A. Car environment model

Appendix A illustrates the code implementation of the Car Environment Model. The Car Environment Model was implemented in C code, based on the design model in Section 3. The implementation of some of the crucial tasks (as illustrated in Section 4) is documented in this section. All the relevant source code in Appendix A was compiled using Open Watcom Version1.1.

Main.C

```
/*-----*
 * Main.C
 *
 * Original Author: Michael J. Pont
 * Modified by Devaraj Ayavoo
 *-----*
 *
 * Simple car simulation using DOS scheduler.
 *
 * *** All timing is in TICKS (not milliseconds) ***
 *
 * Some of this code is adapted from examples which appear in the book:
 * PATTERNS FOR TIME-TRIGGERED EMBEDDED SYSTEMS by Michael J. Pont
 * [Pearson Education, 2001; ISBN: 0-201-33138-1].
 *
 * The code may be freely distributed, provided that this header
 * is clearly visible in all copies.
 */
#include "main.h"
#include "sch_dos.h"

#include "car.h"
#include "PRM_Soft.h"
#include "Record_Data.h"

#include "conio.h"
#include "stdio.h"

/* ..... */
/* .. */

int main(void)
{
    tByte Abort = 0;

    // Set up the scheduler
    // Timings are in ticks (~1 ms tick interval)
    // (Max interval / delay is 65535 ticks)
    SCH_Init();

    // Prepare for the car simulation
    Car_Init();

    // Prepare for PRM output
    PRM_Init();

    // Prepare file to record data
    Open_Output_File();

    // Set LPT1 Data port to inputs! (open collector)
    Port_Write_Bit(LPT1_Control,5,1);

    // Add the Car simulation task - execute every 5 ms
    SCH_Add_Task(Compute_Car_Dynamics, 0, 5);
```

```

// Add the screen update function - execute every 100 ms
SCH_Add_Task(Car_Display_Update, 1, 100);

// Add the PRM output - execute every ms
SCH_Add_Task(PRM_Update, 2, 1);

// Record data to file 'result'
SCH_Add_Task(Write_To_File, 3, 1000);

// Start the scheduler
SCH_Start();

while (!Abort)
{
    // Hitting any key will abort the scheduling
    Abort = SCH_Dispatch_Tasks();
}

// Stop the scheduler
SCH_Stop();

// Set LPT1 Data port back to normal
Port_Write_Bit(LPT1_Control,5,0);

return 0;
}

/*
----- END OF FILE -----
*/

```

Car.C

```

/*
----- Car.C -----
Original Author: Michael J. Pont
Modified by Devaraj Ayavoo
-----
Simple Car Model for Cruise control
Contains functions to be used by the scheduler

Some of this code is adapted from examples which appear in the book:
PATTERNS FOR TIME-TRIGGERED EMBEDDED SYSTEMS by Michael J. Pont
[Pearson Education, 2001; ISBN: 0-201-33138-1]. 

The code may be freely distributed, provided that this header
is clearly visible in all copies.

*/
#include <stdio.h>
#include <conio.h>
#include <math.h>
#include "main.h"
#include "car.h"
#include "PRM_Soft.h"
#include "console.h"

// ----- Private Constants -----
#define FRIC 50
#define MASS 1000
#define ENGINE_POWER 5000

```

```

#define PULSE_PER_MILE 8000

// ----- Private Variables -----

static float Old_Speed;
float Throttle_Setting = 0.0f;
float Speed;
float SAMPLE_RATE = 200.0f;

/*-----*/
car_dynamics()
Updates the dynamics of the car

-*-----*/
void Compute_Car_Dynamics(void)
{
    tByte Raw_Throttle;
    float Accel,Dist;

    // First obtain the current throttle setting
    // Data port is forced open collector by init
    // function in main.c

    Raw_Throttle=inp(LPT1_Data);

    // Raw_Throttle = 255;
    // Invert this reading if cruise control interface pulls
    // active pins 'low'
    // Raw_Throttle=255-Raw_Throttle;

    // Convert to float
    Throttle_Setting=((float)Raw_Throttle)/255;

    // The car model

    // Calculate instantaneous acceleration
    Accel=((Throttle_Setting*ENGINE_POWER)-(Old_Speed*FRIC))/MASS;

    // Calculate distance travelled since last sample
    Dist=(float)((Old_Speed/SAMPLE_RATE) +
        (Accel/(SAMPLE_RATE*SAMPLE_RATE))/2);

    // Then determine speed for this sample
    Speed=sqrt((Old_Speed*Old_Speed)+(2*Accel*Dist));

    // Get ready for next sample
    Old_Speed=Speed;
}

/*-----*/
Car_Init()
The car model initialisation function

-*-----*/
void Car_Init(void)
{
    // Clear screen and init variables
    clrscr();
    gotoxy(8,28);
    printf("Simple Car Simulator\n");
    Old_Speed=Speed=Throttle_Setting=0;
}

/*-----*/
Car_Display_Update()
Update screen display with speed and throttle settings

```

```
-----*/  
void Car_Display_Update(void)  
{  
    gotoxy(13,26);  
    printf("Current Speed: %3.2f MPH      \n", Speed*MS_TO_MPH);  
    gotoxy(15,22);  
    printf("Current Throttle Setting: %3.1f %%      ",  
          Throttle_Setting*100);  
}  
  
/*-----  
 - - - END OF FILE  
-----*/
```

Appendix B. A one-node CCS implementation (C167)

Appendix B illustrates the code implementation of the CCS on a 16-bit C167 Infineon microcontroller. The CCS was implemented in C code, based on the CCS design in Section 5 and the implementation technique discussed in Section 6. The implementation of some of the key tasks (as illustrated in Section 7) is also documented in this section. All the relevant source code in Appendix B was compiled using Keil µVision2.

Main.C

```
-----*
Main.C

Original Author: Michael J. Pont
Modified by Devaraj Ayavoo
-----
CCS program for C167 scheduler.

5MHz xtal, using PLL: 20 MHz CPU frequency. 1 ms ticks.

Some of this code is adapted from examples which appear in the book:
PATTERNS FOR TIME-TRIGGERED EMBEDDED SYSTEMS by Michael J. Pont
[Pearson Education, 2001; ISBN: 0-201-33138-1]. 

The code may be freely distributed, provided that this header
is clearly visible in all copies.

-----*/
#include "Main.h"
#include "Sch_167.h"
#include "LED_167.h"
#include "Counter.h"
#include "Act_Compute_Throttle.h"
#include "ADC_167.h"
#include "PC_O.h"
#include "ASC0.h"
#include "Display_RS232.h"

/* ..... */ /* */

void main(void)
{
    // Set up the scheduler
    SCH_Init();

    // Prepare for the 'Flash_LED' task
    LED_Flash_Init();

    // Initialize Counter 7 to count pulses
    Counter_7_Init();

    // Set up the ADC for the C167
    AD_Init();

    // Initial the port to write throttle value
    Compute_Throttle_Init();

    // Initialize the serial communication
    ASC0_vInit();
```

```

// Prepare the elapsed time library
Display_RS232_Init();

// - timings are in ticks (1 ms tick interval)
// (Max interval / delay is 65535 ticks)

// Periodic update of car speed
SCH_Add_Task(Sens_Compute_Speed, 1, 50);

// Periodic PID calculation
SCH_Add_Task(Compute_Throttle, 4, 50);

// Obtain the new set speed periodically
SCH_Add_Task(Act_Get_Ref_Speed, 8, 1000);

// Add the 'PC_LINK_O_Update' task
SCH_Add_Task(PC_LINK_O_Update, 5, 10);

// Update the set speed display once per second
SCH_Add_Task(Display_RS232_Update, 6, 1000);

// Add the 'Flash LED' task (on for 1000 ms, off for 1000 ms)
SCH_Add_Task(LED_Flash_Update, 0, 1000);

// Start the scheduler
SCH_Start();

while(1)
{
    SCH_Dispatch_Tasks();
}
}

/*
----- END OF FILE -----
*/

```

Counter.C

```

/*
----- Counter.C -----
Original Author: Michael J. Pont
Modified by Devaraj Ayavoo
-----
Counter Initialization and scaling functions to calculate the
incoming speed pulses

Some of this code is adapted from examples which appear in the book:
PATTERNS FOR TIME-TRIGGERED EMBEDDED SYSTEMS by Michael J. Pont
[Pearson Education, 2001; ISBN: 0-201-33138-1]. 

The code may be freely distributed, provided that this header
is clearly visible in all copies.

*/
#include "Main.h"
#include "Port.h"
#include "Counter.h"

// ----- Public variable -----
#define FILTER_COEFF 0.9
#define SCALING_FAC 4

float Scaled_speed_G;
float Old_speed_G;

```

```

/*-----*/
Counter_7_Init()
- See below.

/*-----*/
void Counter_7_Init(void)
{
    // setup Timer 7 for pulse counting
    // set Timer 7 mode as counter mode with positive edge trigerring
    // External input form P2.15
    T78CON &= 0xFF00;
    T78CON |= 0x0009;

    // clear Timer 7 counter value
    T7      = 0x0000;

    // load CAPCOM2 Timer 7 reload register
    T7REL  = 0x0000;

    // Interrupt on Timer 7 is disabled, just for pulse counting so
    // no need for generating interrupt
    T7IE   = 0;

    Scaled_speed_G = 0.0f;
    Old_speed_G   = 0.0f;
}

/*-----*/
Get_Raw_Speed()
Reads the number of pulses that have been sent from the car

/*-----*/
tWord Get_Raw_Speed(void)
{
    tWord raw_speed = 0;

    // stop the Timer 7 counting
    T7R = 0;

    // Clear T7 interrupt request flag and retrieve the counter value
    T7IR = 0;

    raw_speed = T7;

    // clear the counter
    T7 = 0X0000;

    // start Timer 7 to count again
    T7R = 1;

    return raw_speed;
}

/*-----*/
Sens_Compute_Speed()
Gets the raw speed and scales the value to obtain the actual
speed of the car.

/*-----*/
void Sens_Compute_Speed(void)
{
    tWord raw_speed;

    raw_speed = Get_Raw_Speed();
}

```

```

Scaled_speed_G = ((float)(FILTER_COEFF * Old_speed_G) +
(float)((1 - FILTER_COEFF) * (raw_speed * SCALING_FAC)));
Old_speed_G = Scaled_speed_G;
}

/*-----
--- END OF FILE
-----*/

```

Act_Compute_Throttle.C

```
/*-----*/
```

Act_Compute_Throttle.C

Original Author: Michael J. Pont
Modified by Devaraj Ayavoo

Calculates the necessary throttle position using a PID algorithm

Some of this code is adapted from examples which appear in the book:

PATTERNS FOR TIME-TRIGGERED EMBEDDED SYSTEMS by Michael J. Pont
[Pearson Education, 2001; ISBN: 0-201-33138-1].

The code may be freely distributed, provided that this header
is clearly visible in all copies.

```
-----*/
```

```
#include "Main.h"
#include "Port.h"
#include "Act_Compute_Throttle.h"

// ----- SFRs -----
// ----- Private variable definitions -----
extern tByte Ref_Speed_G;
extern float Scaled_speed_G;

#define PID_KP (0.005)           // Proportional gain
#define PID_KI (0.0)             // Integral gain
#define PID_KD (0.01)            // Differential gain
#define PID_WINDUP_PROTECTION (1) // Set to TRUE (1) or FALSE (0)
#define PID_MAX (1)               // Maximum PID controller output
#define PID_MIN (0)               // Minimum PID controller output
#define SAMPLE_RATE (20)          // Define sample rate in Hertz

static float Sum_G;           // Integrator component
static float Old_error_G;     // Previous error value
```

```
-----*/
```

Compute_Throttle_Init()

- See below.

```
-----*/
void Compute_Throttle_Init(void)
{
    // P8.0 ... P8.7 switch on standard TTL input levels
    // P7.0 ... P7.7 switch on standard TTL input levels
    // P3.8 ... P3.15 switch on standard TTL input levels
    // P3.0 ... P3.7 switch on standard TTL input levels
    // P2.8 ... P2.15 switch on standard TTL input levels
    // P2.0 ... P2.7 switch on standard TTL input levels

    P7    = 0xFFFF; // set port data register
    ODP7 = 0x0000; // set port open drain control register
    DP7  = 0xFFFF; // set port direction register
```

```

}

/*-----
 Compute_Throttle()

Gets the speed of car and computed the new throttle position and
writes it to P6

-----*/
void Compute_Throttle(void)
{
    unsigned char pc_throttle = 0; // throttle setting value for PC
                                    // (8-bit unsigned)
    static float   throttle   = 0; // actual throttle in floating
                                    // point format
    float         car_speed  = 0; // actual speed (from the Pulse
                                    // calculation)
    float         set_speed  = 0; // desired speed
    float         speed_error= 0; // speed error between desired
                                    // speed and the actual speed

    car_speed = Scaled_Speed_G;
    set_speed = Ref_Speed_G;

    speed_error = set_speed - car_speed;

    // perform the PID control for getting the appropriate
    // throttle settings
    throttle = PID_Control(speed_error, throttle);

    pc_throttle = (unsigned char)(throttle * 255);

    Throttle_Port = pc_throttle;
}

/*-----
 PID_Control()

Performs the PID algorithm

-----*/
float PID_Control(float Error, float Control_old)
{
    float Control_new;

    // Proportional term
    Control_new = Control_old + (PID_KP * Error);

    // if the desired speed has been achieved then should clear the
    // accumulated error
    if(Error==0)
    {
        Sum_G = 0;
    }
    else
    {
        // Integral term
        Sum_G += Error;
    }

    Control_new += PID_KI * Sum_G;

    // Differential term
    Control_new += (PID_KD * SAMPLE_RATE * (Error - Old_error_G));

    // Optional windup protection
    if (PID_WINDUP_PROTECTION)
    {
        if ((Control_new > PID_MAX) || (Control_new < PID_MIN))

```

```

        {
            Sum_G -= Error; // Don't increase Sum...
        }

// Control_new cannot exceed PID_MAX or fall below PID_MIN
if (Control_new > PID_MAX)
{
    Control_new = PID_MAX;
}
else
{
    if (Control_new < PID_MIN)
    {
        Control_new = PID_MIN;
    }
}

// Store error value
Old_error_G = Error;

return Control_new;
}

/*-----
----- END OF FILE -----
-----*/

```

ADC_167.C

```

/*-----
----- ADC_167.C
----- Original Author: Michael J. Pont
----- Modified by Devaraj Ayavoo
----- Simple, single-channel, 10-bit A-D (input) library for C167
----- Some of this code is adapted from examples which appear in the book:
----- PATTERNS FOR TIME-TRIGGERED EMBEDDED SYSTEMS by Michael J. Pont
----- [Pearson Education, 2001; ISBN: 0-201-33138-1].
----- The code may be freely distributed, provided that this header
----- is clearly visible in all copies.
-----*/
#include "Main.H"
#include "ADC_167.h"

// ----- Public variable definitions -----
#define MAX_CAR_SPEED 90 // Maximum car speed is set to 90m/s
                        // or 200 mph

// Stores the latest ADC reading
tWord AD_result_G;
tByte Ref_Speed_G;

long int Counter_G = 0;

/*-----
AD_Init()
Set up the A-D converter.
-----*/
void AD_Init(void)

```

```

{
// fixed channel single conversion
// converts channel 0
// ADC start bit is set
// conversion time = TCL * 24
// sample time = conversion time * 1
ADCON = 0x0080;
ADCIR = 0;
ADCIE = 0;
ADEIR = 0;
ADEIE = 0;
}

/*-----*/
AD_Get_Sample()
Get a single data sample from the (10-bit) ADC.

-*-----*/
void Act_Get_Ref_Speed(void)
{
tWord Time_out_loop = 1;
tWord AD_result = 0;

Counter_G++;

// Start the conversion
ADST = 1;

// Take sample from A-D (with simple loop time-out)
while ((ADBSY == 1) && (Time_out_loop != 0))
{
    Time_out_loop++;
}

if (!Time_out_loop)
{
    AD_result_G = 0;
}
else
{
    // 10-bit A-D result is now available
    AD_result_G = ADDAT;
}

// Scale the set speed to the maximum car speed
Ref_Speed_G = (tByte)((AD_result_G / 1023.0f) * MAX_CAR_SPEED);
}

/*-----*/
---- END OF FILE -----
-*-----*/

```

Appendix C. A two-node CCS implementation (C167)

Appendix C illustrates the code implementation of the CCS on the 16-bit C167 Infineon microcontroller. The CCS was implemented using the Shared-Clock CAN architecture with a Master and Slave node. Some of the files are illustrated in here. All the relevant source code in Appendix C was compiled using Keil µVision2.

Main.C (Master node)

```
/*-----*
Main.C

Original Author: Michael J. Pont
Modified by Devaraj Ayavoo
-----*

PROGRAM FOR A DISTRIBUTED 2-NODE CRUISE-CONTROL TTT SYSTEM

MASTER NODE (SPEED SENSOR)

***** 1ms tick *****

Some of this code is adapted from examples which appear in the book:

PATTERNS FOR TIME-TRIGGERED EMBEDDED SYSTEMS by Michael J. Pont
[Pearson Education, 2001; ISBN: 0-201-33138-1]. 

The code may be freely distributed, provided that this header
is clearly visible in all copies.

-----*/
#include "Main.h"

#include "LED_167.h"
#include "SCC_M167.H"
#include "Counter.h"

/* ..... */ /* */

void main(void)
{
    // Set up the scheduler
    SCC_A_MASTER_Init_T2_CAN();

    // Set up the flash LED task
    LED_Flash_Init();

    // Initialize Counter 7 to count pulses
    Counter_7_Init();

    // -----
    // TIMING IS IN TICKS (1 ms tick interval)
    //

    // Periodic update of car speed
    SCH_Add_Task(Sens_Compute_Speed, 0, 50);

    // Periodic flashing of LED
    SCH_Add_Task(LED_Flash_Update, 1, 1000);

    // Start the scheduler
    SCC_A_MASTER_Start();
}
```

```

while(1)
{
    SCH_Dispatch_Tasks();
}

/*-----
----- END OF FILE -----
-----*/

```

Main.C (Slave node)

```

/*-----*
Main.c

Original Author: Michael J. Pont
Modified by Devaraj Ayavoo
-----*

PROGRAM FOR A DISTRIBUTED 2-NODE CRUISE-CONTROL TTT SYSTEM

SLAVE NODE (CONTROLLER/ACTUATOR NODE)

***** 1ms tick *****

Some of this code is adapted from examples which appear in the book:

PATTERNS FOR TIME-TRIGGERED EMBEDDED SYSTEMS by Michael J. Pont
[Pearson Education, 2001; ISBN: 0-201-33138-1]. 

The code may be freely distributed, provided that this header
is clearly visible in all copies.

-----*/

```

```

#include "Main.h"

#include "LED_167.h"
#include "SCC_S167.H"
#include "PC_O.h"
#include "ASC0.h"
#include "Display_RS232.h"
#include "Act_Compute_Throttle.h"
#include "ADC_167.h"

/* ..... */
/* .....
```

```

void main(void)
{
    // Set up the scheduler
    SCC_A_SLAVE_Init_CAN();

    // Set up the flash LED task (demo purposes)
    LED_Flash_Init();

    // set up the ADC for the C167
    AD_Init();

    // Initial the port to write throttle value
    Compute_Throttle_Init();

    // Initialize the serial communication
    ASC0_vInit();

    // Prepare the elapsed time library
    Display_RS232_Init();

    // -----
    // TIMING IS IN TICKS (1 ms tick interval)
    // -----

```

```

// Periodic PID calculation
SCH_Add_Task(Compute_Throttle, 1, 50);

// Obtain the new set speed periodically
SCH_Add_Task(Act_Get_Ref_Speed, 0, 1000);

// Add the 'PC_LINK_O_Update' task
SCH_Add_Task(PC_LINK_O_Update, 4, 10);

// Update the display set speed once per second
SCH_Add_Task(Display_RS232_Update, 5, 1000);

// Periodic flashing of LED
SCH_Add_Task(LED_Flash_Update, 2, 100);

// Start the scheduler
SCC_A_SLAVE_Start();

while(1)
{
    SCH_Dispatch_Tasks();
}

/*-----
----- END OF FILE -----
-----*/

```

The Use of ASIPs and Customised Co-processors in an Embedded Real-Time System

Jack Whitham and Neil Audsley

Department of Computer Science

University of York, York, YO10 5DD, UK

{jack|neil}@cs.york.ac.uk

July 13, 2005

Abstract

Embedded systems are increasingly being designed in a system-on-chip (SoC) form to save costs. Often, a field programmable gate array (FPGA) device is used to contain the processor, memory and devices used by the system. In this environment, it is possible to make use of an application-specific instruction processor (ASIP) in place of a conventional processor, and it is also possible to add customised co-processors to the device. Both of these technologies can be optimised to accelerate a particular application, potentially reducing hardware costs and power consumption.

However, embedded systems are generally also real-time systems, and the current approaches for determining where optimisations should be applied fail to take this into account, neglecting important real-time properties such as deadlines and response times. We demonstrate that designers may be led to poor optimisation decisions by such approaches, then show how timing properties may be combined with profile data to allow informed optimisation choices to be made.

1 Introduction

Embedded systems designers frequently wish to reduce per-unit cost and power consumption, while at the same time avoiding increasing design costs. The use of Field Programmable Gate Array (FPGA) and Application Specific Integrated Circuit (ASIC) technologies allow per-unit cost, power consumption and overall system size to be reduced by fitting most system components onto a single chip. This is known as a system-on-chip (SoC) architecture.

FPGAs and ASICs allow hardware to be customised to a particular application, which can allow substantial savings to be made. The general approach involves moving frequently executed sections of code into hardware, which can increase the speed of the entire system. A simpler processor or slower system clock frequency can be used, perhaps saving cost, power, physical space, or some combination of the three.

Application Specific Instruction Processors (ASIPs) allow system performance to be optimised at the architectural level by inclusion of custom hardware within the processor to support new instructions (e.g. [20, 4, 3]). Thus, they require modification of an existing processor core. ASIP manufacturers sell this existing core with the tools needed to customise it.

Custom co-processor devices allow customised hardware to be included at the bus level. They can be added to any system with any processor core. They also execute independently of the main processor, so some parallelism is possible.

In both cases, the key design time issue is that of function selection, i.e. which functions of the application should be implemented with custom instructions or using a co-processor. Conventionally, this is achieved by revealing frequently executed areas of code using a profiler [20, 24, 2]. These areas are then translated into either a hardware description language (HDL) for inclusion in a co-processor, or, in the case of an ASIP, into a combination of assembly code and custom instruction descriptions in a HDL. This process can be carried out automatically [2], but it is best to do the work by hand,

because a designer will have a high level understanding of the function of a particular piece of code which cannot be derived from source code, allowing the hardware to be used more efficiently [12].

In this paper, we assume that the platform is such that either an ASIP approach or co-processor approach could be taken, and that a hybrid approach combining elements from both is possible. In practice, the designer may be constrained to one particular choice by hardware or cost restrictions. In this case, the work is still applicable, but some of the choices are restricted.

In a hard real-time system, many distinct processes with different timing characteristics must share processing resources. In this environment, current approaches for function selection are not effective. Profiling individual processes does not reveal where the best system-wide improvement can be made, especially where processes interact with each other. This paper focuses on the need for improvements to existing approaches for function selection, and proposes methods that may be used to improve the existing approaches. These methods are then applied within a case study.

The paper is structured as follows. In section 2 background and previous work is discussed. Discussion of instruction selection issues for real-time systems is given in section 3, and section 4 details the solution that we propose to avoid the problems inherent in current approaches. Section 5 presents the results of a practical case study. Finally, conclusions are given in section 6.

2 Background

2.1 ASIPs

ASIPs are conventional processors that allow additional custom instructions to be added to the standard instruction set architecture (Figure 1(a)). Custom instructions are defined at design time: the hardware to execute them essentially forms a fixed part of the ASIC (or FPGA softcore). Usually, the instructions are incorporated within the RTL for the entire processor [4].

The number of custom instructions allowed is limited primarily by the target hardware, not the ASIP or instruction set. The overhead of incorporating custom instructions, in terms of the additional hardware required, is significant. In [23], “a few additional instructions” need “18,000 additional gates” in one sample application, with “image filtering and color-space conversion” custom instructions needing an “additional 64,100 gates” for another application. The basic processor requires only 25,000 gates [21].

Usually, the types of custom instructions permitted can be quite complex, e.g. single-instruction multiple-data (SIMD) vector processing and multiply-accumulate (MAC) instructions [23]. Often, custom instructions are kept to a single cycle in length, although they may have state, enabling more complex functionality to be implemented [23].

2.2 Customised Co-processors

In some applications, it may be inappropriate or impossible to include custom hardware within the processor data path. For example, the processor may not be user-modifiable, being a hard core within an FPGA, a soft core that is sold without RTL source code, or a prefabricated chip. It is also possible that the designer wishes to exploit the isolation between a co-processor and the main processor and have tasks running in parallel on both of them, or that the RTL for a particular function may require a level of complexity that is not available within the data path, such as a need for extremely wide registers.

In these cases, a customised co-processor may be placed on an external bus (Figure 1(b)). The co-processor is often a memory mapped device, with the ability to generate an interrupt when processing is complete. Co-processors must be self-contained devices, including both the hardware required to carry out the desired function, and control hardware to manage bus accesses, which does mean that they take up more hardware space than the equivalent custom instruction based implementation.

We consider only the case of customised co-processors which are used to speed up processing functions. In practice, it is common for device driving hardware to do some processing itself in order to assist the main processor (such as CRC/parity checks in communications hardware), but as this function is not easily separable from the hardware itself, device driving processors are not considered to be co-processors for the purposes of this paper.

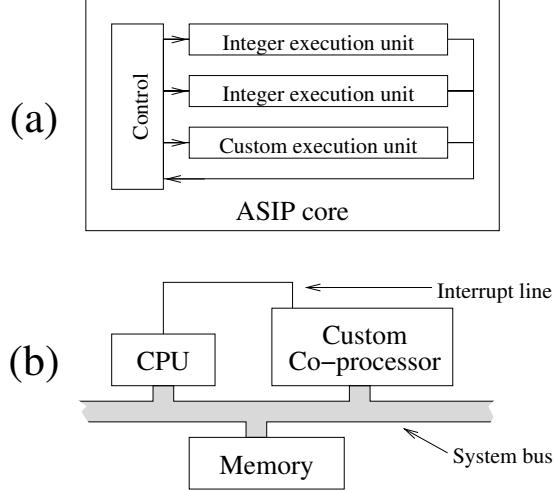


Figure 1: (a): an ASIP core provides customisable execution units on the processor data path. (b): a customised co-processor is generally an additional memory mapped device on the system bus, connected to an interrupt line which is used to signal task completion.

Cascade [2] is a commercial co-processor generation tool providing similar features to ASIP design tools [20]. Unlike custom instructions, co-processors can be arbitrarily complex, and may execute in parallel with the main processor. The number of co-processors that can be introduced is effectively limited only by the size of the hardware platform. However, the communication overhead with a co-processor is substantial. Co-processors have no access to processor registers, and all data must be transferred to and from them via the system bus. For these reasons, co-processors are more suited to complex tasks that take a substantial length of time to process.

2.3 Common Features

The use of either technology may lead to a need for a large FPGA or ASIC, with associated higher costs and power consumption. However, these problems can be reduced by effective selection of code for optimisation. Conventionally, an iterative process is used [20, 24, 2, 11], involving code profiling to find the best places for optimisation, and then implementation of that optimisation using custom instructions or a co-processor. This process is repeated until the code executes sufficiently quickly.

The use of profiling for optimisation selection is similar to the simulation approach used in classic co-design methodologies [13, 17], where profiling is used to decide which application functions are to be implemented in a co-processor. In that case, the implementation is automatic, essentially a direct translation of the software source code to hardware. This highlights an important distinction between co-design and the more recent approaches, in that ASIP approaches rely on the designer to implement the custom instructions, and to make the final decisions about implementation strategies (Cascade makes similar provisions for co-processors). Better optimisations can be made by applying a high-level understanding of the application's requirements, which avoids the sub-optimality of automatic software to hardware translation [12].

However, the profiling approach is limited, as it does not address real-time performance issues arising from multiple interacting processes with complex timing requirements. While [3] does propose tools to profile process performance characteristics within a real-time system, there is no attempt to relate the profiling information to any form of timing analysis in order to guide the selection of custom instructions or co-processor functions.

As the number of custom instructions and co-processors will always be limited by the amount of hardware available, correct selections must be an essential part of optimising system design. The correct choices can minimise hardware cost and power consumption, and may allow the system clock frequency to be reduced. However, making these choices remains an open issue in the context of real-time systems.

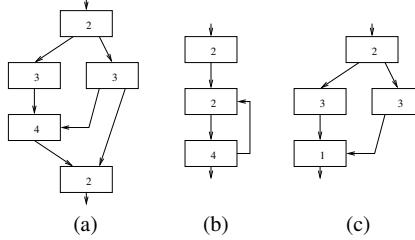


Figure 2: WCET Calculation and Refinement.

3 ASIPs, Co-processors and Real-Time Systems

The correctness of a real-time system is dependent on *when* an application function occurs, as well as the correct operation of that function [7]. Too early and too late are equally bad. Typically, a real-time system consists of a number of application processes, each with specific timing constraints. Hard real-time systems (such as safety-critical systems) contain processes with deadlines that cannot be missed without possibility of catastrophe (e.g. aircraft engine failure).

Hard real-time systems have processes that are either periodic, i.e. invoked at regular (fixed) intervals; or sporadic, i.e. are invoked at random times, but with an enforced minimum time between successive invocations. Note that the effects of sporadic processes are bounded, but the precise times at which they are invoked are not known *a priori*. User input and network traffic are both examples of sporadic processes.

Processes may share resources, such as a software buffer (i.e. memory block) in an exclusive manner, such that blocking may occur when a process wishes to access a shared resource. Also, processes may be affected by interrupts, and operating system overheads (e.g. context switch times).

For hard real-time systems, temporal predictability must be proved prior to system execution - i.e. timing requirements must be shown to be met offline. Testing cannot show the absence of timing faults, hence analytical approaches are used [7]. Timing analysis models the worst-case timing behaviour of the system, given the scheduling (and resource management) policies incorporated in the operating system. For each process, its worst-case response time (i.e. elapsed time between invocation and completion) must be no greater than its deadline. The response time is dependent upon its own computation time, the execution times of other processes, and any time that the process must wait (i.e. whilst blocked) for access to shared resources (e.g. shared memory used in inter-process communication and synchronisation, physical devices). Many such analysis methods have been proposed, for different scheduling policies, e.g. fixed-priority (FP), earliest deadline first (EDF) [7].

Within real-time systems, there lie a number of opportunities for the use of custom instructions and co-processors to improve performance, in terms of process response times. The remainder of this section proposes different strategies that could be used.

3.1 Reducing WCET

The response time of an individual process is dependent upon its own computation time. For hard real-time systems, the worst-case execution time (WCET) is used. This represents the worst-case timing path through executable code of a process [7]. This is calculated by breaking the process code into a graph of basic blocks (single entry/single exit blocks of non-branching code), as given in Figure 2(a). A basic block will always execute in a constant time (assuming conservative pipeline and cache behaviours). The WCET for a process can be computed by finding the path through the graph that maximises the total process time, in this case 11. Note that to calculate WCET requires bounded code, i.e. bounded loops (Figure 2(b) shows a process with undefined WCET, due to its unbounded loop).

ASIP-style custom instructions and co-processors can often replace one or more basic blocks. Conditional statements and loops can be parallelised so that they always execute in a fixed time. Figure 2(a) might be simplified to Figure 2(c) by the addition of a custom instruction which replaces some of the blocks, resulting in a revised WCET of 6. In all cases, if the WCET is known before the

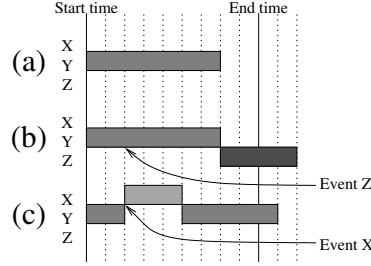


Figure 3: Response Time Example.

use of custom instructions, it will be known afterwards.

3.2 Reducing Main Processor Utilisation

Co-processors can also be used to replace basic blocks, but a co-processor is capable of much more than a single custom instruction. There is potential for parallelism between a co-processor and the main processor: both can execute at once, whereas custom instructions operate just like regular machine instructions - if one takes many clock cycles to execute, the processor pipeline will become stalled and the entire processor will wait for it to complete. Co-processors can take as long as they need to complete without stalling other processes, if the system is designed to take advantage of parallelism. Additionally, it is possible for a co-processor to speed up the system as a whole even if it is no faster than the equivalent software implementation.

The limited parallel model proposed in [5] may be used to model the parallel operation of co-processors. [5] distinguishes between co-processors that are invoked in a master/slave scenario, in which the main processor (master) waits for the co-processor (slave) to complete, and those that are used in a parallel scenario, and describes how co-processors used in the latter scenario can be incorporated into a timing analysis.

However, [5] does not consider how the functions of co-processors should be selected. Nor is context switch and communication time considered. However, all of these factors are of high importance in a practical application. Using a co-processor takes time for communication of data across the memory bus, and will also result in a context switch if parallelism is being utilised, as the main processor will begin executing another task after the co-processor has begun execution. Context switch and communication time are significant unless the co-processor execution time is far greater than both of them.

3.3 Improving Response Times using Custom Instructions

Consider processes X, Y and Z with WCETs of 3, 7 and 4 units, and deadlines of 4, 9 and 25 respectively. Both X and Z are sporadic (with minimum inter-arrival times 6 and 40 respectively), Y is periodic (with period 20). They are assigned priorities such that $X > Y > Z$.

For a fixed priority scheduling policy, the worst-case response times of X, Y and Z are 3, 15 and 22 respectively. Thus, X and Z meet their deadlines, whilst Y does not ($15 > 9$). At run-time some invocations of Y will meet their deadlines if X is not invoked at its maximum frequency (Figures 3(a) and (b)), but in the worst-case it will miss its deadline if X is invoked (Figure 3(c)).

Conventional profiling may suggest targeting process Y for custom instructions as it has the highest WCET. This is not supported by timing analysis, as even reducing it to 4 units (from 7) leaves the modified response time (10) still exceeding the deadline (9). This is due to the worst-case involving two invocations of X (see Figure 4).

In this case, X is a better target for optimisation, even though it is the shortest process when profiled in isolation. Reducing its WCET to 1 unit reduces the response times of X, Y and Z to 1, 8 and 13 respectively.

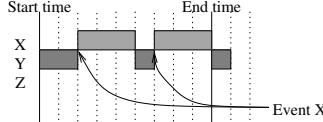


Figure 4: Optimisation of Response Time.

3.4 Reducing Effects of Process Interaction

A further sporadic process (W) is added to those in the example of section 3.3. W has a WCET of 3 units, and both W and Z make use of a resource, R. Each time W runs, it locks R for 3 units, and each time Z runs, it locks R for 1 unit.

Z's WCET (4) is greater than W's (3). Conventionally, Z would be targeted before W. However, W locks a shared resource for a longer period of time than Z. W may block Z by up to 3 units, as Z has to wait for W to release the lock on R. In this situation, it may be beneficial to target W and reduce the amount of time that R is locked for. As before, profiling W and Z in isolation will not reveal this behaviour.

3.5 Summary

Hardware implementations of software functions can speed up the entire system, but the correct function must be chosen for hardware implementation. A method for finding that function is needed. Furthermore, because there are different types of hardware implementation, the method must be also able to suggest which type of implementation would be best: custom instructions, or a co-processor. The method must take the timing properties of processes (e.g. deadlines) into account, along with their run time behaviour (e.g. inter-process interaction).

4 Our Solution to the Real-Time Function Selection Problem

We present a solution to the problem. The solution first describes the selection of a set S of basic blocks from a process P ($S \subset P$), which is the set to be optimised by hardware implementation. Then, a method of selecting the best type of implementation is described.

Function selection can be carried out either by measurement or analysis. An analytical method for finding S would take the known worst-case minimum period between each execution of a process, and use it to calculate the worst-case execution frequency of each basic block in that process. However, this approach omits information about process interaction and is likely to be overly pessimistic, as it must assume the worst case in all cases.

Measurement is a more straightforward alternative, requiring far less analysis and yielding “typical” performance values instead of the worst cases. It is done by profiling a prototype of the system. This will reveal a set of candidates for S, which we will call C. The prototype must be run in conditions that simulate actual operation, as system behaviour may be dependant on input data. The advantage of this measurement approach over analysis is relative simplicity.

This is not conventional profiling of a single process or application. The entire real-time system, including operating system, is profiled, thus implicitly incorporating information about process interaction. We refer to this type of profiling as whole-system profiling.

4.1 Choosing S by Analytical Methods

We now have the set C, containing some sets of basic blocks that are frequently executed. These sets are candidates for S. An analytical method may be used to find the best candidate.

First, calculate the WCET of every process. Next, calculate the direct effect that the choice of each candidate will have on the WCET of each process. Each candidate will affect only one process (the one that it is part of). The effect will be a reduction in the WCET, due to some part of the process being implemented in hardware. Put this data into a table similar to Table 1, which lists resulting WCETs for each candidate choice.

Candidate	Proc. A WCET	B WCET	C WCET
None chosen	15	10	6
Choice 1	15	8	6
Choice 2	10	10	6
Choice 3	9	10	6
Choice 4	15	10	2

Table 1: The effects of each choice of S on the WCET of each process.

Candidate	Proc. A WCRT	B WCRT	C WCRT
None chosen	15	25	56
Choice 1	15	23	29
Choice 2	10	20	26
Choice 3	9	19	25
Choice 4	15	25	27

Table 2: The effects of each choice of S on the WCRT of each process. We have assumed that process A has a period of 30, and B and C have periods of 40.

Using these figures, it is possible to calculate the worst-case response time of each process for each candidate. Table 2 shows an example using the figures from Table 1. The worst-case response times must all be less than or equal to their associated deadlines, so this will allow some candidates to be eliminated. For example, if the deadline for process C was 26, then choices 1 and 4 (and “none”) would be eliminated in this example.

The only remaining candidates are “safe” - no matter which one is chosen, deadlines will be met. The best one should be chosen. We consider the best choice to be the one that minimises overall system utilisation, as this may allow the designer to use a lower clock frequency or a cheaper implementation platform. Table 3 shows the utilisation figure for each (remaining) candidate, calculated by dividing the WCET of each process by the period of that process, and adding up the results.

This indicates that the third candidate is the best choice for S. However, there is one aspect of this choice that has not yet been checked: what is the effect on other processes? This aspect is best examined by simulation or prototyping. Attempting to model process interaction is defeated by the same problems that defeat fully static analysis of large programs - there are too many variables. We recommend either implementing S, or implementing a software model of S, and then testing it out. The next section will describe the use of software models as part of an alternative to an analytical approach.

4.2 Choosing S by Non-analytical Methods

Analytical methods are not always appropriate. In the previous section, we made the assumption that the WCET of every process could be calculated. This is not always possible. We also assumed that the set of processes was fixed, and that the system was a hard real-time system in which every process has a precisely defined period or minimum inter-arrival time. In practice, embedded systems are not always so simple. Many are soft real-time, and meeting deadlines is a quality-of-service issue instead of a safety-critical issue. These systems are often under-engineered from a hard real-time perspective - they work perfectly provided that processes don’t need much more time than their average execution time.

For such systems, we propose the use of a non-analytical method. One could evaluate each possible candidate for S in a prototype or simulation, and choose the one that works best. This is similar to

Candidate	Utilisation
Choice 2	1.48
Choice 3	1.40

Table 3: Utilisation figures for each remaining candidate.

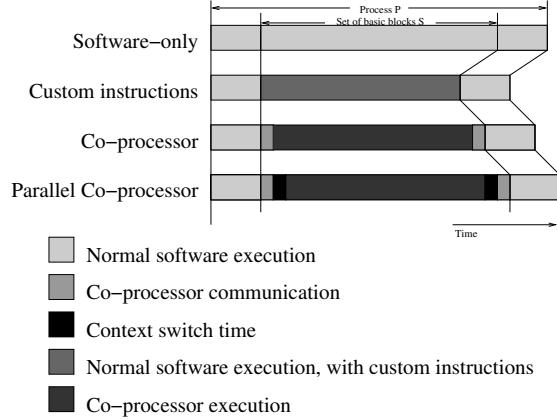


Figure 5: Four different implementations of process P.

the co-design approach to finding the best hardware/software partition, but it is directed only at a few likely candidates rather than all functions.

Evaluating each candidate would normally involve substantial implementation work. Fortunately, this does not have to involve any hardware design, as simulators should allow the use of a software model of some hardware. The software model carries out the same job that the hardware would do. It will generally be based on the original software. It is even possible to use software models that are incomplete implementations, provided that they have the correct timing properties, because this step only examines the effect of a particular choice on the overall timing of the system.

This technique allows each candidate to be modelled in the simulator or in the prototype. The designer can try out all the possibilities with very little implementation effort, and use profiling data from the prototype or simulator to choose the candidate that best fits the requirements. This approach will also show up any problems arising from process interaction. In effect, the period information that was incorporated by analysis in the previous section is incorporated implicitly by this technique.

4.3 Choosing the Implementation Type for S

Having found a suitable set of basic blocks for hardware implementation (S), we consider four possible implementations, which are shown in Figure 5.

Using custom instructions to carry out some of the operations in S reduces the execution time in comparison to software alone, and no extra communication or context switch time is introduced, because the custom instructions are fully inlined in the program and their operations are entirely register-bound.

Using a non-parallelised co-processor introduces some extra communication time, because the data that the co-processor uses must be sent to it, and read back after completion. Co-processors have no access to main processor registers. The software process waits for the co-processor to finish, so no context switching takes place.

Using a parallelised co-processor introduces context switch time, because other processes may execute while the co-processor is operating. Figure 6 shows a process P, which is partly implemented using a co-processor. Processes Q and R, which have lower priorities than P, execute while the co-processor is active. As soon as the co-processor finishes, the operating system returns control to P. As Figure 6 illustrates, two extra context switches are needed to support the parallel operation of the co-processor. Even if there were no lower priority tasks, this would still be the case, as the operating system would switch to the idle process.

Clearly, the different approaches have distinct advantages in different scenarios. Fortunately, a simple timing model can be used to predict which approach is best matched to a particular scenario. Let:

- N be the number of invocations of S during P,
- T_p be the co-processor WCET for S,

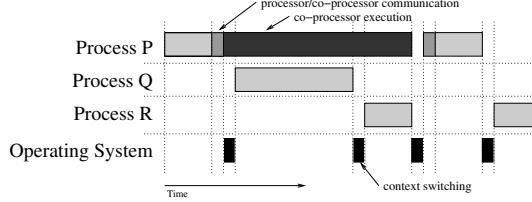


Figure 6: If a parallelised co-processor carries out some of the work of process P, lower priority processes Q and R can execute while the co-processor is active.

Implementation	WCET	Utilisation
Software only	NT_s	NT_s
Custom inst.	NT_i	NT_i
Co-proc.	$N(T_c + T_p)$	$N(T_c + T_p)$
Co-proc. (par.)	$N(T_c + T_w + T_p)$	$N(T_c + 2T_w)$

Table 4: Equations to allow WCET and utilisation time to be calculated for each implementation.

- T_s be the software-only WCET for S,
- T_i be the custom instruction WCET for S,
- T_w be the context switch time, and
- T_c be the overall (bi-directional) co-processor communication time.

For each implementation, Table 4 gives equations to work out the WCET and utilisation time. Utilisation time is equal to WCET in all cases, except for the parallel co-processor case. In that case, the main processor is not in use during co-processor execution, and utilisation time may be considerably less than the process execution time. Designers can use each of these equations to work out the overall utilisation for that implementation and choose the minimum.

We can expect T_i to be approximately equal to T_p . Any optimisation that could be applied to a co-processor to make it more efficient could also be applied to a custom instruction execution unit. We can also expect both T_i and T_p to be less than T_s : there is always a speed gain from moving software components into hardware, provided that the move is carried out correctly. However, no predictions can be made about the relationship between the other figures. The advantage of one approach over another depends critically on the values of T_c and T_w in relation to T_p and T_i .

To summarise this informally, if the context switch time dominates the co-processor execution time, then a custom instruction approach is likely to be better. However, if this is not the case, then significant processor time could be made available to other processes by the use of a parallel co-processor.

5 Case-Study: PDA

This section describes the application of our methods toward effective use of custom instructions and co-processors in a sample real-time platform: a personal digital assistant (PDA) with wireless networking and telephony support.

5.1 Experimental Platform

The SimpleScalar [6] ARM simulator was augmented to form a generic ASIP with capacity for 256 custom instructions, accessed using opcodes that are undefined on a standard ARM processor. Custom instructions are implemented in C, and attached to SimpleScalar using a plug-in system. Additionally, support for simulated memory-mapped co-processors was added to the simulator. Again, these co-processors are implemented in C, and attached as plug-ins.

	WCET/ μ s	Deadline/ μ s	Period/ μ s
V	59,400	250,000	250,000
N	2160	2160	700
I	60,000	100,000	100,000
J	280,000	1,000,000	1,000,000

Table 5: PDA process timing characteristics.

Due to a limitation in Simplescalar, any custom instruction taking more than one clock cycle to complete is assumed to stall the processor until it completes. Simplescalar does not support execution units that require an instruction-dependent number of clock cycles for their work.

The RTEMS [19] real-time operating system was ported to run within this simulator. RTEMS supports multiple processes with fixed priority scheduling. The port of RTEMS has been extended to make process execution statistics available. The amount of time spent in each process (including the idle process) is accounted.

5.2 Description of the PDA

The PDA has next-generation capabilities: wireless networking support and voice-over-IP (VOIP) support which can be used to place and receive telephone calls. The PDA also has typical features, such as a touch-sensitive screen. The effects of the different approaches towards custom instruction selection can be shown by concentrating upon three high-priority processes of particular interest:

1. V: the periodic process for VOIP processing;
2. N: the sporadic process driving the network interface;
3. I: the sporadic process which responds to user input.

A fourth process is also of interest. J is a graphics rendering process which runs sporadically to support a web browser. Its primary function is decoding JPEG images for display on screen. This process has a lower priority than the other three.

Processes V and N are active during a phone call, because voice encoding and decoding are taking place, and data is sent and received over a network. They must run at a high priority, as a call cannot be interrupted by other applications on the PDA. Process I becomes active when the user enters data into the PDA, by using a stylus on the touch screen. This must be responsive, even during a call.

The functionality of V, N, I and J is described further below. Table 5 shows the timing characteristics of software implementations¹. The WCET values were discovered by measurement using the same sample data used throughout the case study. As can be seen by the disparity between the WCET and period of N, this is a soft real-time system.

5.2.1 N: Network Subsystem Process

The PDA wireless networking interface requires a software protocol stack, a low-level hardware driver, and an encryption system (for security). Wireless networks generally use WEP (wired equivalent privacy) technology, as it is specified as part of the 802.11b standard. We modelled this encryption system with 3DES, implemented using libdes [10].

The networking subsystem is modelled as a high priority process (N) that executes whenever a network event is signalled by the release of a semaphore. On execution, it copies a 1kb “packet” from one area of RAM to another, and encrypts that data using DES. This model captures both reception and transmission of data.

Network events are signalled by V to model the flow of data in and out of the system, but they are also signalled randomly to simulate traffic generated by other applications or other computers. The frequency of these random events is one of the parameters of the system that can be changed.

¹700 μ s is the minimum possible inter-arrival time for N. It was computed by assuming that packets of 1kb arrive at the maximum rate over an 11Mb/s wireless network.

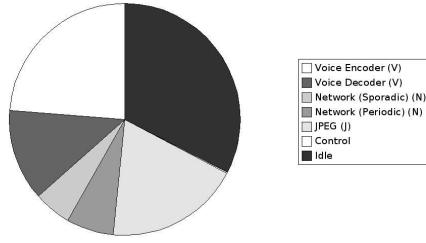


Figure 7: Distribution of CPU time between processes, with no user input or random network events.

5.2.2 V: Voice Subsystem Process

The PDA assumes a voice subsystem consisting of a G.721 codec. G.721 is a predecessor of the GSM standard, used for telephone-quality voice compression. Since the PDA receives and transmits voice data, V is split into a decoding and an encoding process which use the codec.

The MediaBench [16] benchmark suite implementation of G.721 was utilised, as it is ideal for use in an embedded environment. The codec operates on real voice data that is sourced from RAM, at a fixed rate of 4000 bytes per second. The data is handled in blocks of 1000 bytes which arrive every 250ms.

5.2.3 I: Input Subsystem Process

PDAs allow the user to enter data by using a stylus to draw glyphs on the screen. A simple form of handwriting recognition is performed on the glyphs. The input subsystem used here is modelled using the open-source gesture recognition program Wayv [18]. A recording was made using a desktop PC of various gestures being entered into Wayv. This is played back into a Wayv process running within the embedded system to simulate a user entering several symbols per second into the PDA.

5.2.4 J: Graphics Rendering Process

As the PDA has network connectivity, a web browser is provided. The web browser is able to render graphics in JPEG format. Process J handles JPEG decoding for this purpose.

We assume that the PDA decodes one image every second. Images are taken from a set of small pictures, each between 5k and 25k in size, intended to be representative of the types of image found on most web pages. The libjpeg [14] implementation is used.

5.3 Priority Assignment

N is assigned the highest priority. In a real system, the network subsystem would be interrupt driven, and thus would have one of the highest possible priorities.

Process I presents a choice, as priorities could be assigned so that $I < V < N$, or so that $V < I < N$. It is difficult to tell which configuration will yield the best results, so we experiment with both.

Process J has the lowest priority of all. Loading images is not a critical task.

5.4 Using a Software-Only Approach

The simplest way to implement the four subsystems is through software alone, with no special ASIP instructions. In this configuration, the system was allowed to run for ten seconds of simulation time: just over 250 million clock cycles with a clock frequency of 25MHz. Throughout execution, a simulated phone call was in progress, and a new JPEG image was loaded every simulated second.

Execution statistics were gathered after simulation. Figure 7 shows the distribution of CPU time between the processes over the execution period with the input process disabled, assuming no sporadic network events occur. Here, V takes up most of the processor's non-idle time: it is split into an encoding and decoding process.

Figure 8 shows the change in CPU idle time as the level of network traffic increases. We measure the network traffic by counting the total number of randomly generated network packets received

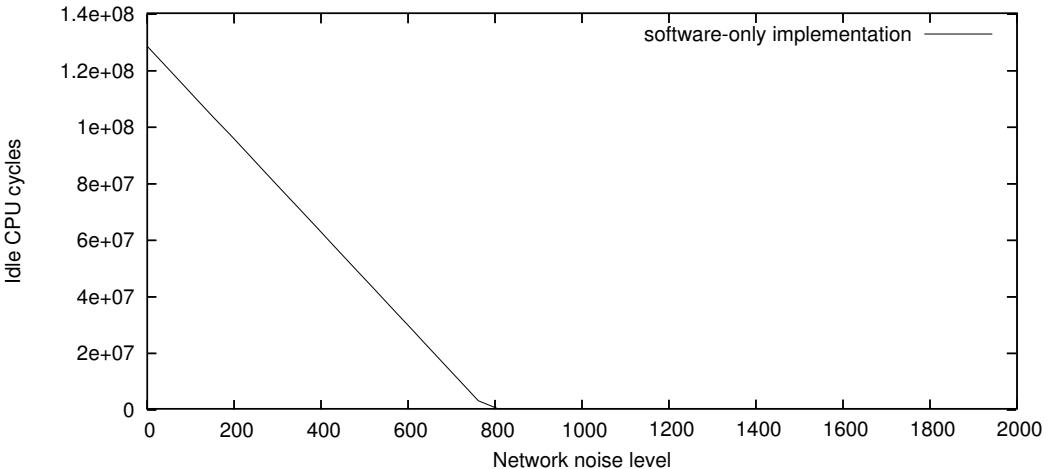


Figure 8: Changes in CPU idle time as network traffic increases, using software-only process implementations.

during the simulation run. In the scenario without any input during the call, the system can support up to 600 packets of network traffic before becoming overloaded.

5.5 Conventional ASIP Improvements

The ASIP method for improving the implementation involves first profiling it to find the most frequently executed parts (the hot spots), and then improving those places by replacing many instructions with a single custom instruction. Each process is isolated, and profiled separately to find its hot spots.

If this is done, then it will immediately become apparent that V takes up much more CPU time than N (Table 5). Using a conventional profiling approach, one might assume that the G.721 code would be the place to make improvements.

Profiling V in isolation reveals that most execution time is spent in the `quan()` function, and most of the calls to that function have a particular parameter set. The entire function can be moved into hardware as a custom instruction: even though it contains a loop and a condition, a high-level understanding of the purpose of the function (evaluation of \log_2) allows it to be replaced with a simple hardware device (a priority encoder). We can expect this to take up just 11 LUTs (look up tables) on a Xilinx Virtex FPGA.

Much execution time is also spent in the `fmult()` function, which can also be replaced by hardware. The function is more complex - the cost of the extra hardware will be 352 LUTs. To reduce this cost, two different implementations were tried: “fmult a” is an all-hardware implementation of the function, and “fmult b” is a hybrid hardware/software implementation.

Figure 9 shows the results of the various enhancements to V. The “fmult a” improvement is the most successful, allowing up to 300 more sporadic network packets than the software-only solution. This improvement requires the definition of two custom instructions for `quan()` and `fmult()`.

5.6 Improving Response Times

The type of improvements discussed in the last section are the type which would be naïvely applied if the developer relied on a conventional profiling approach and improved the high-priority process that used the most CPU time: V. However, when we apply our method (see section 4), we are able to make a better-informed choice.

Our method begins by obtaining a whole-system profile, which we obtain by simulating the entire system using our modified version of SimpleScalar. We obtain a whole-system profile under heavy network load conditions to gain information that is valid in the worst case. The most frequently executed code sections are listed in Table 6.

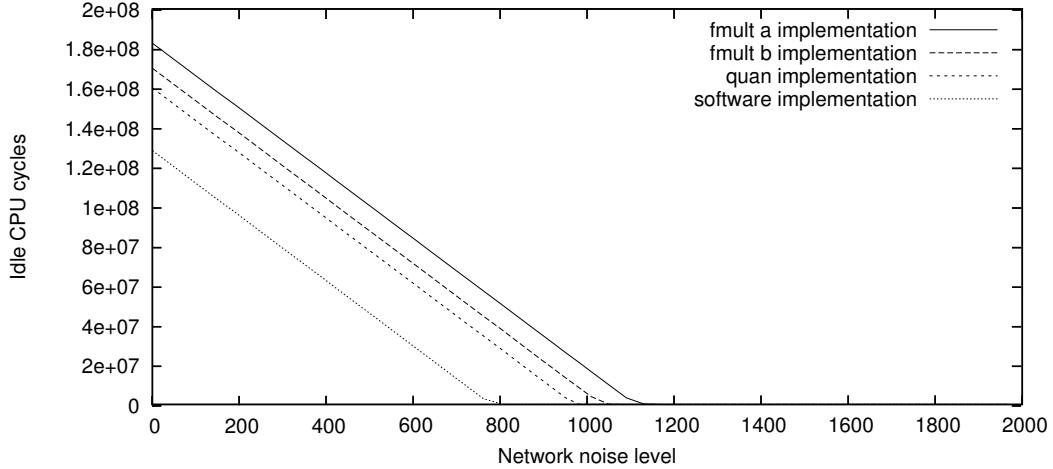


Figure 9: Effects of hardware improvements to V.

Exec. time	Symbol Name	Process
41.2%	des_encrypt	N
15.5%	predictor_zero	V
12.0%	update	V
5.7%	jpeg_idct_ifast	J
5.5%	predictor_pole	V
3.7%	jinit_color_deconverter	J
2.7%	jinit_huff_decoder	J

Table 6: Results of whole-system profiling for the case study.

This profiling turns up one clear candidate for optimisation. Most of the processing time is spent in process N, in the `des_encrypt()` procedure. V is also a candidate. Investigation reveals that an inlined version of `quan()` accounts for most of the time used by `predictor_zero()` and `update()`, as we found in the previous section. So in this case $C = \{\text{des_encrypt}, \text{quan}\}$. Our candidates are the DES encryption code from process N and the `quan()` code from process V.

In contrast to V, each execution of N does not use much CPU time. However, random network activity makes N a very significant user of CPU time, as Figure 10 illustrates. In Figure 10, handling network traffic accounts for over 50% of the processor's time.

In order to apply the analytical method to find the effects of each choice from C on the system, we first compute the effects of a hardware implementation on the WCET of each process, and find that V's WCET is reduced from $59,400\mu\text{s}$ to $23,500\mu\text{s}$ (choosing the best implementation option) and N's WCET is reduced from $2160\mu\text{s}$ to $60\mu\text{s}$. Using this data, we compute the worst-case response time of each process, with each choice from C. Table 7 shows the results.

Table 7 clearly shows that the only candidate that can be chosen for optimisation is `des_encrypt()`. If no optimisations are carried out, then N is capable of starving both V and J of any processor time,

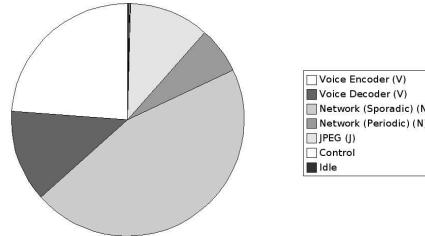


Figure 10: Distribution of CPU time between processes during high levels of network traffic.

Candidate	N WCRT/ μs	V WCRT/ μs	J WCRT/ μs
none	2160	∞	∞
des (N)	60	64,980	436,240
quan (V)	2160	∞	∞

Table 7: Worst-case response times of all processes for each choice from C.

Implementation	WCET/ μs	Utilisation/ μs
Software	2,160	2,160
ASIP-style	60	60
Co-proc.	55	55
Co-proc. (par.)	60	58

Table 8: Values for the WCET and utilisation of process N, with four different implementation options.

as evidenced by their infinite WCRT values. This is because the minimum inter-arrival time of N is less than the WCET of a software implementation of N. In the worst case, network traffic can prevent any other processes from running, even if `quan()` is optimised. Therefore, `des_encrypt()` must be optimised first. Having done this, response time analysis predicts that N, V and J will meet their deadlines (700, 250,000 and 1,000,000 μs respectively).

5.7 Possible Implementations of DES

Hardware optimisations for software are considered to take one of two forms in this work. As described in section 4.3, either a co-processor or ASIP-style custom instructions could be used. The method from section 4.3 will now be applied to find the best method to be used in this case.

The software-only WCET for N (T_s) is 2160 μs . Using an ASIP-style approach, two types of implementation are possible: a single instruction that carries out the entire DES operation, or an implementation based on seven simple custom instructions, as described in the example in [24]. The first approach yields a WCET of 60 μs , and the second yields a WCET of 1570 μs . We choose the first approach here, so $T_i = 60\mu s$. However, the second approach may be valuable in cases where the amount of space available for additional logic is severely restricted.

150 DES operations are required to handle each entire network packet. Conventionally, DES processors encrypt one 64-bit word every 16 clock cycles, but the operation may be pipelined, so only 165 clock cycles are required to carry out 150 operations. This takes $T_p = 6.6\mu s$. The communication time is the time taken to transfer 150 64-bit words back and forth: $T_c = 48\mu s$. The context switch time $T_w = 5.21\mu s$.

We now have values for all of the timing model variables, except N , the number of invocations of DES during process N, which is 1. Applying the equations from Table 4 gives values for the WCET and utilisation of N, with each optimisation (Table 8).

Table 8 suggests that the best implementation is a non-parallelised co-processor. Parallelised co-processors bring extra context-switch time with them: there is a net increase in both the WCET and utilisation of N when a parallelised implementation is used.

Figure 11 shows the effect of each implementation on performance. This confirms the result obtained by analysis: a non-parallelised co-processor implementation is most effective.

5.8 Reducing Process Interaction Effects

Adding a user input process I (as described in section 5.2.3), affects the overall CPU time available to the system. Figure 12 shows three different configurations for process I (no input process, high priority, and low priority), with two types of DES implementation.

V and N make use of a shared network resource, and process I does not. Because execution of V generates a packet to be processed by the network, executions of V lead to additional executions of high-priority process N. When user input events are generated, process I uses CPU time, and reduces

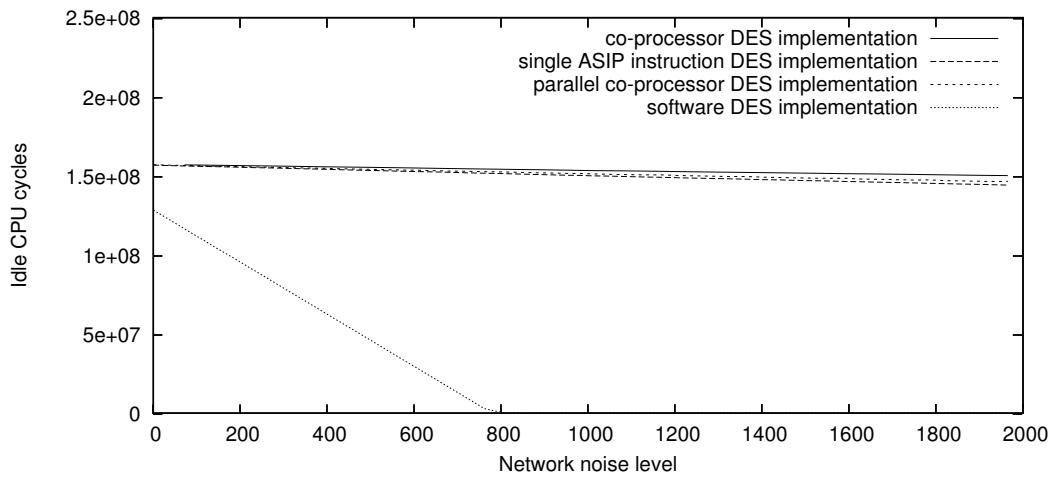


Figure 11: Effects of hardware improvements to DES.

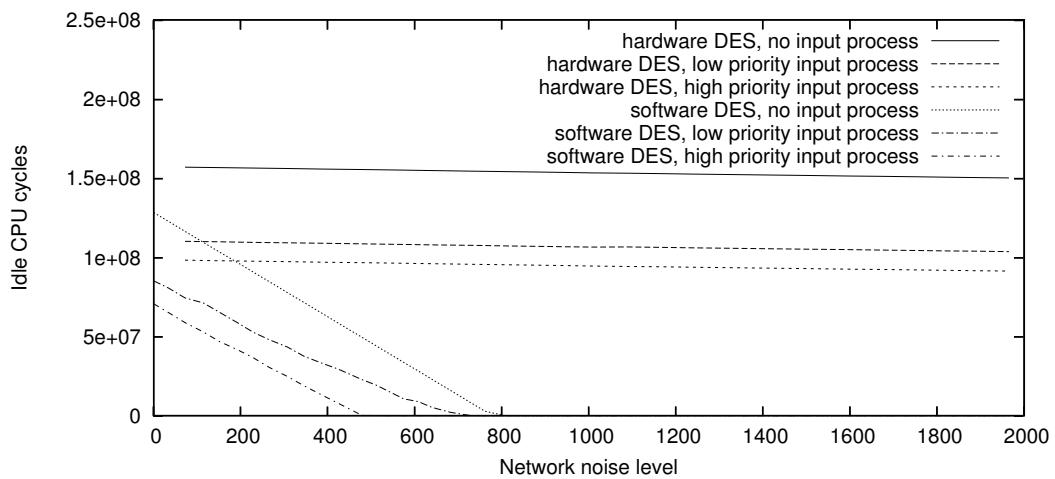


Figure 12: Effects of user input on the system.

Implementation	WCET/ μ s	% Improv. over sw
Software	188,200	0%
Parallel co-proc.	218,500	-16%
Non-parallel co-proc.	177,000	6%

Table 9: Values for the WCET of process J with three different implementation options.

the amount available for V when $I > V$. This reduces the number of executions of N, and causes the divergence between the high priority and low priority lines on Figure 12.

This is an example of how a lower priority process can affect the execution of a high priority process, and thus affect the execution of the entire system, similar to the example in section 3.4. Here, minimising N's WCET will minimise the effects of V's extra executions: when a hardware DES implementation is used, there is no visible difference between high priority and low priority input processes.

5.9 Freeing More Processing Time for Low Priority Processes

Process J has been largely disregarded in this discussion. It has the lowest priority: the assumption being that the user finds good quality of telephone service to be more important than JPEG images loading quickly. However, it is a significant user of processor time, using 19% of available time when the system load is low (Figure 7). One way to speed up J is to attempt to optimise it directly.

The most significant section of J is the IDCT (inverse discrete cosine transform) procedure, the core of the JPEG decoding algorithm [22]. IDCT is used for decoding several common still image and video formats, including MPEG-2 and MPEG-4. As a result, IDCT co-processors are readily available for use on FPGAs and ASICs for accelerating image and video decoding [8]. It is common practice to implement JPEG/MPEG decoders using both a program and an IDCT co-processor. The program handles the aspects of JPEG decoding that require a state machine, access to operating system functions, and access to memory, and the IDCT co-processor handles the decoding function itself. However, it is also common to make use of a complete JPEG decoder core, which is able to handle all decoding functions for data sent to it over the system bus [9]. These cores are much larger than IDCT co-processors. For example, the CAST JPEG core takes up 3923 slices [9] on a Xilinx Virtex FPGA, whereas the CAST IDCT core only takes up 1391 slices [8].

The libjpeg [14] implementation used in J is already very fast. Software modelling shows us that a parallelised co-processor for IDCT brings no benefits, without any requirement to work out execution times, because the processing time required is insignificant compared to the context switch time. However, modelling also shows that a non-parallelised co-processor may provide some improvement over software alone (Table 9).

Taking a whole-system view, much better results can be achieved by optimising process V. Improving V frees up more time for J. If the optimisations for V described in section 5.5 are applied in addition to those for N, then much more idle time is available for J, as can be seen in Figure 13. In fact, the time available to J (idle time, plus time used by J) increases by 29% - far better than the best improvement that could be made by optimising IDCT (6%). Extra hardware is more usefully applied to V, not J.

5.10 Energy Consumption

`sim-panalyzer`, a power modelling extension for SimpleScalar, can be used to calculate the relative energy consumption of systems implemented using the various technologies under different loading conditions. For the purposes of this experiment, we assumed a 25MHz clock frequency, and 250mW power consumption during idle time. We also assumed that the custom instruction hardware used the same amount of power as an additional integer ALU. Figure 14 shows the variation in energy consumption for six implementations.

Figure 14 shares many features with Figure 12. The relationship between idle CPU times and energy usage is roughly linear up until the point where the system becomes overloaded. Reducing WCET and reducing power requirements are related problems - in this example, they are effectively equivalent. Future work could involve applying ASIP optimisations to explicitly maximise idle time,

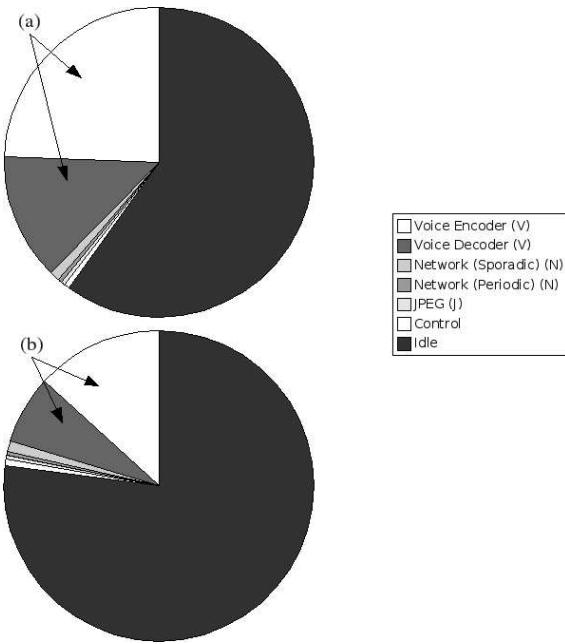


Figure 13: The effect of optimisations to V on the overall system. (a) shows the CPU usage of an unoptimised version of V, and (b) shows the CPU usage of an optimised version. In (b), much more idle time is available for process J.

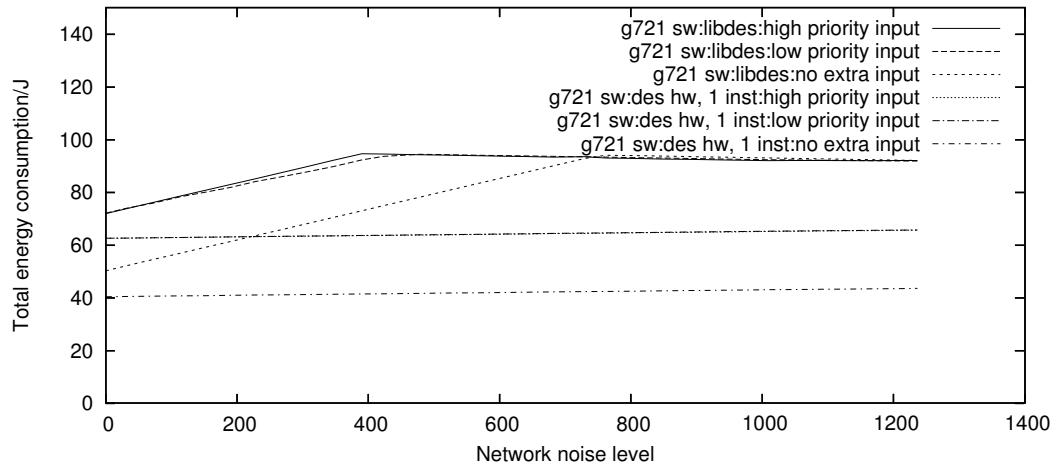


Figure 14: Total system energy consumption due to network traffic, with six different implementations.

which could be combined with the work on maximising procrastination intervals from [15] to reduce energy consumption even further.

5.11 Parallelised Co-processors

When analysing process N, and modelling process J, parallelised co-processors were considered, but eliminated from consideration because they provided no overall improvement. In both cases, this was because the context switch time proved to be significant. From this, we draw the conclusion that in order for a parallelised co-processor to be useful, it must offer a time saving that is substantially greater than the context switch time, and the communication time. There are benefits to using parallelised co-processors, but they are unlikely to be suitable for simple functions that require only a small amount of computation time.

In the case study, we found no examples of parts of processes that could be moved to a co-processor. It seems that the profiling technique is an inadequate way to find parts of processes that can be moved in this way, because profiling tends to find only short sections of code that occupy a great deal of processor time. It does not find long sections that are bottlenecks, such as complete processes, which would potentially work well in a parallel co-processor.

This finding is a general property of profiling, and not specifically a shortcoming in our method. It appears that the allocation of processes to parallel co-processors is a process that can only be guided by profiling data at present. For example, whole-system profiling can reveal which processes occupy significant proportions of processor time. This allows us to see that process V could be moved in its entirety to a co-processor, such as [1]. As V takes up 37% of the processor time with a software implementation, and 20% with an ASIP-style implementation, the benefits of implementing V as a parallelised co-processor could be significant: far more computing time would be available to other processes. The examination of these tradeoffs will be the subject of future work.

5.12 Case Study Findings

Our improvements to the case study have primarily targeted process N. These improvements allow the system to efficiently support far more network traffic than possible with a software-only implementation, by significantly reducing N's WCET. Additionally, we have determined that improvements to process V are a more effective way to speed up process J than direct improvements to J, and showed the effects of process interaction by looking at process I.

6 Conclusion

The conventional methods of applying ASIP and co-processor technology to a multi-tasking real-time system will not work in all cases, because process interaction must be considered. We have described two methods of determining how to take this into account when applying hardware optimisations: analysis, and measurement of software models running under whole-system profiling. Our case study has shown the application of these methods to an example system.

However, profiling the entire system as it executes on the target platform will only produce useful information if all input conditions can be tested. We would not have seen the value of targeting process N in the case study if we had not tried simulating the system under high network load conditions.

Simulation software provides an ideal environment for the types of profiling that are needed for analysis, because it allows both the operating system and applications to be profiled non-intrusively and as a whole. However, executing the whole system in any sort of prototyping environment is sufficient. Although the problem of improving a complex real-time system using ASIP and co-processor technology is not simple, it is not intractable because of the information that can be gained from whole-system profiling, which can then be applied to analysis.

References

- [1] Altera Inc. Multi-standard ADPCM co-processor (accessed 20 may 05). http://www.altera.com/-products/ip/dsp/speech_audio_processing/m-amp-multi-std-adpcm.html.
- [2] Anonymous. Coprocessor synthesis - increasing SoC platform ROI. White paper, CriticalBlue Corporation, 2002.
- [3] ARC International. Integrated profiler (accessed 1 Apr 05). <http://www.arc.com/software/-operatingsystems/prototyping/integratedprofiler.html>.
- [4] ASIP Meister. Home page (accessed 1 Apr 05). <http://www.eda-meister.org/asip-meister/>.
- [5] N. C. Audsley and K. Bletsas. Fixed priority timing analysis of real-time systems with limited parallelism. In *Proceedings of the 16th Euromicro Conference on Real-Time Systems (ECRTS 04)*, Catania, Italy, Jul 2004. IEEE Computer Society.
- [6] D. Burger. SimpleScalar tools (acc. 1 Apr 05). <http://www.cs.wisc.edu/~mscalar/simplescalar.html>.
- [7] A. Burns and A. J. Wellings. *Real-Time Systems and Programming Languages*. Addison Wesley, 2001.
- [8] CAST Inc. IDCT decoder core (accessed 20 may 05). <http://www.cast-inc.com/cores/idct/-index.shtml>.
- [9] CAST Inc. JPEG decoder core (accessed 20 may 05). <http://www.cast-inc.com/cores/jpeg-d/-index.shtml>.
- [10] Eric Young. libdes (accessed 1 Apr 05). <http://www.shmoo.com/crypto/>.
- [11] T. Glöckler and H. Meyr. *Design of Energy-Efficient Application-Specific Instruction Set Processors*. Kluwer Academic Publishers, 2004.
- [12] B. Grattan, G. Stitt, and F. Vahid. Codesign-extended applications. In *Proc. 10th Int. Symp. Hardware/Software Codesign*, pages 1–6, 2002.
- [13] R. K. Gupta and G. D. Micheli. Hardware-software cosynthesis for digital systems. *IEEE Des. Test*, 10(3):29–41, 1993.
- [14] Independent JPEG Group. libjpeg (accessed 19 May 05). <http://www.ijg.org/>.
- [15] R. Jejurikar and R. Gupta. Procrastination scheduling in fixed priority real-time systems. In *Proc. LCTES*, pages 57–66, 2004.
- [16] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *Int. Symp. Microarchitecture*, pages 330–335, 1997.
- [17] G. D. Micheli, W. Wolf, and R. Ernst. *Readings in Hardware/Software Co-Design*. Morgan Kaufmann Publishers Inc., 2001.
- [18] Mike Bennett. wayv (accessed 1 Apr 05). <http://www.stressbunny.com/wayv/>.
- [19] OAR Corporation. RTEMS (accessed 1 Apr 05). <http://www.rtems.com/>.
- [20] Tensilica Corporation. Accelerating existing C code (accessed 1 Apr 05). http://www.tensilica.com/html/accelerating_existing_c_code.html.
- [21] Tensilica Corporation. Xtensa cores excel in EEMBC benchmarks (accessed 1 Apr 05). http://www.tensilica.com/html/technology_eembc.html.
- [22] G. K. Wallace. The JPEG still picture compression standard. *Commun. ACM*, 34(4):30–44, 1991.

- [23] A. Wang, E. Killian, D. Maydan, and C. Rowen. Hardware/software instruction set configurability for system-on-chip processors. In *DAC '01: Proc. 38th conf. on Design automation*, pages 184–188, 2001.
- [24] Z. A. Ye, A. Moshovos, S. Hauck, and P. Banerjee. Chimaera: a high-performance architecture with a tightly-coupled reconfigurable functional unit. In *Proc. 27th Int. Symp. Computer Architecture*, pages 225–235, 2000.

On-line IDDQ testing of security circuits

Alex Bystrov, Julian Murphy
School of Electrical, Electronic and Computer Engineering
Merz Court, University of Newcastle,
Newcastle upon Tyne, UK, NE1 7RU
e-mail: {a.bystrov, j.p.murphy}@ncl.ac.uk
phone: +44 (0)191 222 8184; **fax:** +44 (0)191 222 8180

Abstract

The new at-speed on-line IDDQ testing method is based upon the properties of a special class of security circuits. These circuits implement dual-rail encoding and return-to-spacer protocol, where the spacer is either all-zeroes or all-ones. The alternating spacers of different polarity guarantee that all wires switch once within each clock cycle, thus making energy consumed within a clock cycle independent from data processed. This property earlier used for security purposes is now exploited in order to separate the transient current from the quiescent current, thus making it possible to measure the latter under normal operation. Power signatures are analysed in the frequency domain and the fault signature filtration method is proposed. The proposed method can be used in both production, where it covers all interconnect stuck-at faults in just two clock periods; and on-line testing, where it guarantees the bounded and short period of self-test independent from data. From security point of view, it directly detects a side channel created under a fault injection attack.

Index Terms: On-line testing, IDDQ testing, dual-rail encoding, hardware security, hazard-free design

1. Introduction

Secure applications such as smart cards require measures to resist attacks, e.g. Differential Power Analysis (DPA) [1, 2]. Dual-rail encoding provides a method to enhance the security properties of a system making DPA more difficult. As an example, in the design described in [3] the processor can execute special secure instructions. These instructions are implemented as dual-rail circuits, whose

switching activity is meant to be independent from data. Special types of CMOS logic elements have been proposed in [4], but this low-level approach requires changing gate libraries and hence is costly for a standard cell or FPGA user. A method using balanced data encoding together with self-timed design techniques has been proposed in [5, 6]. In recent work [7] a method integrated in a standard design flow was described. Independently, we proposed a different method strongly linked to the industry CAD tools and based upon synchronous dual-rail circuits operating under a special protocol [8, 9, 10].

All these methods improve certain aspects of security, but still suffer from vulnerability to fault injection attacks. The idea behind a fault injection attack is simple: to modify the behaviour of a circuit so that the secret data became “visible” at either the unprotected outputs or at a side-channel such as power waveform or EMI. In this paper we are looking at data exposure in the form of power supply current variations. A particular form of the fault injection attack we are concerned with is the illumination of the die surface by a thin laser beam. Such a beam potentially can be focused at an individual gate, causing its pull-up or pull-down transistors to “leak” the quiescent current, which is strongly related to the data at the output of the gate under attack. Depending upon the intensity of the beam, the fault may or may not change the logic behaviour of the circuit. We analyse the “signature” of the power source current and filter out its quiescent current (IDDQ) component. The switching protocol that makes our circuits secure is also used in the optimal filter for IDDQ signature.

Apart from the security enhancement the method provides *massive controllability* by implementing a special switching protocol for all gates, and *massive observability* by using IDDQ testing. I guarantees that all faults of the given class are detected within a bounded and very short period of time. Similar to traditional IDDQ testing methods [11, 12] it also covers many additional faults.

The rest of this paper is organised as follows: Section 2 describes the class of security circuits we are dealing with, Section 3 presents the new method for on-line IDDQ testing of these circuits and Section 4 draws the conclusions.

2. Security circuits

2.1. Return-to-zero dual-rail

Dual-rail code uses two rails with only two valid signal combinations $\{01, 10\}$, which encode values 0 and 1 respectively. Dual-rail code is widely used to represent data in self-timed circuits [13, 14], where a specific protocol of switching helps to avoid hazards. The protocol allows only transitions from all-zeroes $\{00\}$, which is a non-code word, to a *code word* and back to all-zeroes as shown in Figure 1(a); this means the switching is monotonic. The all-zeroes state is used to indicate the absence of data, which separates one code word from another. Such a state is often called a *spacer*.

An approach for automatic converting single-rail circuits to dual-rail, using the above signalling protocol, that is easy to incorporate in the standard RTL-based design flow has been described in [15]. Within this approach, called Null-Convention Logic [16] one can follow one of two major implementation strategies for logic: one is with full completion detection through the dual-rail signals (NCL-D) and the other with separate completion detection (NCL-X). The former one is more conservative with respect to delay dependence while the latter one is less delay-insensitive but more area and speed efficient. For example, an AND gate is implemented in NCL-D and NCL-X as shown in Figure 1(b,c) respectively. NCL methods of circuit construction exploit the fact that the negation operation in dual-rail corresponds to swapping the rails. Such dual-rail circuits do not have negative gates (internal negative gates, for example in XOR elements, are also converted into positive gates), hence they are race-free under any single transition.

If the design objective is only power balancing (as in our case), one can abandon the completion detection channels, relying on timing assumptions as in standard synchronous designs; thus saving a considerable amount of area and power. This approach was followed in [8], considering the circuit in a clocked environment, where such timing assumptions were deemed quite reasonable to avoid any hazards in the combinational logic. Hence, in the clocked environment the dual-rail logic for an AND gate is simply a pair of AND and OR gates as shown in Figure 1(d).

The above implementation techniques certainly help to balance switching activity at the level of dual-rail nodes. Assuming that the power consumed by one rail in a pair is the same as in the other rail, the overall power consumption is invariant to the data bits propagating through the dual-rail circuit. However, the physical realisation of the rails at the gate level is not symmetric, and experiments with these dual-rail implementations show that power source current leaks the data values.

An example of dual-rail flip-flop design can be found in

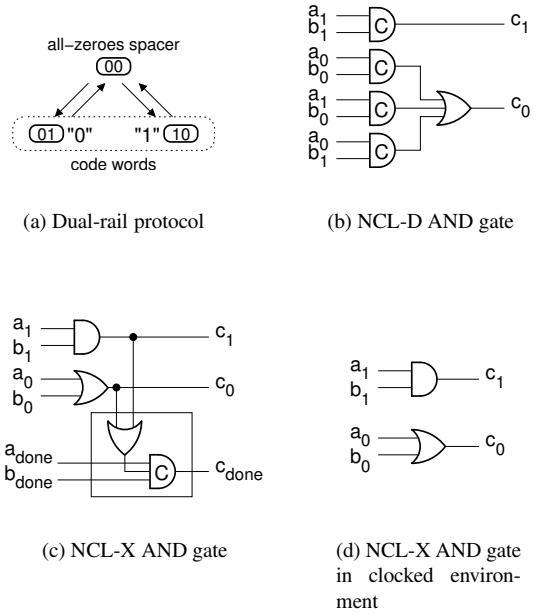


Figure 1. Single spacer dual-rail

[9, 10].

2.2. Alternating spacer dual-rail protocol

In order to balance the power signature we propose to use two spacers [9] (i.e. two spacer states, $\{00\}$ for *all-zeroes spacer* and $\{11\}$ for *all-ones spacer*), resulting in a dual spacer protocol as shown in Figure 2. It defines the switching as follows: *spacer* \rightarrow *code word* \rightarrow *spacer* \rightarrow *code word*. A possible refinement for this protocol is the *alternating spacer protocol* shown in Figure 2. The advantage of the latter is that all bits are switched in each cycle of operation, thus opening a possibility for perfect energy balancing between cycles of operation.

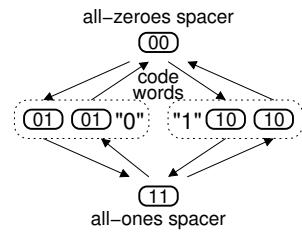


Figure 2. Alternating spacer dual-rail protocol

As opposed to single spacer dual-rail, where in each cycle a particular rail is switched up and down (i.e. the

same gate always switches), in the alternating spacer protocol both rails are switched from *all-zeroes spacer* to *all-ones spacer* and back. The intermediate states in this switching are *code words*. In the scope of the entire logic circuit, this means that for every computation cycle we always fire all gates forming the dual-rail pairs.

This protocol is enforced by non-standard flip-flops as described in [9, 10].

2.3. Alternating spacer dual-rail circuits

In CMOS a positive gate is usually constructed out of a negative gate and an inverter. Use of positive gates is not only a disadvantage for the size of dual-rail circuit, but also for the length of the critical path. Negative gate optimisation of our circuits [8] improves both the speed and area metrics.

The alternating spacer dual-rail circuits are identical in their combinational part to the return-to-zero logic, e.g. Figure 1(d). The difference is only in the flip-flops [9], which generate all-zeroes spacers in all odd clock cycles and all-ones spacers in all even clock cycles. The operation of the AND gate as in Figure 1(d) under such a protocol is shown in Figure 3.

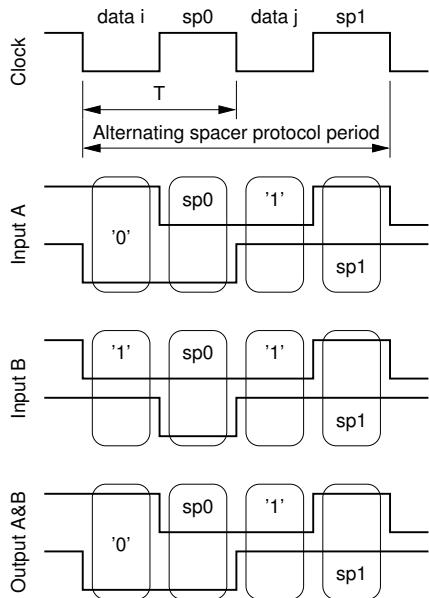


Figure 3. Alternating spacer dual-rail AND-gate in the clocked environment

Our security circuits are automatically generated by replacing all gates in a single-rail netlist by their dual-rail counterparts; this is done for all flip-flops and logic gates. The combinational logic produced by such a replacement

comprises positive gates only (inverters are represented as crossovers between the rails). A useful property of the positive logic (regardless of data encoding) is that if all-ones are applied to the primary inputs, then they propagate to all wires within the circuit. The same is true for the all-zeroes input pattern. Thus, no provisions are needed to ensure spacer propagation. Then the combinational logic is optimised w.r.t. negative gates as illustrated in Figure 4 and described in detail in [9].

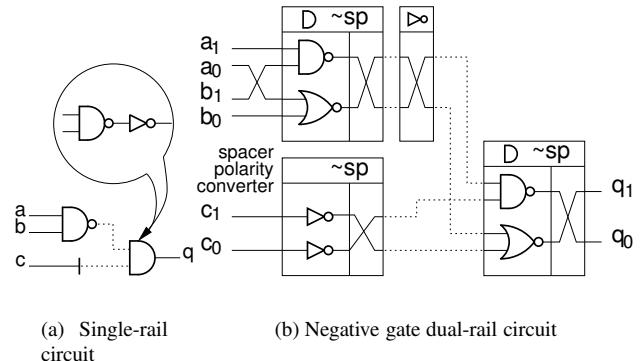


Figure 4. Constructing negative gate dual-rail circuit

Dotted lines in the single-rail circuit, Figure 4(a), indicate the future position of dual-rail signals with inverted spacers (dual-rail data is not inverted as this effect is corrected by rail crossover). The bar on the wire is the location of a spacer polarity converter. The circuit in Figure 4(b) is the result of the conversion.

2.4. Energy balancing

In [9] we introduce two important security characteristics of *energy imbalance* and *exposure time* which in this paper serve as a baseline for the testing method.

Energy imbalance (further referred to as *imbalance*) can be measured as the variation in energy consumed by a circuit processing different data. If e_1 and e_2 are the energy consumption under two input patterns, then the numerical value of imbalance is calculated as:

$$d = \frac{|e_1 - e_2|}{e_1 + e_2} \cdot 100\%$$

The imbalance values can be as large as 11% for some standard gates under certain operational conditions.

The *exposure time* is the time during which the imbalance caused by data is exhibited.

The alternating spacer dual-rail circuits have a nice property of the bounded exposure time, which is illustrated in

Figure 5. In this experiment we use a 2-input AND element converted to the alternating spacer dual-rail circuit, optimised w.r.t. negative gates and implemented in AMS- 0.35μ technology. The figure shows the energy imbalance, which occurs for about one half of the clock period (2ns). In slower circuits, such as those used in smart cards, the upper bound becomes exactly one half of the clock period.

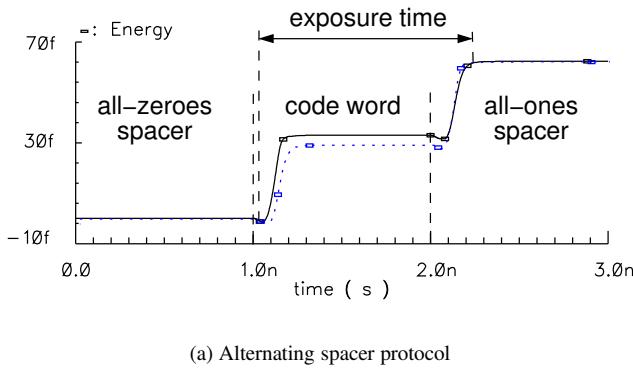


Figure 5. Exposure time for the alternating spacer AND2 gate

In this paper we use this property to separate the random effects of data from the effects of a fault.

3. At-speed IDDQ testing

3.1. Benchmark

The four-byte multiplier which is a part of AES [17, 18] is chosen as a benchmark for our experiments. It is a combinational circuit comprising 294 logic gates. In this paper we restrict ourselves to study of combinational circuits only, and the main reason for this is that we use in our library under optimised flip-flops based upon standard cells. These flip-flops will be optimised at the transistor level and this is a subject of future work.

The multiplier implements the MixColumn operation in AES, which maps one column of the input state (4 by 4 matrix) to a new column in the output state. The multiplier computes one output byte of the MixColumns operation based on a four byte input. This is done by considering each column as a polynomial over $GF(2^8)$ and multiplying it with the constant polynomial $a(x)$:

$$a(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\} \text{ modulo } x^4 + 1$$

$$a(x) = (\{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\}) \text{ modulo } (x^4 + 1).$$

This means that four multipliers are needed to generate a whole column or reuse of the same multiplier four times. The multiplier also implements the InvMixColumns transformation where the constant polynomial changes to $d(x)$:

$$d(x) = \{0B\}x^3 + \{0D\}x^2 + \{09\}x + \{0E\} \text{ modulo } x^4 + 1$$

$$d(x) = (\{0B\}x^3 + \{0D\}x^2 + \{09\}x + \{0E\}) \text{ modulo } (x^4 + 1).$$

The multiplier circuit was synthesised from the Open Core specifications by using Synopsis tools and then converted into dual-rail by our custom tool described in [9].

3.2. IDD signature of a fault

As our circuits exhibit no long-term imbalance, the power consumption current (IDD), more precisely the mean IDD value, is constant from one protocol cycle to another in absence of faults. The power consumption is defined by the transient currents (IDDT) of the gates. If a fault causing increased quiescent current (IDDQ) occurs, then this will be added to the IDDT. So, if the IDD increases, this is the indication of a fault.

Unfortunately, such an intuitive idea has its flaws. First, the IDDT in a large circuit can be so high that it may be difficult to detect the relative IDD increase due to a single fault. Second, a stuck-at fault changes the logic behaviour of a circuit. In a positive logic circuit, and hence in an alternating spacer circuit, it reduces switching activity. As the switching activity is strongly related to IDDT, this effect may compensate for the increase in IDDQ. In our approach these issues are resolved by using optimal filters “tuned” to pick up the IDDQ waveform. The filtration is possible as the alternating spacer protocol modulates the IDDQ of each single fault, and the modulation law is known.

The fault model we use includes all single stuck-at faults at interconnect between logic gates, i.e. this is the output single stuck-at model. It will be shown that many multiple faults are also covered.

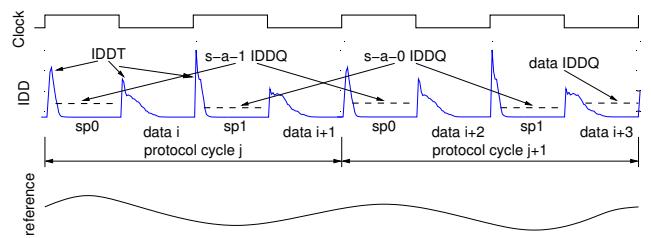


Figure 6. Generalised IDD

Figure 6 shows the power current of the dual-rail circuit under the alternating spacer protocol. One protocol cycle includes two clock periods. In the first period the spacer is all-zeroes (denoted as sp0) and in the second period it is

all-ones (sp1). This is indeed only true for the spacer polarities at the primary inputs/outputs. The internal signals, due to negative gate optimisation may have the opposite spacer polarities. For simplicity, in the further discussion we will assume that all gates are positive, although the actual experiments were conducted using optimised circuits. Data in each clock period is different and it is denoted as “data i” to “data $i+3$ ”. The IDD plot indicates four current peaks (transient current IDDT) per each protocol cycle. All peaks are different in shape and the area under the graph, they correspond to four transitions forming the full protocol cycle (see Figure 2). One can observe that the transitions from data into a spacer are short. This is due to early propagation effects, which *always* take place in these transitions. The transitions from a spacer into data are wider and have distinct shapes. This shape is formed by the computations performed in the logic gates. There is less room for the early propagation here, however it still may take place depending upon the function of the block and the data values. Each pair of subsequent spacer-to-data and data-to-spacer IDDT peaks have the same area under graph if compared to the corresponding pair within any other protocol cycle. This is due to the properties of the alternating spacer protocol. The energies of the first and the second halves of each protocol cycle are also almost identical, at least in absence of faults.

If, however, a single fault occurs, its IDDQ contributes differently to the different halves of protocol cycles. As our fault model includes only stuck-at faults on interconnect, they are guaranteed to be activated by either sp0 or sp1 spacer. Furthermore, if sp0 activates the fault, then under sp1 it will not be activated. The opposite is also true. Thus, the signature of a fault is one spacer activation within each protocol period. These circuits have the property of *massive controllability*, which provides activation of all faults within two subsequent clock periods regardless of data. Each s-a-1 fault is activated by sp0 and each s-a-0 fault is activated by sp1. The IDDQ currents corresponding to the faults of different polarities are shown in Figure 6 as dashed lines. The dashed lines denoted as “s-a-1 IDDQ” and “s-a-0 IDDQ” are the fault currents activated by the corresponding spacers. The label “data IDDQ” corresponds to the fault current activated by a data state. Indeed, under a data state half of wires in a dual-rail circuit have 1 and the other half have 0 values, so they activate the corresponding faults. Fault activation by data, however, cannot be guaranteed in bounded time, because these values are data-dependent and in general case random.

Spectral representation of the IDD of our multiplier running at clock speed of 100MHz and random data is shown in Figure 7(a). The tallest peak is 200MHz, the frequency of transitions irrespective to their designation. The frequency of 100MHz is the signature of gate imbalance. The zero frequency is the mean value of IDD. Finally, 50MHz is the

frequency of spacers of a particular polarity, this is where the fault IDDQ is activated. Figure 7(b) shows the difference between the spectrums with and without a fault. The fault shows up at 0 and 50MHz. The last diagram in Figure 7(c) shows the same difference in relation to the absolute value of the spectrum without fault. At zero frequency the relative difference is 12% (which is not clearly seen in this diagram). This indicates a potential problem with fault detection by IDD mean value. At the frequency 50MHz, however, the increase is about 180%, which is good enough for driving a coarse comparator.

We use the above observations for a quick evaluation of our idea only. So far we were looking at the amplitude spectrum only, which does not represent any information about the phase of signal components. As Figure 6 shows, the fault IDDQ and the IDDT take place in different phases of the protocol sequence, and the experiments described below utilise this additional information.

The principle of cross correlation is used in our approach in order to filter out the fault IDDQ signature. The top diagram in Figure 8 is a fragment of the IDD waveform obtained by Spice simulation of the multiplier benchmark, it has the mean value subtracted in order to simplify the following processing. There were 1000 clock cycles simulated in total and the first protocol cycle was discarded. The reference signal is defined as a sine wave of half clock frequency. The initial phase is chosen so, that the maximum of the wave approximately corresponds to sp0 spacer and the minimum to sp1 spacer (the phase of the reference signal in Figure 8 corresponds to the time in the IDD plot). Then the cross correlation between the IDD wave and the reference signal is calculated (see the Cross Correlation plot, no fault, random data), and the initial phase of the reference signal is adjusted to produce 0 cross correlation under 0 lag.

The effect of data on the cross correlation diagram was investigated. The maximum deviation from the first plot is obtained under the data producing the most asymmetrical IDD waveform (see “No fault, selected data”).

Then a fault was introduced. At first we placed the fault at the output of the circuit in order to have the minimum impact on the switching activity. This produces the cross correlation curve labelled as “Fault, small cone” (the cone is a set of the successor gates of the fault location). At 0 lag the cross correlation value is about 5.5 times greater than the maximum value produced by the “bad” data without faults. This gives a very good margin for fault detection. Then we moved the fault close to the input of the circuit (the third bit in one operand), thus creating the largest cone with reduced switching activity. The result of this experiment is labelled as “Fault, large cone”. The result of cross correlation shows even greater value of 7 at lag 0. We explain this effect by the IDDT imbalance due to violation of the alternating spacer protocol in the cone. An interesting observation is that the

switching activity reduction in the cone reduces the mean value of IDD (frequency zero), but in the same time introduces more imbalance, which is good for the proposed fault detection method. In this sense our method uses not only the quiescent current, but also to some extent the transient current to detect faults.

3.3. Fault model extension

Although we designed the method for single stuck-at faults, it should also work for multiple unidirectional faults. By unidirectional we understand all faults increasing the IDDQ during either sp0 or sp1 spacer of the protocol. If the faults are not unidirectional, then they can cancel the effect of each other. Furthermore, the analysis of such faults should take into account their location w.r.t. the cones generated by them. This analysis is the subject of the future work.

We have also conducted a preliminary investigation on detection of parametric faults. In our setup we modeled a leakage fault as a small open transistor, e.g. 1μ pull-up transistor at the output of an inverter that uses 2μ pull-up and 1μ pull-down devices. The effect of such a fault could be reliably detected by our method producing the effect of about one-half of the short-circuit fault.

4. Conclusions

The proposed IDDQ method is applicable to a special class of security circuits: synchronous, dual-rail, return-to-spacer protocol with alternating spacer states. The data-independent power consumption of such circuits opens an opportunity for on-line application of IDDQ testing.

The method includes the provision of *massive controllability*, i.e. all potential faults are activated within two clock periods, and *massive observability* (IDDQ measurement). This can be useful for both production testing (minimisation of time on tester and minimisation of the number of test pins), and on-line testing. In the on-line testing application the method guarantees a bounded and very short self-test period, independent from data, which is different from many other on-line testing methods. Furthermore, the method also detects parametric (leakage) faults, which are often used in attacks on security devices. In this sense, the proposed approach directly detects the information leakage through a side channel.

It was shown that the increase of the mean IDD value due to a fault cannot be reliably used for fault detection, because of two reasons. First, a stuck-at fault in the given class of circuits reduces switching activity and, thus, the transient current. This effect may compensate for the quiescent current increase. Second, the transient current in large circuits

is so high, that the relative increase in the overall current due to IDDQ is small. In our example it was only 12%.

The method is based upon the cross correlation operation and it is close to the optimal filtration approach. It is different from the optimal filter as it uses a sine wave as a reference signal. In the optimal filter it would be a pattern matching the shape of the IDDQ. We believe that using signals more complex than a sine wave will significantly increase the complexity of the filter, giving little benefits. More experiments are needed to support this statement.

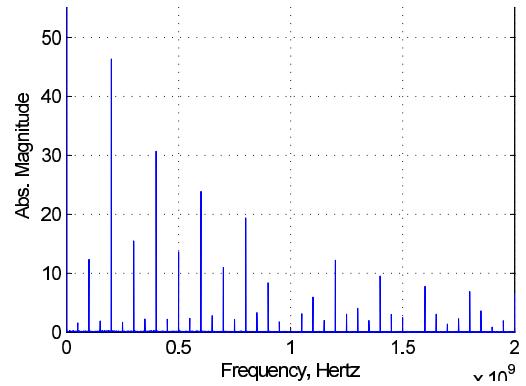
The method was applied to a benchmark circuit, which is an important part of AES cryptographic block. The circuit comprises 294 logic gates. It is important to do more case studies in order to determine the maximum size of a circuit served by a single current sensor. The design of the current sensor itself and the signal processing part are the subject of future work.

Acknowledgements: We are grateful to A. Yakovlev, G. Russell and A. Koelmans for useful comments, and to D. Sokolov for designing the tool for synthesis of the dual-rail security circuits. EPSRC supports this work via GR/S81421 (SCREEN).

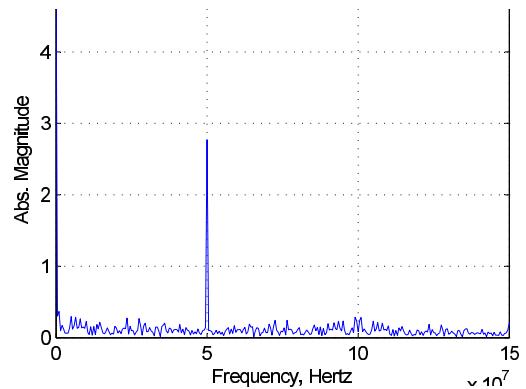
References

- [1] P.Kocher, J. Jaffe, B. Jun, "Differential Power Analysis". Proc. Crypto, 1999.
- [2] T.Messerges, E.Dabbish, R.Sloan: "Examining smart-card security under the threat of power analysis attacks". IEEE Trans. on Computers, 2002, 51(5), pp. 541–552.
- [3] H.Saputra, N.Vijaykrishnan, M.Kandemir, M.J.Irwin, R.Brooks, S.Kim, W.Zhang: "Masking the energy behaviour of DES encryption". Proc. DATE, 2003.
- [4] K.Tiri, M.Akmal, I.Verbauwheide: "A dynamic and differential CMOS logic with signal independent power consumption to withstand differential power analysis on smart cards". Proc. ESSCIRC, 2002.
- [5] S.Moore, R.Anderson, P.Cunningham, R.Mullins, G.Taylor: "Improving smart card security using self-timed circuits". Proc. ASYNC, 2002, pp. 211–218.
- [6] Z.Yu, S.Furber, L.Piana: "An investigation into the security of self-timed circuits". Proc. ASYNC, 2003, pp. 206–215.
- [7] K.Tiri, I.Verbauwheide: "A logical level design methodology for a secure DPA resistant ASIC or FPGA implementation". Proc. DATE, 2004.

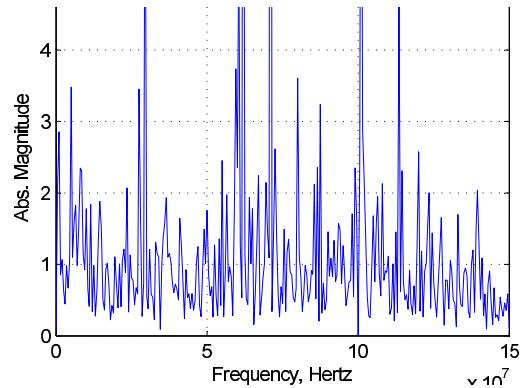
- [8] A.Bystrov, D.Sokolov, A.Yakovlev, A.Koelmans: “Balancing power signature in secure systems”. 14th UK Asynchronous Forum, 2003.
- [9] D.Sokolov, J.Murphy, A.Bystrov, A.Yakovlev: “Improving the security of dual-rail circuits”. Proc. CHES, 2004.
- [10] D. Sokolov, J. Murphy, A. Bystrov, A. Yakovlev: “Design and analysis of dual-rail circuits for security applications”. Accepted IEEE Trans. on Comput., 2004
- [11] M. W. Levi, "CMOS is most testable," in Int. Test Conf., 1981, pp. 217-220
- [12] Rajsuman, R.: “Iddq testing for CMOS VLSI”. Proceedings of the IEEE , Volume: 88 , Issue: 4 , April 2000, pp.544-568
- [13] V.Varshavsky (editor): “Self-timed control of concurrent processes” Kluwer, 1990 (Russian edition 1986).
- [14] I.David, R.Ginosar, M.Yoeli: “An efficient implementation of boolean functions as self-timed circuits”. IEEE Trans. on Computers, 1992, 41(1), pp. 2–11.
- [15] A.Kondratyev, K.Lwin: “Design of asynchronous circuits using synchronous CAD tools”. Proc. DAC, 2002, pp. 107–117.
- [16] K.Fant, S.Brandt: “Null Convention Logic: a complete and consistent logic for asynchronous digital circuit synthesis”. Proc. ASAP, IEEE CS Press, 1996, pp. 261–273.
- [17] National Institute Of Standards and Technology: “Federal Information Processing Standard 197, The Advanced Encryption Standard (AES)”. <http://csrc.nist.gov/publications/fips/fips197/fips197.pdf>, 2001.
- [18] R.Usselmann: “Advanced Encryption Standard / Rijndael IP Core”. <http://www.asic.ws/>.



(a) IDD frequency spectrum under fault



(b) spectral difference fault – no fault



(c) relative spectral difference

Figure 7. Power signatures in the frequency domain

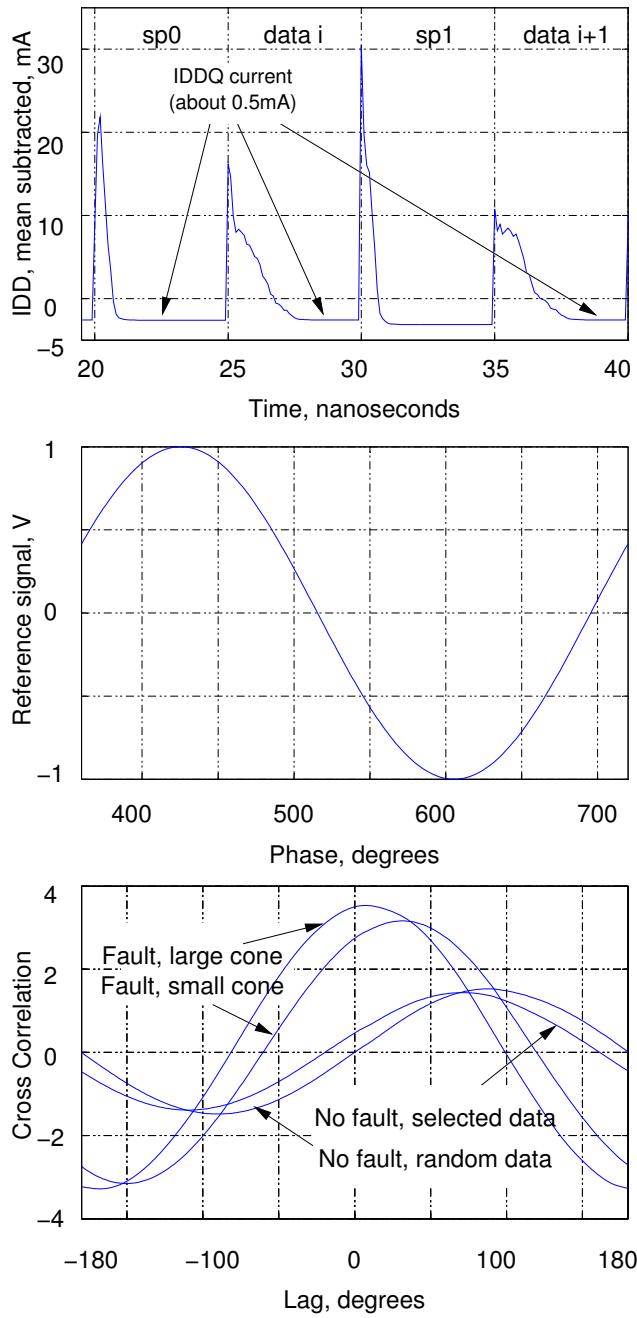


Figure 8. Fault IDQ detection by cross-correlation

Design flow for low-power and high-security based on timing diversity

Danil Sokolov, Alex Yakovlev
University of Newcastle upon Tyne, UK

Abstract

The BESSST tool kit is used for asynchronous system synthesis based on Petri Nets. It incorporates software tools for high-level partitioning, scheduling, direct mapping and logic synthesis. These are used to generate efficient speed-independent circuits from behavioural Verilog specification.

1. Introduction

In 1965 a co-founder of Intel Gordon Moore noticed that the number of transistors doubled every year since the invention of the integrated circuit. He predicted that this trend would continue for the foreseeable future [23]. In subsequent years the pace slowed down and now the functionality of the chip doubles every two years. However, the growth of circuit integration level is still faster than the increase in the designers productivity. This creates a design gap between semiconductor manufacturing capability and the ability of Electronic Design Automation (EDA) tools to deal with the increasing complexity.

One of the ways to deal with the increasing complexity of logic circuits is to improve the efficiency of the design process. In particular, design automation and component reuse help to solve the problem. Systems-on-Chip (SoC) synthesis has proved to be a particularly effective way in which design automation and component reuse can be facilitated. An important role in the synthesis of SoCs is given to the aspects of modelling concurrency and timing [16]. These aspects have traditionally been dividing systems into *synchronous* (or *clocked*) and *asynchronous* (or *self-timed*). The division has recently become fuzzier because systems are built in a mixed timing style: partly clocked and partly self-timed. The argument about the way how the system should be constructed, synchronously or asynchronously, is moving to another round of evolution. It is accepted that the timing issue should only be addressed in the context of the particular design criteria, such as speed, power, security, modularity, etc. Given the complexity of the relationship between these criteria in every single practical case, the design of an SoC is increasingly going to be a mix of timing styles. While industrial designers have a clear and established notion of how to synthesise circuits with a global clock using EDA tools, there is still a lot of uncertainty and doubt about synthesis of asynchronous circuits. The latter remains a hot research field captivating many academics and graduate students.

The main goal of this paper is to review a coherent subset of synthesis methods for self-timed circuits based primarily on a common underlying model of computation and using a relatively simple example in which these methods can be compared. Such a model is Petri nets, used with various interpretations. The Petri nets can play a pivotal role in future synthesis tools for self-timed systems, exhibiting advanced concurrency and timing paradigms. This role can be as important as that of a Finite State Machine (FSM) in designing clocked systems. To make this paper more practically attractive the use of Petri nets is considered in the context of a design flow with a front-end based on a hardware description language.

2. Petri nets

The convenient behavioural models for logic synthesis of asynchronous control circuits are 1-safe *Petri Net* (PN) and *Signal Transition Graph* (STG). The datapath operations can be modelled by *Coloured Petri Nets* (CPN).

A PN is formally defined as a tuple $\Sigma = \langle P, T, F, M_0 \rangle$ comprising finite disjoint sets of *places* P and *transitions* T , flow relation $F \subseteq (P \times T) \cup (T \times P)$ and initial marking M_0 . There is an arc between x and y iff $(x, y) \in F$. An arc from a place to a transition is called *consuming arc*, and from a transition to a place - *producing arc*. The *preset* of a node x is defined as $\bullet x = \{y \mid (y, x) \in F\}$, and the *postset* as $x\bullet = \{y \mid (x, y) \in F\}$. A *marking* is a mapping $M : P \rightarrow N$ denoting

the number of *tokens* in each place ($N = \{0, 1\}$ for 1-safe PNs). It is assumed that $\bullet t \neq \emptyset \neq t\bullet, \forall t \in T$. A transition t is *enabled* iff $M(p) \neq 0, \forall p \in \bullet t$. The evolution of a PN is possible by *firing* the enabled transitions. Firing of a transition t results in a new marking . $M' : M'(p) = \begin{cases} M(p) - 1, & \forall p \in \bullet t, \\ M(p) + 1, & \forall p \in t\bullet, \\ M(p), & \forall p \notin \bullet t \cup t\bullet \end{cases}$.

An extension of a PN model is a *contextual net* [22]. It uses additional elements such as *non-consuming arcs*, which only control the enabling of a transition and do not consume tokens. The reviewed methods use only one type of non-consuming arcs, namely *read-arcs*. A set of read-arcs R can be defined as follows: $R \subseteq (P \times T)$. There is an arc between p and t iff $(p, t) \in R$.

A *Labelled Petri Net* (LPN) is a PN whose transitions are associated with a labelling function λ , i.e. $LPN = \langle P, T, F, R, M_0, \lambda \rangle$.

An STG is an LPN whose transitions are labelled by signal events, i.e. $STG = \langle P, T, F, R, M_0, \lambda \rangle$, where $\lambda : T \rightarrow A \times \{+, -\}$ is a labelling function and A is a set of signals. A set of signals A can be divided into a set of *input signals* I and a set of *output and internal signals* O , $I \cup O = A$, $I \cap O = \emptyset$. Note that a set of read-arcs R has been included into the model of STG, which is an enhancement w.r.t. [25].

An STG is *consistent* if in any transition sequence from the initial marking, rising and falling transitions of each signal alternate.

A signal is *persistent* if its transitions are not disabled by transitions of another signal.

An STG is *output persistent* if all output signals are persistent and input signals cannot be disabled by outputs.

An STG is *delay insensitive (DI) to inputs* if no event of one input signal is switched by an event of another input signal.

A CPN is a formal high-level net where tokens are associated with data types [17]. This allows the representation of datapath in a compact form, where each token is equipped with an attached data value.

3. Asynchronous circuit design flow

For a designer it is convenient to specify the circuit behaviour in a form of a high-level HDL, such as Verilog, VHDL, SystemC, Balsa, etc. The choice of HDL is based on personal preferences, EDA tool availability, commercial, business and marketing issues [29].

There are two main approaches to synthesis of asynchronous circuits from high-level HDLs: *syntax-driven translation* and *logic synthesis*.

In *syntax-driven translation* the language statements are mapped into circuit components and the interconnect between the components is derived from the syntax of the system specification. This approach is adopted by Tangram [24] and Balsa [1] design flows. The initial circuit specification for these tools is given in the languages based on the concept of processes, variables and channels, similar to Communicating Sequential Processes (CSP) [13].

In *logic synthesis* the initial system specification is transformed into an intermediate behavioural format convenient for subsequent verification and synthesis. This approach is used in PipeFitter [2], TAST [11], VeriSyn [27], where Petri nets are used for intermediate design representation. Other examples are MOODs [26] and CASH [34]. The former starts from VHDL and uses a hardware assembly language ICODE for intermediate code. The latter starts from ANSI-C and uses Pegasus dataflow graph for intermediate representation, which is further synthesised into control logic for Micropipelines [32].

Some tools do not cover the whole design, but can be combined with the other tools to support the coherent design flow. For example, Gate Transfer Level (GTL) [28], Theseus Logic NCL-D and NCL-X [20] are developed for synthesis of asynchronous data path from Register Transfer Level (RTL) specifications. Other tools, such as Minimalist [12], 3D [6] and Petrify [8] are aimed at asynchronous controller synthesis from intermediate behavioural specifications. In turn, controller synthesis tools, can be combined with decomposition techniques [35] to reduce the complexity of the specification.

The BESS design flow whose diagram is shown in Figure 1 is based on a logic synthesis approach. The initial specification in a high-level HDL (System-C, Verilog or VHDL) is first split into two parts: the specification of control path and specification of the data path. Both parts are synthesised separately and subsequently merged into the system implementation netlist. The industrial EDA place and route tools can be used to map the system netlist into silicon. The existing simulation EDA tools can be reused for the behavioural simulation of the initial system specification. These tools can be also adopted for timing simulation of the system netlist back-annotated with timing information from the layout.

The variations in the design flow appear in the way of (a) extracting the specifications of control and data paths from the system specification; (b) synthesis of the data path; (c) synthesis of the control path either by direct mapping or by explicit logic synthesis . The following sections consider each of these issues separately.

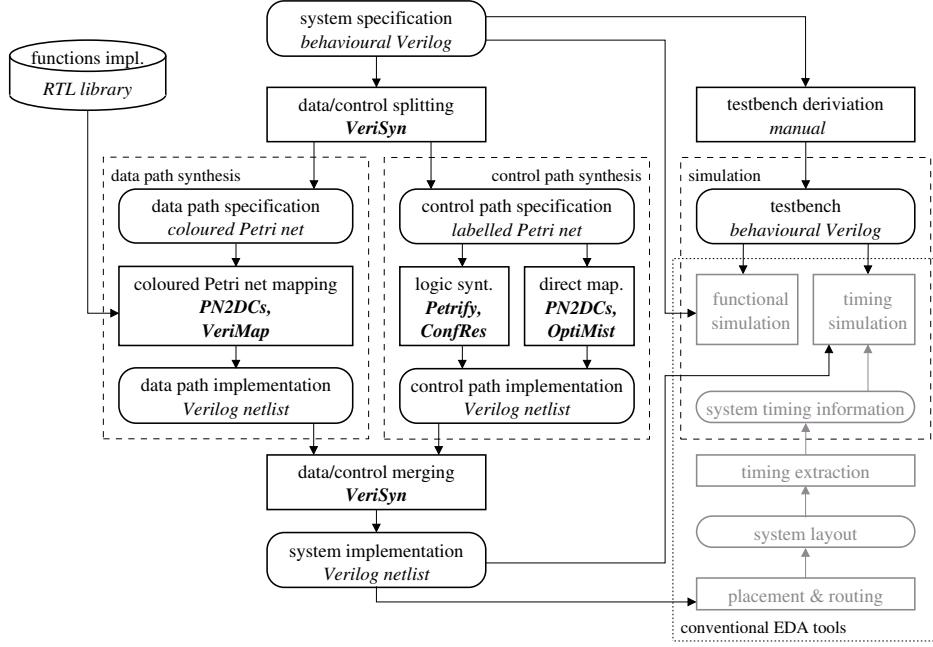


Figure 1. BESST design flow

4. Splitting of control and data paths

The first step in the logic synthesis of a circuit is the extraction of control and data paths specifications from the high-level description of the system. Often the partitioning of the system is performed manually by the designers. However, this might be impracticable for a large system or under a pressure of design time constraints. At the same time, the tools automating the extraction process are still immature and require a lot of investment to be used outside a research lab.

The primary output of the VeriSyn tool is a *global net*, which is a PN whose transitions are associated with the atomic operations and whose places divide the system functioning into separate stages. Initially the global net transitions represent complex operations corresponding to the high-level functional blocks (modules in Verilog, processes in VHDL, functions in SystemC). Then the global net transitions are refined iteratively, until all transitions represent atomic operations. This global net is used to derive a Labeled Petri net (LPN) for control path and a Coloured Petri net (CPN) for data path. The interface between control and data paths is modelled by a *local control net*, which connects the generated LPN and CPN. These PNs are subsequently passed to the synthesis tools for optimisation and implementation.

The derivation of a global net and the extraction of an LPN for control path and a CPN for the data path is illustrated on the GCD benchmark. For this the VeriSyn tool is applied to the following Verilog specification of GCD algorithm:

```

module gcd(x, y, z);
  input [7:0] x;
  input [7:0] y;
  output reg [7:0] z;
  reg [7:0] x_reg, y_reg;
  always @(x or y)
  begin
    x_reg = x;
    y_reg = y;
    while (x_reg != y_reg)
    begin
      if (x_reg > y_reg)
        x_reg = x_reg - y_reg;
      else
        y_reg = y_reg - x_reg;
    end

```

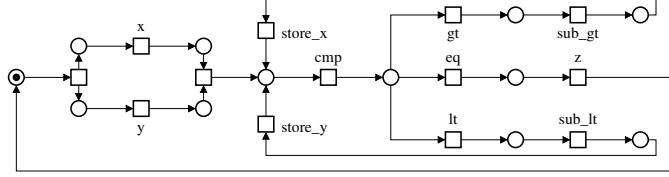


Figure 2. Global net for GCD algorithm

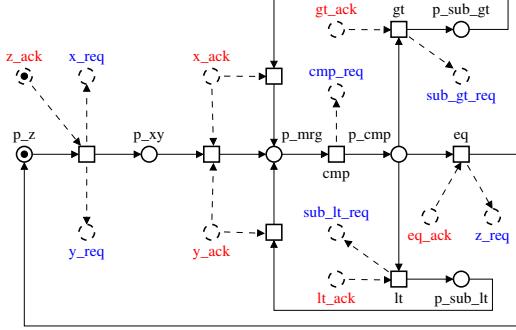


Figure 3. LPN for GCD control path

```

z = x_reg;
end
endmodule

```

As the benchmark contains one process only, the global net consists of one transition representing this process. This transition is refined by parsing the system specification and using an ASAP scheduling algorithm.

The refined model of the system is shown in Figure 2. Two wait-statements for synchronisation on the x and y channels are scheduled concurrently. It is possible because both x and y are defined as inputs and are independent. The input operation, the while-loop and the result output on the z channel are scheduled in sequence. The while-loop is subject to further refinement together with the nested if-statement, during which the conditions of while-loop and if-statement are merged. The labels gt , eq and lt correspond to the result of comparison between x and y values and stand for ‘greater than’, ‘equal’ and ‘less than’ respectively. The assignment of the subtraction result to a signal is split into the subtraction operation (sub_gt , sub_lt) and storage of the result ($store_x$, $store_y$). The transitions of the global net are given short and distinctive labels convenient for further reference. The global net is ready for the extraction of the control path LPN and the data path CPN.

The LPN for the GCD control path produced by the PN2DCs tool is shown in Figure 3 using the solid lines. The dashed arcs and places represent the local control net. Signals z_ack and z_req compose the handshake interface to the environment. When set, the z_req signal means the computation is complete and output data is ready to be consumed. The z_ack signal is set when the output of the previous computation cycle is consumed and the new input data is ready to be processed.

The data path CPN generated by VeriSyn is presented in Figure 4 using the solid arcs, places and transitions. The dashed arcs and places represent the local control net. All the communication between the control and data paths is carried out by means of this net, as shown in Figure 5. For example, when the z_ack signal is received, the control generates x_req and y_req signals which enable the MUX_x_0 and MUX_y_0 transitions in the data path. When the multiplexing is finished the values of x and y are stored using REG_x and REG_y respectively. The data path acknowledges this by x_ack and y_ack signals. The acknowledgement signals enable the $dum1$ transition in the control path LPN. After that, the control path requests the comparison operation by means of the cmp_req signal. When the comparison is complete

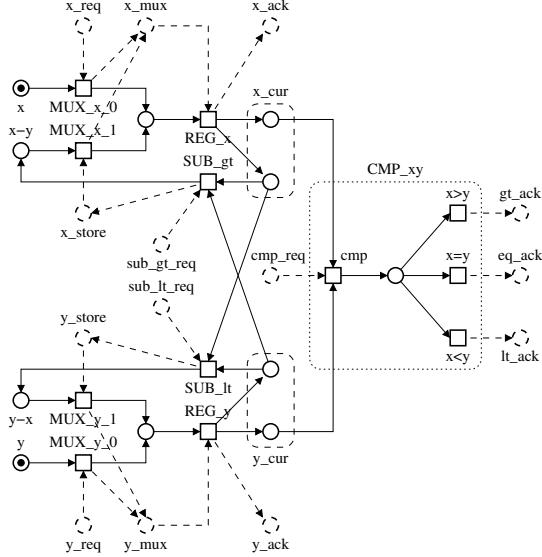


Figure 4. CPN for GCD data path

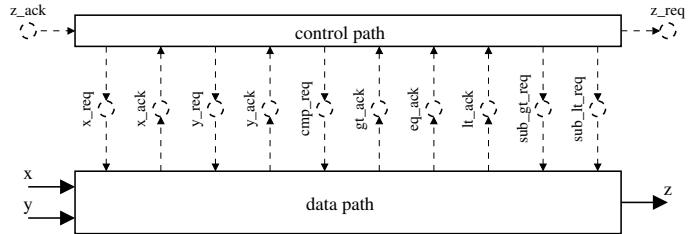


Figure 5. GCD control / data path interface

in the data path, one of the signals *gt_ack*, *eq_ack* or *lt_ack* is returned to the control. If *gt_ack* is received, the control path generates *sub_gt_req* request, which activates *SUB_xy* transition in the data path. This results in subtracting the current value of *y* from *x* and storing the difference using *REG_x* transition. The data path acknowledges this by *x_ack* and the comparison operation is activated again. If the *lt_ack* signal is issued by the data path then the operation of the system is analogous to that of *gt_ack*. However, as soon as *eq_ack* is generated, the control path issues the *z_req* signal to the environment, indicating that the calculation of GCD is complete.

Note that the local control net *x_mux* between *MUX_x_0* and *REG_x* does not leave the data path, thereby simplifying the control path. Similarly, other signals, *y_mux*, *x_store* and *y_store*, in the local control net are kept inside the data path.

The same control path LPN, data path CPN and local control net can be obtained from VHDL or SystemC specifications of GCD algorithm. These PNs are passed to synthesis tools for deriving the implementation of control and data paths.

5. Synthesis of data path

The method of data path synthesis employed in PN2DCs is based on the mapping of CPN fragments into predefined hardware components. A part of the library of such components and corresponding CPN fragments are shown in Figure 6. The solid places and arcs in the CPN column correspond to data inputs and outputs; the dashed arcs and places denote the control signals (request and acknowledgement).

A block diagram for the GCD data path is presented in Figure 7. It is mapped from the CPN specification shown in Figure 4. The CPN is divided into the following fragments, which have hardware implementations in the library shown in Figure 6: 2 multiplexers, 2 registers, 1 comparator and 2 subtractors. These hardware components are connected according

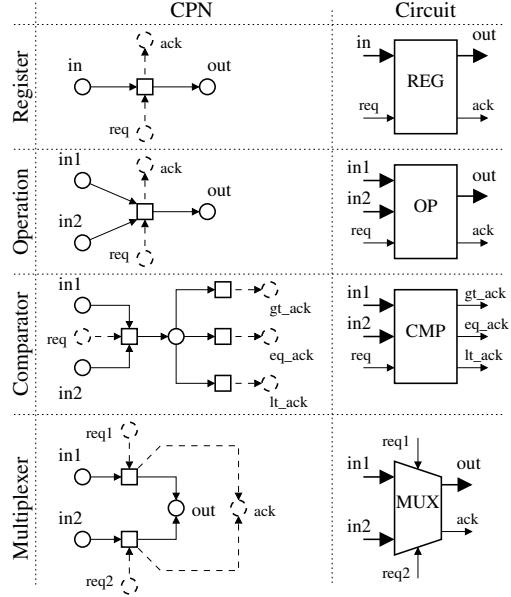


Figure 6. Mapping from CPN into circuit

to the arcs between the corresponding fragments of the CPN. To save the hardware, the output z is not latched in its own register. Instead it is taken from the register y and is valid when the controller sets the z_req signal.

If the library of data path components does not have an appropriate block, the latter should be either manually constructed or automatically generated from RTL, using the VeriMap software tool [31]. Its input is a single-rail circuit synthesised from behavioural specification by an RTL tool. The method employs dual-rail encoding and monotonic switching which facilitate the hazard-free logic. The completion detection built on dual-rail logic provides the average case performance of data path components. The method extends the traditional single-spacer dual-rail encoding with two spacers (all-zeros and all-ones) alternating in time. The alternating spacers provides strictly periodic refreshing of all wires in the data path which is beneficial for testing, dynamic logic and security applications. The VeriMap tool is compatible with the conventional EDA design flow at the level of Verilog netlists.

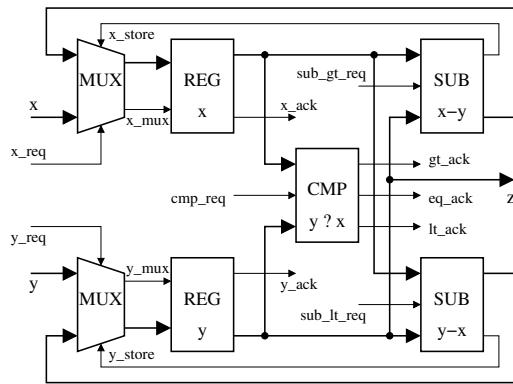


Figure 7. GCD data path

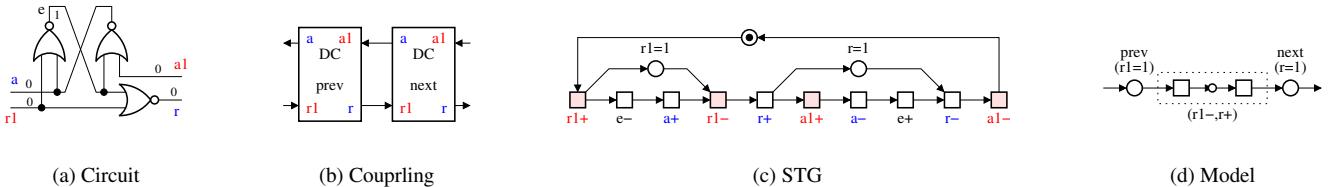


Figure 8. David cell

6. Direct mapping of control path

The main idea of the *direct mapping* is that a graph specification of a circuit can be translated directly (without computationally hard transformations) into the circuit netlist in such a way that the graph nodes correspond to the circuit elements and graph arcs correspond to the interconnect. Direct mapping approach originates from [15], where a method of *the one-relay-per-row realisation* of an asynchronous sequential circuit is proposed. This approach is further developed in [33] where the idea of the *1-hot state assignment* is described. The 1-hot state assignment is then used in the method of concurrent circuit synthesis presented in [14].

A circuit diagram of a *David cell* (DC) is shown in Figure 8(a). DCs can be coupled using a 4-phase handshake protocol, so that the interface $\langle a1, r \rangle$ of the previous stage DC is connected to the interface $\langle a, r1 \rangle$ of the next stage as shown in Figure 8(b). Output r of a DC is used to model the marking of the associated PN place. If the output r is low the corresponding place is empty and if it is high then the corresponding place is marked with a token. The operation of a single DC is illustrated in Figure 8(c). The transitive places *prev* and *next* represent the high level of signals $r1$ and r respectively. Their state denotes the marking of places associated to previous stage and next stage DCs, see Figure 8(d). The dotted rectangle depicts the transition between *prev* and *next* places. This transition contains an internal place, where a token ‘disappears’ for the time $t_{r1 \rightarrow r+}$. In most cases this time can be considered as negligible, because it corresponds to a single two input NOR-gate delay.

The circuits built of DCs are speed independent and do not need fundamental mode assumptions. On the other hand, these circuits are autonomous (no inputs/outputs). The only way of interfacing them to the environment is to represent each interface signal as a set of abstract processes, implemented as request-acknowledgement handshakes, and to insert these handshakes into the breaks in the wires connecting DCs. This restricts the use of DCs to high-level design.

6.1. Direct mapping from LPNs

The PN2DCs tool [27] uses the *direct mapping from LPNs* approach based on [19]. In this approach the places of the control path LPN are mapped into DCs. The request and acknowledgement functions of each DC are generated from the structure of the LPN in the vicinity of corresponding place as shown in Figure 9. The request function of each DC is shown in its top-left corner and the acknowledgement function in its bottom-right corner.

The GCD control path described by the LPN in Figure 3 is mapped into the netlist of DCs shown in Figure 10. Each DC in this netlist corresponds to the LPN place with the same name. The requests to the data path (x_{req} , y_{req} , cmp_{req} , sub_gt_{req} , sub_lt_{req} , z_{req}) and the acknowledgements from the data path (x_{ack} , y_{ack} , gt_{ack} , lt_{ack} , eq_{ack} , z_{ack}) are defined by the local control net places with the same names.

There is a space for further optimisation of the obtained control path circuit. For example, two DCs p_mrg and p_cmp could be merged. Also the request function of the p_mrg DC could be simplified by introducing additional memory elements. However, the obtained circuit is implementable in standard libraries e.g. AMS already and exhibits good size and speed characteristics. It consists of 120 transistors and has the worst case latency of 4 negative gates in the $x_ack \rightarrow cmp_req$ path.

6.2. Direct mapping from STGs

The method of direct mapping from STGs proposed in [3] is implemented in the OptiMist software tool [30]. The tool first converts the system STG into a form convenient for direct mapping. Then it performs optimisation at the level of specification

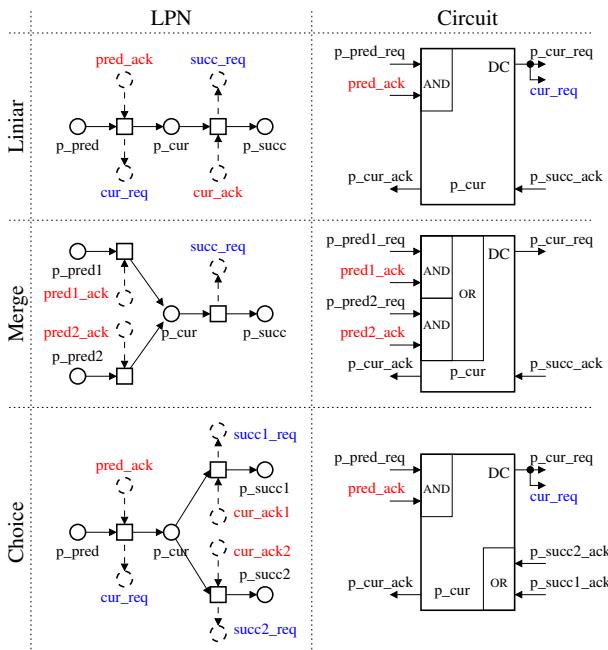


Figure 9. Mapping of LPN places into DCs

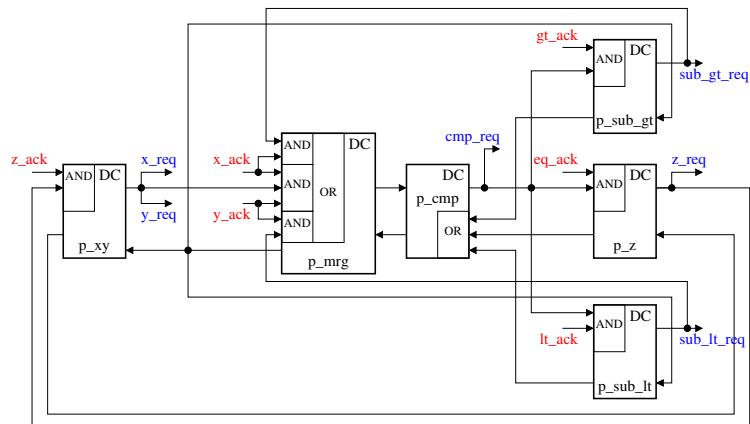


Figure 10. GCD control path scheme

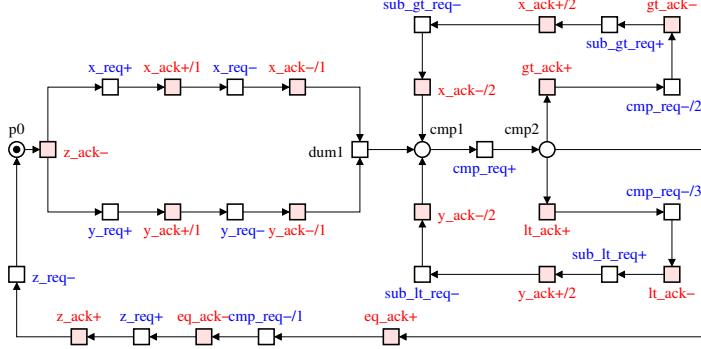


Figure 11. GCD control path STG

and finally maps the optimised STG into circuit netlist. In order to transform the initially closed system specification into the open system specification, the concepts of *environment tracking* and *output exposure* are applied. These concepts can be applied to an STG that is consistent, output persistent and delay insensitive to inputs.

Consider the application of the OptiMist tool to the example of the GCD control path. Its STG is obtained by refining the LPN generated from the HDL specification by PN2DCs tool. In order to produce the control path STG shown in Figure 11 the events of the LPN are expanded to a 4-phase handshake protocol. After that, the GCD datapath schematic shown in Figure 7 is taken into account to manually adjust the STG to the datapath interface. In the modified STG the request to the comparator *cmp_req* is acknowledged in 1-hot code by one of the signals: *gt_ack*, *eq_ack* or *lt_ack*. The request to the subtracter *sub_gt_req* is acknowledged by *x_ack*. This is possible because the procedure of storing the subtraction result into the register is controlled directly in the datapath and does not involve the control path. Similarly *sub_lt_req* is acknowledged by *ack_y*.

When the OptiMist tool is applied to the original STG of GCD controller it produces the device specification which is divided into a tracker and a bouncer parts, as shown in Figure 12. The bouncer consists of elementary cycles representing the outputs of GCD controller, one cycle for each output. The elementary cycles for the inputs are not shown as they belong to the environment. The tracker is connected to inputs and outputs of the system by means of read arcs, as it is described in the procedure of outputs exposure.

There are two types of places in the tracker part of the system: *redundant* (depicted as small circles) and *mandatory* (depicted as big circles). The redundant places can be removed without introducing a coding conflict while the mandatory places should be preserved. OptiMist tool determines the sets of redundant and mandatory places using the heuristics described in [30].

The first heuristic, most important in terms of latency, states that each place whose all preceding transitions are controlled by inputs and all successor transitions are controlled by outputs can be removed from the tracker. Removal of such a place does not cause a coding conflict as the tracker can distinguish the state of the system before the preceding input-controlled transitions and after the succeeding output-controlled transitions. However, a place should be preserved if any of its preceding transitions is a direct successor of a choice place. Preserving such a place helps to avoid the situation when the conflicting transitions (direct successors of the choice place) are controlled by the same signal. The removal of the redundant places detected by the first heuristic reduces both the size and the latency of the circuit.

The redundant places detected by the above heuristic in GCD example are *px1*, *py1*, *px3*, *py3*, *px5*, *py5*, *pgt3*, *plt3*, *pgt5*, *plt5*, *peq3* and *peq5*. The places *pgt1*, *plt1* and *peq1* which follow the choice place *cmp2* should be preserved. Their removal would cause an ambiguous situation when the first transitions to the three conflicting branches are controlled by the same signal *cmp_req=0*.

The next heuristic for redundant places detection traverses the chains of non-redundant places between input-controlled transitions. The traversing of a chain starts from the place after an input-controlled transitions and progresses in the direction of consuming-producing arcs. For each place in the chain it is checked if its removal causes a coding conflict. The potency of a coding conflict is checked assuming that all the places which are currently tagged as redundant are already removed from the tracker. If the coding conflict does not occur then the place is tagged as redundant. The traversing of the chain stops when a non-redundant place is found. After that the next chain is processed.

The redundant places *cmp2*, *pgt2*, *plt2* and *peq2* are detected by this heuristic in the GCD example. Place *cmp1* can also

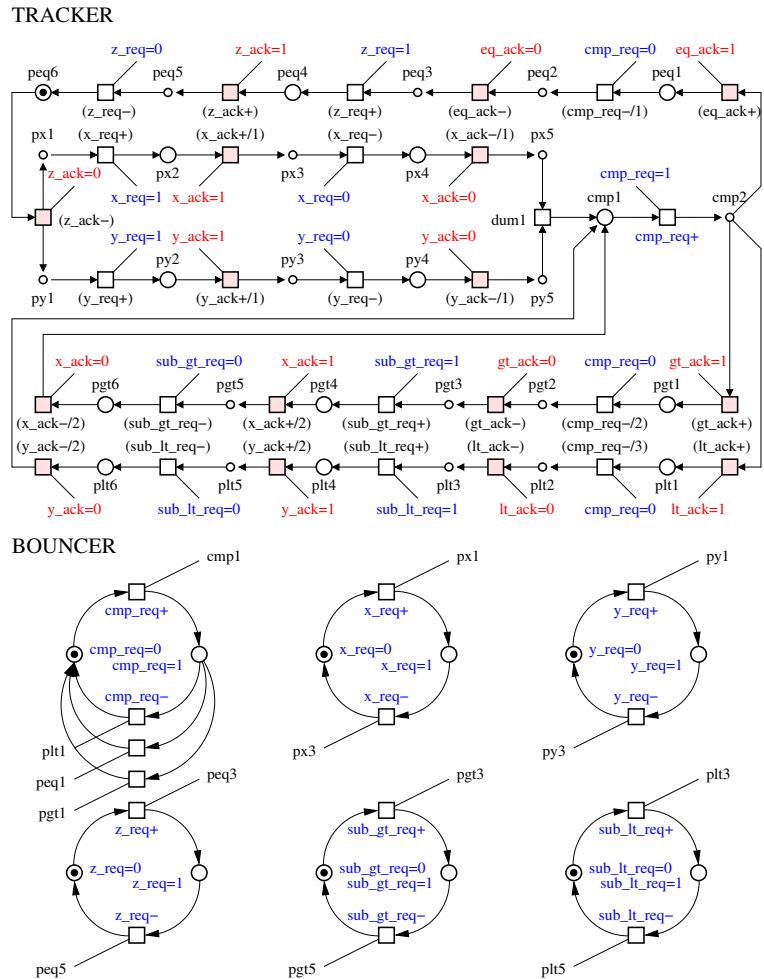
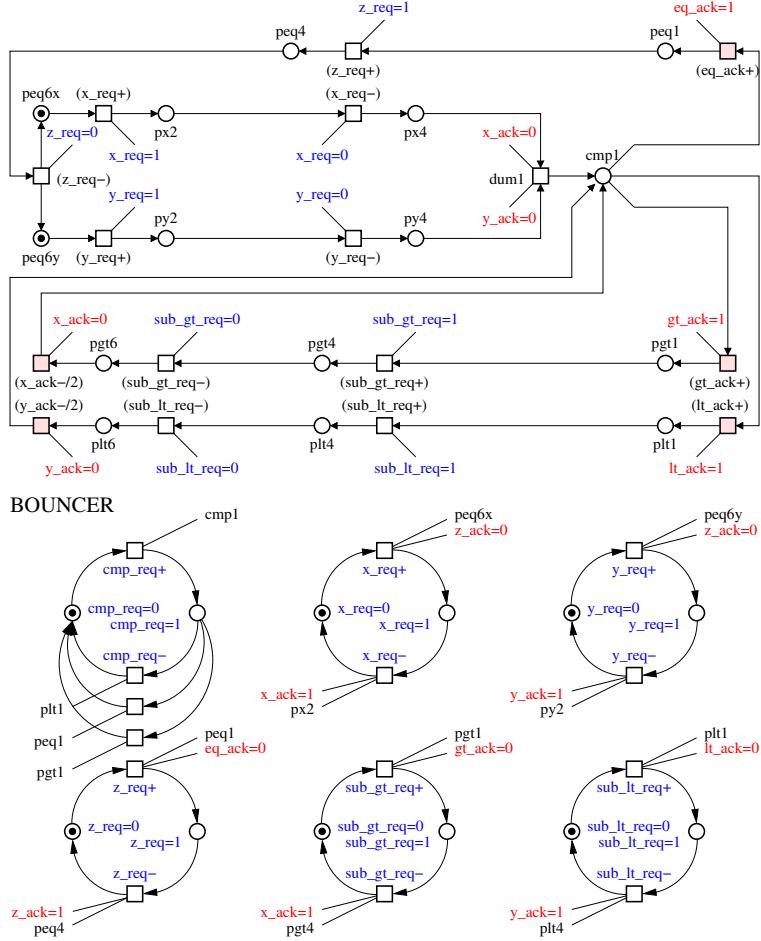


Figure 12. Exposure of the outputs and detection of the redundant places

TRACKER



BOUNCER

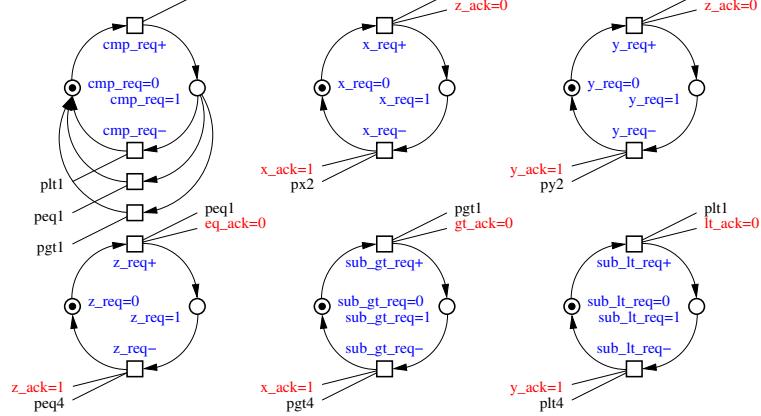


Figure 13. Optimisation

be tagged as redundant but it is kept by the OptiMist tool in order to preserve the simplicity of the *cmp_req* elementary cycle. Without this place the positive phase of the *cmp_req* would be controlled by two context signals from the tracker (read-arcs from *px4* and *py4*) and two trigger signals from the environment (*x_ack=0* and *y_ack=0*). The trade-off between the complexity of elementary cycles and the number of places in the tracker can be set as an OptiMist command line parameter. Removal of any of the other places from the tracker causes the coding conflict and such places should be preserved.

After the outputs are exposed and the redundant places are detected, the OptiMist tool optimises the tracker by removing the redundant places and preceding dummy transitions. The removal of a place involves the change of the STG structure but preserves the behaviour of the system w.r.t. input-output interface. The result of GCD control path optimisation is presented in Figure 13.

This STG can now be used for circuit synthesis. For this each tracker place is mapped into a DC and each elementary cycle is mapped into a FF. The request and acknowledgement functions of a DC are mapped from the structure of the tracker in the vicinity of the corresponding place, similar to the mapping from LPNs technique. The set and reset functions of a FF are mapped from the structure of the set and reset phases of the corresponding elementary cycle.

The GCD controller circuit obtained by this technique consist of 15 DCs and 6 FFs. The maximum number of transistor levels in pull-up and pull-down stacks is 4. This transistor stack appears in the request function of the DC for *cmp1* and formed by the signals *x_ack=0*, *y_ack=0*, *px4_req* and *py4_req*.

The longest latency, which is the delay between an input change and reaction of the controller by changing some outputs, is exhibited by *cmp_req* signal. The latency of its set and reset phases is equal to the delay of one DC and one FF. The other

outputs are triggered directly by input signals which means that their latencies are equal to one FF delay plus the delay of one inverter when the trigger signal requires inversion.

7. Explicit logic synthesis of control path

The *explicit logic synthesis* methods work with the low-level system specifications which capture the behaviour of the system at the level of signal transitions, such as STGs. These methods usually derive boolean equations for the output signals of the controller using the notion of *next state functions* obtained from STGs [7].

An STG is a succinct representation of the behaviour of an asynchronous control circuit that describes the causality relations among the events. In order to find the next state functions all possible firing orders of the events must be explored. Such an exploration may result in a state space which is much larger than the STG specification. Finding efficient representations of the state space is a crucial aspect in building synthesis tools.

The synthesis method based on state space exploration is implemented in the Petrify tool [8]. It represents the system state space in form of a *State Graph* (SG), which is a binary encoded reachability graph of the underlying PN. Then the theory of regions [9] is used to derive the boolean equations for the output signals.

The explicit representation of concurrency results in a huge SG for a highly concurrent STG. This is known as the *state space explosion* problem, which puts practical bounds on the size of control circuits that can be synthesised using state-based techniques.

The other interesting issue is the unambiguous state encoding. It appears when the binary encoding of the SG state alone cannot determine the future behaviour of the system. Hence, an ambiguity arises when trying to define the next-state function. Roughly speaking, this phenomenon appears when the system does not have enough memory to ‘remember’ in which state it is. When this occurs, the system is said to violate the *Complete State Coding* (CSC) property. Enforcing CSC is one of the most difficult problems in the synthesis of asynchronous circuits. The general idea of solving CSC conflicts is the insertion of new signals, that add more memory to the system. The signal events should be added in such a way that the values of inserted signals disambiguate the conflicting states.

7.1. Automatic CSC conflict resolution

One of the possibilities to resolve the CSC conflicts is to exploit the Petrify tool and the underlying theory of regions. In Petrify all calculations for finding the states in conflict and inserting the new signal events are performed at the level of SG. The tool relies on the set of optimisation heuristics when deciding how to insert new transitions. However, the calculation of regions involves the computationally intensive procedures which are repeated when every new signal is inserted. This may result in long computation time.

When the system becomes conflict-free, the SG may be transformed back into STG. Often the structure of resultant STG differs significantly from the original STG, which might be inconvenient for its further manual modification. Actually, the STG look may change even after simple transformation to SG and back to STG, because the structural information is lost at the level of SG.

The conflict-free STG for the GCD control path obtained by Petrify is shown in Figure 14. There are two changes to the structure of the STG which are not due to new signal insertion. Firstly, the transition *cmp_req+* is split into *cmp_req+/1* and *cmp_req+/2*; Secondly, the concurrent input of *x* and *y* is synchronised on *cmp_req+/1* instead of dummy now.

Petrify resolves the CSC conflicts in GCD control path specification adding five new signals, namely *csc0*, *csc1*, *csc2*, *csc3*, *csc4*. The insertion of signals *csc0*, *csc3* and *csc4* is quite predictable. They are inserted in three conflicting branches (one in each branch) in order to distinguish between the state just before *cmp_req+/1* and just after *eq_ack-*, *gt_ack-*, *lt_ack-* respectively.

For example, the state of the system before and after the following sequence of transitions is exactly the same: *cmp_req+/1* → *eq_ack+* → *cmp_req-/1* → *eq_ack-*. In order to distinguish between these states *csc0+* transition is inserted inside the above sequence. As the behaviour of the environment must be preserved, the new transition can only be inserted before the output transition *cmp_req-/1*. There are two possibilities for its insertion: sequentially or concurrently. The former type of insertion is usually (but not always) preferable for smaller size of the circuit, the latter for lower latency. Relying on its sophisticated heuristics Petrify decides to insert *csc0+* sequentially. Signal *csc0* is reset in the same branch outside the above sequence of transitions.

Similarly, signals *csc1* and *csc2* are inserted to distinguish the states before and after the sequence of transitions *x_req+→x_ack+→x_req→x_ack-* and *y_req+→y_ack+→y_req→y_ack-* respectively. However the reset of *csc2* is not

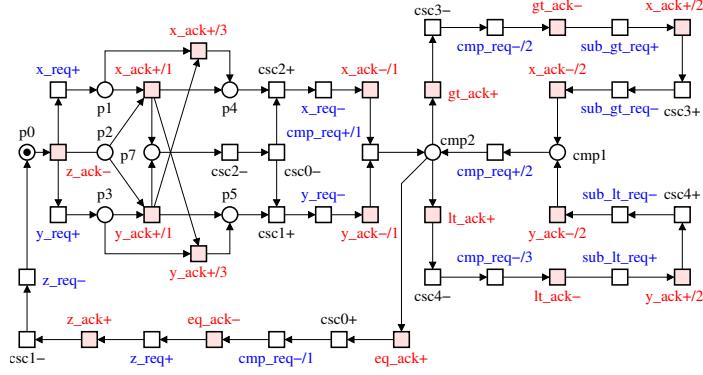


Figure 14. Resolution of CSC conflicts by Petrify

symmetrical to the reset of $csc1$ (as is expected) and involves a significant change of the original STG structure, see Figure 14.

The synthesis of the conflict-free specification with logic decomposition into gates with at most four literals results in the following equations:

```
[x_req] = z_ack' (csc0 csc1' + csc2');
[y_req] = z_ack' csc1;
[z_req] = csc0 eq_ack' csc1;
[3] = csc4' + csc3' + csc0 + csc2';
[cmp_req] = [3]' x_ack' y_ack' csc1;
[sub_gt_req] = csc3' gt_ack';
[sub_lt_req] = csc4' lt_ack';
[csc0] = csc2 csc0 + eq_ack;
[csc1] = csc0' y_ack + z_ack' csc1;
[9] = csc0' (csc2 + x_ack);
[csc2] = x_ack' csc2 y_ack' + [9];
[csc3] = gt_ack' (csc3 + x_ack);
[csc4] = lt_ack' (csc4 + y_ack);
```

The estimated area is 432 units and the maximum and average delay between the inputs is 4.00 and 1.75 events respectively. The worst case latency is between the input x_ack+/l and the output x_req- . The trace of the events is $x_ack+/l \rightarrow csc_2 \rightarrow csc_0 \rightarrow csc_2 \rightarrow x_req-$. Taking into account that CMOS logic is built out of negative gates these events correspond to the following sequence of gates switching: $[x_ack \uparrow] \rightarrow [x_ack \downarrow] \rightarrow [csc2 \uparrow] \rightarrow [csc2 \downarrow] \rightarrow [csc0 \uparrow] \rightarrow [9 \downarrow] \rightarrow [9 \uparrow] \rightarrow [csc2 \downarrow] \rightarrow [x_req \uparrow] \rightarrow [x_req \downarrow]$. This gives the latency estimate equal to the delay of 9 negative gates.

7.2. Semi-automatic CSC conflict resolution

A semi-automatic approach to CSC conflict resolution is adopted in the ConfRes tool [21]. The main advantage of the tool is its interactivity with the user during CSC conflict resolution. It visualises the cause of the conflicts and allows the designer manipulate the model by choosing where in the specification to insert new signals.

The ConfRes tool uses STG unfolding prefixes [18] to visualise the coding conflicts. An unfolding prefix of an STG is a finite acyclic net which implicitly represents all the reachable states of the STG together with transitions enabled at those states. Intuitively, it can be obtained by successive firings of STG transitions under the following assumptions: (a) for each new firing a fresh transition (called an *event*) is generated; (b) for each newly produced token a fresh place (called a *condition*) is generated. If the STG has finite number of reachable states then the unfolding eventually starts to repeat itself and can be truncated (by identifying a set of *cut-off events*) without loss of information, yielding a finite and complete prefix.

In order to avoid the explicit enumeration of coding conflicts, they are visualised as *cores*, i.e. the sets of transitions ‘causing’ one or more of conflicts. All such cores must eventually be eliminated by adding new signals.

Two experiments are conducted. In the first one the strategy of sequential signal insertion is exploited in order to compete automatic conflict resolution in circuit size. In the second experiment the new signals are inserted concurrently (where possible) in order to achieve lower latency. The resultant conflict-free STG of the GCD controller is shown in Figure 15.

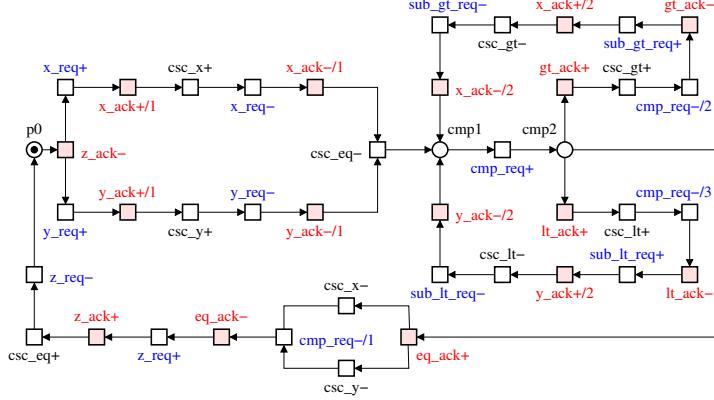


Figure 15. Resolution of CSC conflicts by ConfRes

Petrify synthesises this STG with logic decomposition into gates with at most four literals into the following equations:

```

[x_req] = csc_x' z_ack' csc_eq;
[y_req] = csc_y' z_ack' csc_eq;
[z_req] = csc_x' eq_ack' csc_eq';
[3] = csc_y' csc_x' + csc_gt + csc_lt;
[cmp_req] = [3]' y_ack' x_ack' csc_eq';
[sub_gt_req] = csc_gt gt_ack';
[sub_lt_req] = csc_lt lt_ack';
[csc_x] = eq_ack' (x_ack + csc_x);
[csc_y] = csc_y eq_ack' + y_ack;
[csc_gt] = x_ack' csc_gt + gt_ack;
[10] = csc_eq (csc_x' + csc_y') + z_ack;
[csc_eq] = csc_eq (x_ack + y_ack) + [10];
[csc_lt] = y_ack' (lt_ack + csc_lt);
    
```

The estimated area is 432 units, which is the same as when the coding conflicts are resolved automatically. However, the maximum and average delays between the inputs are significantly improved: 2.00 and 1.59 events respectively. The worst case latency of the circuit is between gt_ack+ and cmp_req-2 (or between eq_ack+ and cmp_req-1). If the circuit is implemented using CMOS negative gates then this latency corresponds to the following sequence of gates switching: $[gt_ack \uparrow] \rightarrow [csc_gt \downarrow] \rightarrow [csc_gt \uparrow] \rightarrow [3' \downarrow] \rightarrow [cmp_req' \uparrow] \rightarrow [cmp_req \downarrow]$. This gives the latency estimate equal to the delay of 5 negative gates, which is significantly better than in the experiment with automatic coding conflict resolution.

The other experiment with semi-automatic CSC conflict resolution aims at lower latency of the GCD control circuit. Now the new signal transitions are inserted as concurrently as possible. Namely, csc_x+ is concurrent to x_ack+1 ; csc_y+ is concurrent to y_ack+1 ; csc_gt- is concurrent to x_ack+2 ; and csc_lt- is concurrent to y_ack+2 . The other transitions are inserted the same way as in the previous experiment.

Two new signals, $map0$ and $map1$, are added by Petrify in order to decompose the logic into library gates with at most four literals. This results in larger estimated circuit size, 592 units. The average input-to-input delay of the circuit becomes 1.34 events, which is smaller than in the previous experiment. However, the maximum latency of the circuit is 7 negative gates delay. It occurs, for example, between gt_ack+ and cmp_req- transitions. The gates switched between these transitions together with the direction of switchings are: $[gt_ack \uparrow] \rightarrow [csc_gt \downarrow] \rightarrow [csc_gt \uparrow] \rightarrow [map0' \downarrow] \rightarrow [map0 \uparrow] \rightarrow [5' \downarrow] \rightarrow [cmp_req' \uparrow] \rightarrow [cmp_req \downarrow]$. The worst case latency in this implementation is greater than the latency in the previous design due to the internal $map0$ and $map1$ signals, which are used for decomposition of non-implementable functions.

The complex gate implementation of the GCD controller, where CSC conflict is resolved manually by inserting new signals in series is the best solution (in terms of size and latency) synthesised by Petrify with the help of the ConfRes tool. It consists of 120 transistors and exhibits the latency of 5 negative gates delay.

Clearly, semi-automatic conflict resolution gives the designer a lot of flexibility in choosing between the circuit size and latency. The visual representation of conflict cores distribution helps the designer to plan how to insert each phase of a new signal optimally, thus possibly destroying several cores by one signal. The diagram of core distribution is updated after every new signal insertion. As all the modifications to the system are performed on its unfolding prefix, there is no need to

Tool	Area (μm^2)	Speed (ns)		computation time (s)
		x=y	x=12, y=16	
Balsa	119,647	21	188	< 1
PN2DCs	100,489	14	109	< 1
Improvement	16%	33%	42%	0

Table 1. Comparison between Balsa and PN2DCs

Tool	number of transistors	latency (units)	computation time (s)
OptiMist	174	4.5	< 1
Petrify	automatic	116	13.0
	sequential	120	8.0
	concurrent	142	11.0

Table 2. Comparison between OptiMist and Petrify

recalculate the state space of the system, which makes the operation of ConfRes tool extremely fast.

Another approach to CSC conflicts resolution which avoids the expensive computation of the system state space is proposed in [4]. The approach is based on the structural methods, which makes it applicable for large STG specifications. Its main idea is to insert a new set of signals in the initial specification in a way that unique encoding is guaranteed in the transformed specification. The main drawback of this approach is that the structural methods are approximate and can only be exact for well-formed PNs.

8. Tools comparison

In this section the tools are compared using GCD benchmark in two categories: 1) system synthesis from high-level HDLs and 2) synthesis of the control path from STGs.

Table 1 presents characteristics of asynchronous GCD circuits synthesised by Balsa and PN2DCs tools from high-level HDLs. Both solutions are implemented using the AMS- $0.35\mu m$ technology and dual-rail datapath components. The size of each circuit is calculated using Cadence Ambit tool and the speed is obtained by circuit simulation in SPICE analog simulator.

The benchmark shows that the circuit generated by PN2DCs tool is 16% smaller and 33-42% faster than the circuit synthesised by Balsa. The size and speed improvement in PN2DCs comparing to Balsa solution is due to different control strategies. Note that the intermediate controller specification for the PN2DCs tool is manually optimised by removing redundant places and transitions. This reduces the control path area by four DCs ($732\mu m^2$). However, the optimisation algorithm is straightforward, the redundant places and transitions removal can be automated.

The time spent by Balsa and PN2DCs to generate the circuit netlists is negligible. This is because both tools use computationally simple mapping techniques, which allows to process large system specifications in acceptable time.

The characteristics of the circuits synthesised from STGs are shown in Table 2. The number of transistors for the circuits generated by Petrify is counted for complex gate implementation. The technology mapping into the library gates with at most four literals is applied.

In all experiments, the latency is counted as the accumulative delay of negative gates switched between an input and the next output. The following dependency of a negative gate delay on its complexity is used. The latency of an inverter is associated with 1 unit delay. Gates which have maximum two transistors in their transistor stacks are associated with 1.5 units; 3 transistors - 2.0 units; 4 transistors - 2.5 units. This approximate dependency is derived from the analysis of the gates in AMS $0.35\mu m$ library. The method of latency estimation does not claim to be very accurate. However, it takes into account not only the number of gates switched between an input and the next output, but also the complexity of these gates.

The Petrify tool was used to synthesise the circuits with three alternatives of CSC conflict resolution. In the first circuit the coding conflict is solved by inserting new signals automatically. In the second and the third circuits the semi-automatic method of conflict resolution is employed by using the ConfRes tool. In the second circuit the transitions of new signals are

inserted sequentially, and in the third one concurrently.

The experiments show that the automatic coding conflict resolution may result in a circuit with high output latency which is due to non-optimal insertion of the new signals. The smallest circuit is synthesised when the coding conflicts are resolved manually by inserting the new signals sequentially. This solution also exhibits lower latency than in the case of automatic and concurrent signal insertion. The circuit with the new signals inserted concurrently lacks the expected law latency because of its excessive logic complexity.

The circuit with the lowest latency is generated by the direct mapping technique using the OptiMist tool. This tool also exhibits the smallest synthesis time which is due to low algorithmic complexity of the involved computations. This allows processing large specifications, which cannot be computed by Petrify in acceptable time because of the state space explosion problem.

This can be illustrated on the scalable benchmark with 3, 4, 5 and 6 concurrent branches. When the concurrency increases the Petrify computation time grows exponentially (0.2s, 1m15s, 7m52s and 33m12s respectively), while the OptiMist computation time grows linearly on the same benchmark (0.09s, 0.12s, 0.14s and 0.16s respectively). However, the GCD controller synthesised by the OptiMist tool is about 45% larger than the Petrify's solutions.

9. Conclusions

The state of the art in the synthesis of asynchronous systems from high-level behavioural specifications has been reviewed. Two main approaches of circuit synthesis have been considered: syntax-driven translation and logic synthesis.

The syntax-driven approach is studied on the example of Balsa design flow. It uses a CSP-based language for the initial system specification. Its parsing tree is translated into a handshake circuit, which is subsequently mapped to the library of hardware components. This approach enables the construction of large-size asynchronous systems in a short time, due to its low computational complexity. However, the speed and area of the circuit implementations may not be the best possible. Therefore this approach benefits from peep-hole optimisations, which apply logic synthesis locally, to groups of components, as was demonstrated in [5].

The logic synthesis approach is reviewed using the PN2DCs, OptiMist, Petrify and ConfRes tools. The PN2DCs tool partitions the VHDL system specification on control and datapath. Petri nets are used for their intermediate behavioural representation. The datapath PN is subsequently mapped into a netlist of datapath components. The controller PN can be either mapped into a David cell structure or further refined to an STG. The control path can be synthesised by one of the above mentioned tools. Logic synthesis approach is computationally harder than the syntax-based one. The direct mapping of Petri nets and STGs in PN2DCs and OptiMist, helps to avoid state space explosion involved in the state encoding procedures used in Petrify. At the same time, this comes at the cost of more area.

It should be clear that tools like Petrify and ConfRes should be used for relatively small control logic, for instance in interfaces and pipeline stage controllers (see [10]), rather than complex data processing controllers, where PN2DCs is more appropriate. The latter is however not optimal for speed because it works at a relatively high-level of signalling. OptiMist, working at the STG level, combines the advantages of low computational complexity with high-speed due to its latency-aware implementation architecture with a bouncer and a tracker.

The greatest common divisor benchmark is used to evaluate all of the above mentioned tools. The size and speed of the resultant circuits are compared. They demonstrate the various possible enhancements in the design flow, such as the use of an interactive approach to state encoding in logic synthesis.

In the future a combination of techniques in a single tool flow might prove most advantageous. For example, at first, each output signal which has a complete state coding can be synthesised individually by Petrify, with or without use of ConfRes. Then, all the remaining outputs, whose CSC resolution is hard or impossible, can be mapped into logic at once by OptiMist. Here, the best trade-off between area and performance may be achieved.

Acknowledgements

The authors are grateful to Agnes Madalinski, Delong Shang, Julian Murphy and Alex Bystrov for their useful comments. EPSRC supports this work via GR/R16754 (BESST), GR/S12036 (STELLA) and GR/S81421 (SCREEN).

References

- [1] Andrew Bardsley and Doug Edwards. The Balsa asynchronous circuit synthesis system. In *Forum on Design Languages*, September 2000.

- [2] Ivan Blunno and Luciano Lavagno. Automated synthesis of micro-pipelines from behavioral Verilog HDL. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, pages 84–92. IEEE Computer Society Press, April 2000.
- [3] Alex Bystrov and Alex Yakovlev. Asynchronous circuit synthesis by direct mapping: Interfacing to environment. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, pages 127–136, Manchester, UK, April 2002. IEEE Computer Society Press.
- [4] Josep Carmona. *Structural methods for the synthesis of well-formed concurrent specifications*. PhD thesis, Software Department, Universitat Politècnica de Catalunya, March 2004.
- [5] Tiberiu Chelcea, Andrew Bardsley, Doug Edwards, and Steven M. Nowick. A burst-mode oriented back-end for the Balsa synthesis system. In *Proc. Design, Automation and Test in Europe (DATE)*, pages 330–337, March 2002.
- [6] Wei-Chun Chou, Peter A. Beerel, and Kenneth Y. Yun. Average-case technology mapping of asynchronous burst-mode circuits. *IEEE Transactions on Computer-Aided Design*, 18(10):1418–1434, October 1999.
- [7] Tam-Anh Chu. *Synthesis of self-timed VLSI circuits from graph-theoretic specifications*. PhD thesis, Jun, Benjamin 1987.
- [8] Jordi Cortadella, Michael Kishinevsky, Alex Kondratyev, Luciano Lavagno, and Alex Yakovlev. Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. In *XI Conference on Design of Integrated Circuits and Systems*, Barcelona, Spain, November 1996.
- [9] Jordi Cortadella, Michael Kishinevsky, Alex Kondratyev, Luciano Lavagno, and Alex Yakovlev. A region-based theory for state assignment in speed-independent circuits. *IEEE Transactions on Computer-Aided Design*, 16(8):793–812, August 1997.
- [10] Jordi Cortadella, Michael Kishinevsky, Alex Kondratyev, Luciano Lavagno, and Alex Yakovlev. *Logic synthesis for asynchronous controllers and interfaces*. Springer-Verlag, Berlin, Germany, March 2002.
- [11] A. Dindhuc, J.-B. Rigaud, A. Rezzag, A. Sirianni, J. L. Fragoso, L. Fesquet, and M. Renaudin. Tima asynchronous synthesis tools. In *Communication to ACID Worshop*, January 2002.
- [12] R. M. Fuhrer, Steven M. Nowick, Michael Theobald, N. K. Jha, B. Lin, and Luis Plana. Minimalist: an environment for the synthesis, verification and testability of burst-mode asynchronous machines. Technical Report TR CUCS-020-99, Columbia University, July 1999.
- [13] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
- [14] Lee A. Hollaar. Direct implementation of asynchronous control units. *IEEE Transactions on Computers*, C-31(12):1133–1141, December 1982.
- [15] D. A. Huffman. The synthesis of sequential switching circuits. In E. F. Moore, editor, *Sequential Machines: Selected Papers*. Addison-Wesley, 1964. Reprinted from J.Franklin Institute, vol. 257, no. 3, pp. 161–190, Mar. 1954, and no. 4, pp. 275–303, Apr. 1954.
- [16] Axel Jantsch. *Modelling Embedded Systems and SoCs: Concurrency and Time in Models of Computation*. Morgan Kaufmann, 2004.
- [17] K. Jensen. *Coloured Petri nets: basic concepts, analysis methods and practical use*, volume 1. Springer-Verlag, 1997.
- [18] Victor Khomenko. *Model checking based on Petri net unfolding prefixes*. PhD thesis, School of Computer Science, University of Newcastle upon Tyne, 2002.
- [19] Michael Kishinevsky, Alex Kondratyev, Alexander Taubin, and Victor Varshavsky. *Concurrent hardware: the theory and practice of self-timed design*. Series in Parallel Computing. Wiley-Interscience, John Wiley& Sons, Inc., 1994.
- [20] Alex Kondratyev and Kelvin Lwin. Design of asynchronous circuits using synchronous CAD tools. *IEEE Design & Test of Computers*, 19(4):107–117, 2002.

- [21] Agnes Madalinski, Alex Bystrov, Victor Khomenko, and Alex Yakovlev. Visualization and resolution of coding conflicts in asynchronous circuit design. In *Proc. Design, Automation and Test in Europe (DATE)*. IEEE Computer Society Press, March 2003.
- [22] U. Montanari and F. Rossi. Acta informacia. Technical report, 1995.
- [23] Gordon Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):68–70, April 1965.
- [24] Ad Peeters. Support for interface design in Tangram. In Alex Yakovlev and Reinder Nouta, editors, *Asynchronous Interfaces: Tools, Techniques, and Implementations*, pages 57–64, July 2000.
- [25] Leonid Rosenblum and Alex Yakovlev. Signal graphs: from self-timed to timed ones. In *Proceedings of International Workshop on Timed Petri Nets*, pages 199–207, Torino, Italy, July 1985. IEEE Computer Society Press.
- [26] M. Sacker, A. Brown, , and A. Rushton. A general purpose behavioural asynchronous synthesis system. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, pages 125–134. IEEE Computer Society Press, April 2004.
- [27] Delong Shang, Frank Burns, Albert Koelmans, Alex Yakovlev, and F. Xia. Asynchronous system synthesis based on direct mapping using VHDL and Petri nets. *IEE Proceedings, Computers and Digital Techniques*, 151(3):209–220, May 2004.
- [28] Alexander Smirnov, Alexander Taubin, , and Leonid Rosenblum. Gate transfer level synthesis as an automated approach to fine-grain pipelining. In *Proc. International Conference on Application and Theory of Petri Nets*, Jun, Benjamin 2004.
- [29] Douglas Smith. VHDL and Verilog compared and contrasted plus modeled example written in VHDL, Verilog and C. In *Proc. ACM/IEEE Design Automation Conference*, pages 771–776, Jun, Benjamin 1996.
- [30] Danil Sokolov, Alex Bystrov, and Alex Yakovlev. STG optimisation in the direct mapping of asynchronous circuits. In *Proc. Design, Automation and Test in Europe (DATE)*, Munich, Germany, March 2003. IEEE Computer Society Press.
- [31] Danil Sokolov, Julian Murphy, Alex Bystrov, and Alex Yakovlev. Design and analysis of dual-rail circuits for security applications. *IEEE Transactions on Computers*, 54(4):449–460, April 2005.
- [32] Ivan Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, Jun, Benjamin 1989. The 1988 Turing award lecture.
- [33] S. H. Unger. *Asynchronous sequential switching circuits*. Wiley-Interscience, John Wiley& Sons, Inc., New York, USA, 1969.
- [34] Girish Venkataramani, Mihai Budiu, Tiberiu Chelcea, and Seth Copen Goldstein. C to asynchronous dataflow circuits: an end-to-end toolflow. In *International Workshop on Logic Synthesis*, Jun, Benjamin 2004.
- [35] Walter Vogler and Ralf Wollowski. Decomposition in asynchronous circuit design. In Jordi Cortadella, Alex Yakovlev, and G. Rozenberg, editors, *Concurrency and Hardware Design*, volume 2549 of *Lecture Notes in Computer Science*, pages 152–190. Springer-Verlag, 2002.

The Use Of Scenarios To Improve The Flexibility In Embedded Systems: Research Challenges And Preliminary Results

I. Bate and P. Emberson
Department of Computer Science
University of York
York, United Kingdom
e-mail: {paul.emberson,iain.bate}@cs.york.ac.uk

Abstract

Flexibility in real-time, embedded systems is important. As in any type of system, changes arise from maintenance, enhancements and upgrades. These changes are only feasible if timing requirements imposed by the real-time nature of the system can still be met. A flexible design will allow tasks to be added without impinging on other tasks, causing them to miss deadlines. The design space for these systems consists of many configurations describing how tasks and messages are allocated to hardware and scheduled on a hardware platform. The changeability of a system can be considered as a quality attribute and enhanced using scenario based analysis and architecture trade off methods. Heuristic search is a well recognised strategy for solving allocation and scheduling problems but most research is limited to finding any valid solution for a current set of requirements. The technique proposed here incorporates scenario based analysis into heuristic search strategies where the ability of a solution to meet a scenario is included as another heuristic for the changeability of a system. This allows future requirements to be taken into account when choosing a solution so that future changes can be accommodated with minimal alterations to the existing system.

1 Introduction

Designing to cope with change has become increasingly important as complexity increases. Changes to a poorly designed system are likely to lead to ripple effects where the cost of the change is disproportionate to its size and may even outweigh the benefits received from the change. Ideally both the size and cost of change should be comparable to the resulting effect. An extreme example of this can be found in the avionics domain. The huge cost of developing new aircraft platforms means that systems are likely to be in use for decades, requiring several upgrades. Any changes must be re-certified adding to the importance of keeping the number of changes to perform an upgrade minimal. There are similar circumstances in other industries, such as automotive, where changes to independently developed modules shouldn't ripple through the entire system.

A major difficulty in designing for change is being able to predict the changes that will be required. It is common for large systems to use a phased development program. In this instance, future phases should be carefully planned with documented future requirements. Alternatively, a possible enhancement may arise during development that is postponed for a future, larger upgrade. Both of these situations suggest anticipated changes. Unanticipated changes can arise from problems found during testing or from new requirements submitted by a customer. In reality, low level changes for an anticipated future requirement, such as increases to task execution times, cannot be fully known without actually implementing the change. In this sense changes are rarely 100% anticipated but rather estimated to within some degree of correctness.

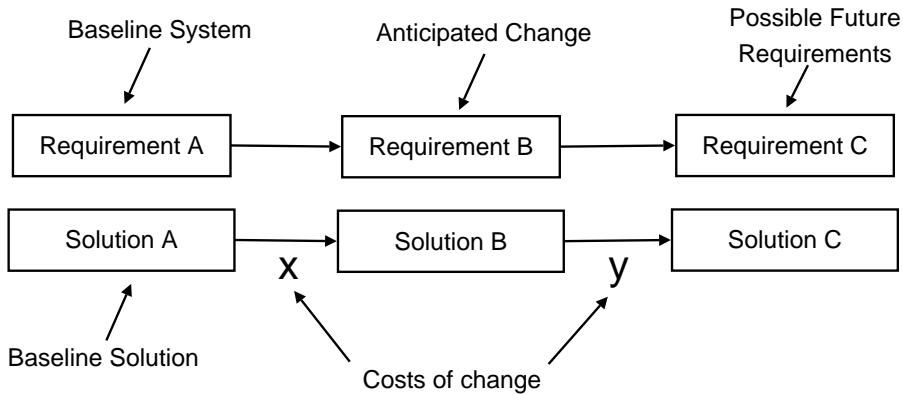


Figure 1: The cost of meeting future requirements, y , is dependent upon the solution chosen to meet immediate changes. Minimising this cost, x , for imminent changes may not minimise the total cost of change, $x + y$, over the lifespan of the project.

The method outlined in this paper uses anticipated change scenarios to select a design for a system under development that will allow the eventual upgrade described in the scenario to be realised with minimal changes to the design. In addition, there is an investigation of how using scenarios for anticipated changes increases system flexibility for unanticipated changes. The chosen design for the current system requirements is referred to as the baseline design, shown as the baseline solution in figure 1. In addition to creating a flexible baseline design, a simple metric is suggested for measuring the cost of an upgrade to meet new requirements. This enables us to select an upgrade that requires fewest and cheapest changes. A longer term aim is to consider how minimising the cost of an upgrade trades off with the flexibility of the new baseline being created. In a system being developed over several phases, it is important to consider all phases not just the next one as proposed in figure 1. The further into the future a change is due to occur, the fuzzier and less certain the requirements become and hence these scenarios should be given less weight.

To show how scenarios may be used in embedded systems a task allocation example is used. These systems will contain several tasks distributed between a number of processors communicating via messages on a network. A particular *configuration* of such a system describes the way tasks and messages are allocated and scheduled within the system. Separating software application development from the selection of a configuration can greatly ease upgrades as the

configuration can be changed post initial development. In the avionics domain, Integrated Modular Avionics (IMA) provides a standardised interface between software and hardware [2, 10]. Our work is motivated by finding a method to configure an IMA system that will produce flexible solutions with respect to future upgrades. A valid allocation will be required to meet non-functional constraints such as timing deadlines and having tasks allocated to a processor appropriate to its criticality. As far as future changes are concerned, the baseline configuration should be designed so that changes (e.g. adding additional tasks or changing existing worst-case execution times) can be handled without having to alter the configuration of the existing tasks.

In previous work [5], we presented a heuristic search based environment for selecting a configuration. Each configuration is evaluated using a function to measure the quality attributes of the system under that configuration. This paper integrates scenario based assessment into that environment to produce more flexible baselines. An additional component is introduced to the quality evaluation function which is able to measure the cost when a change has to be made. Adding this component allows the cost of upgrades to be minimised.

The structure of the paper is as follows. The next section describes the task allocation problem in further detail and is followed by section 3 describing the research challenges involved in creating flexible real time distributed systems. The paper continues in section 4 with an overview of scenarios and scenario based assessment. Section 5 outlines the subject of heuristic searching and some details of how it is used in our tool. Following this, section 6 explains how the two subjects are integrated to perform configuration selection. The technique is assessed in section 7 by attempting some upgrades on baselines to check whether the use of scenarios reduces the cost of change. Finally the potential future work and conclusions are presented.

2 The Distributed Task Allocation Problem

Our previous work defined a trade-off analysis process [6] to define the objectives and design options for selecting an architecture with initial results for the task allocation problem presented in [4]. In this section, the objectives for a distributed real time embedded systems architecture are presented along with a description of the design choices available.

It is necessary to outline the objectives for the optimisation framework so that appropriate verification techniques can be derived and the corresponding results fed into the cost function which is used to judge the design. The primary objective is to meet the timing requirements of the system. This can be split into two categories. Those applicable to individual tasks and those applicable to multiple tasks. The requirements for individual tasks are period, deadline and completion jitter (completion jitter is the allowed variation in when a task might complete). The requirements for multiple tasks are:

- *transaction deadline* is the time by when an ordered sequence of tasks, known as a transaction, must complete.
- *transaction completion jitter* is the allowed variation in completion time of the transaction

- *separation* is a requirement for the minimum interval from when a task completes to the time another task starts executing for a particular pair of tasks.

Any tasks which communicate in a transaction communicate with a message. If the tasks are allocated to the same processor, this communication time will be negligible compared to if communicating tasks are distributed across different processors. However, sometimes it is necessary to force communicating tasks on to different processors. This occurs when either the required resources for a transaction are too high for a single processor or a task is not suited to running on a particular type of processor for other reasons, e.g. fault tolerance. So far our work supports transactions and transaction deadlines. Problems with completion jitter have been considered previously [4] but are not covered as yet in this body of work.

As previously stated, the objectives of the work are also to better support scalability and flexibility. At a lower-level this means that the design can cope with both changes and failures, although in this paper only change is handled.

The changes are of two types:

1. Changes to the execution profile of tasks. In this paper, the execution profile is described using Worst Case Execution Time (WCET).
2. Addition of new tasks and messages to the system. It should be noted the removal of tasks is not considered.

Therefore, the cost function should have components that assess how the design copes with changes (scenario-based assessment) and also determines the limits of changes that can be made before the timing requirements are no longer met (sensitivity analysis). Sensitivity analysis is applied to all individual tasks and task sets on each processor. Scenario-based assessment is used to fully explore part of the system which is expected to change (e.g. the effect of increasing the WCET for a set of tasks not on the same processor).

A final objective of the work is that the number of processors used should be minimised. Clearly solutions can be derived where a large number of processors are used and hence the system is able to cope with a great deal of change and failures. However, from a cost perspective this is not desirable. Hence an objective should be to keep the number of processors to a minimum.

The design choices considered in this paper are:

1. Ordering - this can mean priority in priority scheduling schemes or slot position in static scheduling
2. Allocation - the processor on which a task executes or communications bus for a message

Included within the allocation choice is the ability not to allocate any tasks to a particular processor, hence reducing the number of processors used.

3 Research Challenges

It is widely acknowledged that obtaining a set of consistent and stable requirements is difficult [18], especially on the first iteration of the development process. Using and relying on future requirements and properties of the system

may seem optimistic or even damaging as they may be incorrectly predicted. However, it may be easier to predict non-functional requirements and properties than functional ones. For example, there may be any number of ideas regarding the additional functionality for a particular subsystem. In contrast, it may be possible to estimate a bound for the number of additional tasks and spare execution time required to the necessary changes. Whether this can be achieved can be assessed using sensitivity [7] and scenario analysis. A key benefit of the approach is that irrespective of how accurate the predictions are, the baseline requirements and properties will still be met so nothing is lost over not performing any change analysis. However it should be recognised that indiscriminate use of scenarios is likely to be ineffective, and the wide spread use of scenarios may lead to tractability and scalability concerns.

Using scenarios allows design to have flexibility built into the appropriate parts of the system. A problem arises when a subsystem that was predicted to be relatively stable requires a large change. Without infinite resource, scalability must be traded off between subsystems hence a subsystem incorrectly determined to be stable may be difficult to change. This leads to the general problem of how to generate realistic scenarios for a system and how they should be prioritised [14]. For the method to be successful, some effort needs to be put into considering future requirements. This additional cost has greater benefit as the systems being considered become larger or more expensive to change, e.g. avionics. Often these types of systems are considered so difficult to change that anything other than major Mid-Life Updates (MLU) are prohibited [10].

To be able to apply an automated design process, the architecture must be modelled in such a way that allows it to be automatically modified. Our work has concentrated on varying the allocation of tasks to processors and the changing of scheduling parameters, but it also allows variations in hardware in term of the number of processors used. An architecture model must be accurate enough for it to be representative of the system but simple enough for it to be calculated efficiently. For example, there are different computational model for scheduling tasks which generally trade off the amount of pessimism in the model with calculation time. The model chosen must be accurate enough to be useful but if a large search space is to be covered in a tolerable amount of time a reasonable amount of pessimism has to be allowed.

There are challenges involved in calculating quantitative values to assess some qualities such as the changeability or dependability of an architecture. These are qualities which are hard to accurately assess without building, running and maintaining a system. Values assigned to these properties are subjective and the methods for measuring them need to be adaptable to different architectures and system requirements. This is among the issues that needs to be addressed to be able to generalise the method over different architectural styles, different quality attributes (e.g. power consumption) and different systems design problems.

In summary, the research challenges are:

1. Defining suitable scenarios for examining the impact of potential changes.
2. Prioritising the use of scenarios.
3. Creating a framework based on automated search techniques to solve the allocation problem that makes effective use of scenarios.

4. Showing the benefit of the approach to a wide variety of problem domains.
5. Dealing with scalability issues.
6. Broadening the approach to other non-functional properties such as power.
7. Developing analysis that trades off fidelity against computational complexity.

4 Scenarios

A scenario is a textual description of a possible requirement for a system. They may be qualitative as in “how secure is the system against a particular type of attack” or quantitative as in “how much will response times increase if the number of expected users increases by a factor of 10”. Applying a number of possible scenarios to a system architecture in the early stages of development can help to elicit new lower level requirements. A scenario might be that the layout of a Head Up Display (HUD) is to move from being fixed to being mission dependent. This may introduce a layout component into the architecture and hence a placement algorithm will need to be introduced into the lower level design.

For this to be most useful it is necessary to quantify the non-functional attributes from an architectural model. This is easier for some qualities, such as timing and memory usage, than for more subjective ones such as security and modifiability. Scenarios can also be used to consider future changes after the current system has been developed.

One of the first techniques proposed which used scenarios to assess the flexibility of an architecture was the Scenario Architecture Analysis Method (SAAM)[13]. This technique applies a scenario to a model of an architecture and marks the subsystems that would be required to change to meet the scenario. After a number of scenarios have been applied, the sensitivity of a subsystem is said to correspond to the number of scenarios assigned to it. This information can be used to decide if any components of the architecture need to be made more scalable or if the architecture should be changed. It is not possible to make a whole system scalable but instead to trade off scalability and other quality attributes between subsystems of an architecture. SAAM suggests scenarios as one way of doing this though it does not consider trading off qualities other than modifiability. The Architecture Tradeoff Analysis Method (ATAM)[14] covers a much wider range of quality attributes. It is more of an iterative approach than SAAM allowing architectures to be reworked and re-evaluated.

Much of the difficulty with any scenario based analysis technique is generating the scenarios. Different stake-holders such as customers, managers and developers will have different perspectives on the system. There may be organisational difficulties in eliciting information from all parties at once and the information may conflict. It follows that a scenario will never be precise and therefore any scenario based method should have tolerance to inaccuracies in the scenarios.

Our work looks at how knowledge contained in scenarios can be used in an automated environment for producing allocation and scheduling configurations to cope with change. Consideration is given to the inexact nature of scenarios

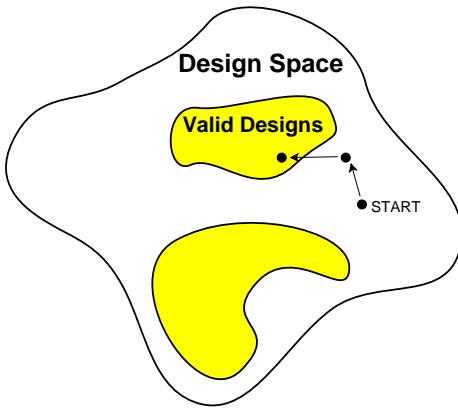


Figure 2: The search is guided towards valid solutions by the quality evaluation function

and whether using incorrect scenarios still aids modifiability or whether it may in fact have a negative impact on the cost of an upgrade.

5 Heuristic Searching

A heuristic search is guided through a search space by a function which measures the quality of a solution. The search takes steps to optimise the quality of the system and usually rejects steps which worsen it. An element of randomness allows the search to occasionally move via solutions of worse quality to avoid local optima. The technique is often targeted at problems for which the search space is NP large such as allocation and scheduling problems [11, 3, 9, 1, 17]. The uniqueness of the work presented here is that all other approaches have mainly tackled the allocation problem for the baseline design. That is, it has not addressed the significant problem of tolerating change and attempting to minimise the alterations when change is required.

The section is split into four parts. Firstly, the heuristic search algorithm is explained. Secondly the cost function for evaluating baseline designs is presented. Next, a component for measuring change when moving from a baseline to new requirements is described. Finally all the components are combined into a single cost function.

5.1 The Heuristic Search Algorithm

There are many heuristic search strategies each with their own variations [8]. The tool referred to in this paper and used in the evaluation uses simulated annealing [15]. The reason for choosing simulated annealing is based on it previously having been demonstrated as effective for allocation style problems [16]. Future work will explore whether other approaches are more efficient or effective as adding scenarios significantly affects the nature of the problem.

Simulated annealing starts at some point in the design space and uses an evaluation function to direct itself towards valid solutions as depicted in figure 2. The starting point may be chosen at random or in the case of searching for

a configuration to perform an upgrade may start at the baseline configuration. The algorithm for simulated annealing in relation to finding a configuration is written below.

```

t = initial_temperature
current_config = random starting configuration
current_cost = cost(current_config)
best_config = current_config
best_cost = current_cost
loop
    num_moves = 0
    loop
        increment num_moves
        new_config = modify(current_config)
        new_cost = cost(new_config)
        if (new_cost < current_cost)
            current_config = new_config
            current_cost = new_cost
            if (new_cost < best_cost)
                best_config = new_config
                best_cost = new_cost
                moves_since_improvement = 0
        else
            increment moves_since_improvement
        endif
    else
        prob = random probability
        d = new_cost - current_cost
        if ( prob < exp(-d/t) )
            current_config = new_config
            current_cost = new_cost
        endif
        increment moves_since_improvement
    endif
    while (num_moves < M) and
        (moves_since_improvement < N)
        t = t * cooling_factor
    while (moves_since_improvement < N)

```

The algorithm tries to minimise a cost function which assesses the quality of solutions. The temperature parameter governs the likelihood of accepting a higher cost solution and is reduced as the search progresses. The temperature is decreased by multiplying it by a cooling factor every M moves. The initial temperature value will be related to the range of the cost function. For cost values in the range [0, 1], we have found an initial temperature of 0.025 to be suitable. Typically the cooling factor would be set to 0.98 and reduced every 1000 moves.

All heuristic search algorithms have a number of parameters which affect how they move around the search space. For example, the ability to move to a new area via a region of lower quality of solutions, which is governed by the temperature in simulated annealing. This is coupled with the problem of deciding how solutions are evaluated for quality. For these reasons, even though similar problems have been solved with heuristic search techniques, each new application requires an assessment of how to optimise the search to the problem. In particular, there are difficulties with evaluating qualitative non-functional requirements such as safety, security, dependability and changeability. This

work considers how to deal with the latter. In previous work [5] the primary measurement of changeability was task sensitivity. Task sensitivity is where specific task(s) have their execution times extended until the system is no longer schedulable. Without specifying which tasks need to change, only a limited number of uniformly distributed task sets can be selected, e.g. group tasks by processor. The weakness of this approach is that there are a limited number of combinations that can be assessed as part of a scalable search strategy.

Task sensitivity is detailed in the following sub-sections with other measurements used for measuring the quality of a solution. A number of individual cost components are calculated which are then combined using a weighted mean to produce the overall cost value.

5.2 Cost Function Components for the Baseline System

The equations for each cost component are described below. Each component is normalised to the range [0, 1]. This eases the setting of the weightings of each cost component when they are combined into an overall cost function. That is, ensures that if the weightings of two components are equal then their importance in the overall cost function is also equal.

There are three broad classes of equations. There are components which ensure a solution is valid, components which try to give the solution desirable properties and finally components which help guide the search and improve search performance.

The first component gives a cost for unschedulable tasks.

$$f_1 = \frac{\text{num unschedulable tasks}}{\text{num tasks}} \quad (1)$$

It counts the number of unschedulable tasks and divides by the number of tasks in the system to produce a value between 0 and 1.

The second component is identical to equation (1) but for messages instead of tasks.

$$f_2 = \frac{\text{num unschedulable messages}}{\text{num messages}} \quad (2)$$

The next component calculates a cost for the number of unschedulable processors.

$$f_3 = \frac{\text{num unschedulable processors}}{\text{num processors}} \quad (3)$$

A processor is unschedulable if any of the tasks on it are unschedulable. This component has the effect of giving a bonus value to the cost when a processor becomes schedulable.

The following is the same as equation (3) but for networks.

$$f_4 = \frac{\text{num unschedulable networks}}{\text{num networks}} \quad (4)$$

To emphasise the benefit of the whole system being schedulable over, for example, a single task being unschedulable, the following equation is used.

$$f_5 = \text{num unschedulable systems} \quad (5)$$

This component has a value of 1 if any task or message is unschedulable and 0 otherwise.

Below is the component used to restrict the framework from using an arbitrarily large number of processors to achieve a scalable system.

$$f_6 = \frac{\text{num processors used}}{\text{num processors}} \quad (6)$$

The above equation calculates the proportion of available processors used. This component is given a low weighting in the evaluations presented in this paper since the emphasis is on scalability.

Equation (7) is used to calculate a cost for task sensitivity.

$$f_7 = \frac{\sum_{i \in \text{TASKS}} \text{SENS}_i}{\text{num TASKS}} \quad (7)$$

where TASKS is the set of all system tasks and SENS_i is the sensitivity of task i calculated using

$$\text{SENS}_i = e^{-\lambda \text{SCAL}_i} \quad (8)$$

where C_i is the worst case execution time of task i , and SCAL_i is the largest value of a factor $s \in \mathbb{R}$ such that the system is schedulable when the worst case execution time of task i is set to $\lfloor sC_i \rfloor$. The parameter λ can be used to vary the range of scalability values over which the value of the cost component varies most.

A sensitivity rating is also given to each processor as a whole as well as individual tasks.

$$f_8 = \frac{\sum_j \text{PROC_SENS}_j}{\text{num processors}} \quad (9)$$

This component is similar to the one in equation (7) but considers the scalability of all tasks on a processor.

$$\text{PROC_SENS}_j = e^{-\lambda \text{PROC_SCAL}_j} \quad (10)$$

where PROC_SCAL_j is the largest value of a factor $s \in \mathbb{R}$ such that the system is schedulable when the worst case execution time of all tasks on processor j is multiplied by s . The parameter λ can be used to vary the range of scalability values over which the value of the cost component varies most.

Note that it is valid for either of SCAL_i in equation (8) or PROC_SCAL_j in equation (10) to be less than 1 which indicates the system is currently unschedulable. This is used to help guide the search towards a schedulable solution as well as to make the final solution more scalable. In some cases, reducing the execution time of a single task or even all tasks on a single processor is not sufficient to make the system schedulable. In these cases the sensitivity for the corresponding task or processor is set to 1.

Some of the possible allocations are likely to be invalid if tasks are communicating. For example, a task needs to receive a message but that been allocated to a network not connected to the processor the task is allocated to. Instead of stopping these from occurring they are heavily penalised using the equation below. If these allocations were removed from the search space it would become more disjointed and there is a possibility that some good regions of the space may be harder to reach.

$$f_9 = \frac{\text{num incorrect allocations}}{\text{num allocations}} \quad (11)$$

where the number of allocations is the sum of the number of tasks and number of messages.

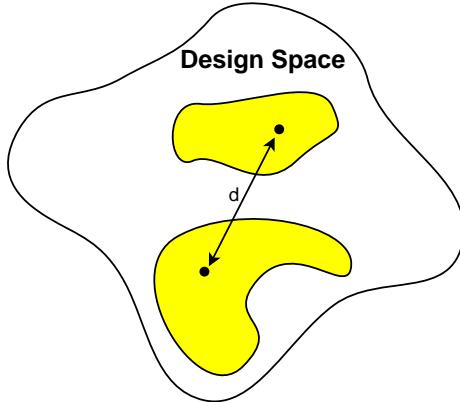


Figure 3: Cost of change as a distance measure

5.3 Measuring the Cost of Change to a System

In order to be able to minimise the cost of change when performing an upgrade, it is necessary to derive a metric for it. The chosen method to do this is to consider the difference between the current candidate configuration to fulfill the upgrade requirements and the baseline configuration. The configuration of a schedulable object, a task or message, is defined by an allocation to a processor and a scheduling priority. It is possible to count the number of allocation differences and number of priority ordering differences for all schedulable objects common to both configurations. Since only changes to the baseline configuration are of interest, new objects in the new configuration can be ignored. In the case of priority changes, only ordering rather than absolute values are considered. This allows any new tasks to be inserted at priority levels between existing tasks without a cost as the priority ordering of existing tasks hasn't changed. In equation (12), tac is the number of task allocation changes, mac is the number of message allocation changes, tpc is the number of task priority changes and mpc is the number of message priority changes.

$$f_{10} = \frac{\alpha_1 tac + \alpha_2 mac + \alpha_3 tpc + \alpha_4 mpc}{\alpha_1 + \alpha_2 + \alpha_3 + \alpha_4} \quad (12)$$

In reality, allocation changes are more costly than priority changes which is balanced using the weightings $\alpha_1, \dots, \alpha_4$.

It is possible to think of the cost of upgrading from a baseline configuration to a new configuration as a distance [12] as suggested in figure 3. Equation (12) does violate some of the properties of a distance function in that the distance between non-identical configurations can be 0. This is the case when one configuration is a subset of the other which can occur when, for example, some new tasks are added but the configuration of all existing tasks is identical. An alternative formula to that given in (12) would be to sum the squares of the numbers of changes. Other equations for measuring the cost of upgrade should be considered as future work.

The cost component pulls the search back towards the starting place. It is worth considering whether the upgrade cost component should be ignored once the distance between the configurations becomes too great. This would

represent the fact that an incremental update is possible but it is better to create a new configuration that will be more flexible to further changes. The point at which this should happen would change from project to project and is a possible avenue for future work.

5.4 Combining the Cost Function Elements

The final cost of a system under a particular configuration is calculated as a weighted mean of the cost components as shown in equation (13). Selecting appropriate weighting values can be guided by the component classes identified earlier. Those components which measure whether the solution is valid (equations (1), (2) and (11)) will be weighted highest. Equations (3), (4) and (5) guide the search by preventing it from moving out more schedulable regions. How highly these are weighted will affect how quickly the search stops; if they are weighted highly, the search will stop soon after finding a valid solution otherwise the search will spend longer looking to improve the desirable properties. The sensitivity equations, (7) and (9), both guide the search towards a valid solution and measure for desirable properties. Equations (6) and (12) are for finding a more desirable solution only. Again, how these are weighted will trade off how long the search takes to complete with the quality of the solution. If they are weighted too highly, there is a risk that the final solution will be invalid.

$$\text{COST} = \frac{w_1 f_1 + \cdots + w_n f_n}{\sum_{i=1}^n w_i} \quad (13)$$

Dividing by the sum of the weights ensures that the simulated annealing algorithm is not affected by absolute weighting values, only relative ones.

6 Scenarios As A Changeability Heuristic

A problem with using task sensitivity alone as a changeability heuristic is that it does not focus on any parts of the system. In general, it is impossible to make all of a system changeable. A future investigation could look at weighting task scalability differently for different subsystems according to some probability of change. That is, equation (7) is generalised to

$$f'_7 = \frac{\sum_{i \in \text{TASKS}} p_i \text{SENS}_i}{\text{num tasks}} \quad (14)$$

where p_i is the probability of task i changing. However, to do this requires a better understanding of change scenarios and a method of converting this into a probability of change for a particular task.

An alternative suggestion is to apply a number of scenarios and calculate the proportion of scenarios where requirements are met to those that are not met for a particular configuration [5]. However, some issues have been identified with this method. Firstly, if a scenario incorporates new tasks, there will be no configuration information for those tasks so a question arises of how to allocate and schedule the new tasks alongside existing tasks. To truly evaluate whether a scenario could be met would require performing a search within the search over the space of configurations for the new tasks. When a search may evaluate over

100000 solutions, this soon becomes infeasible and certainly would not scale to looking at more than one level of future requirements as proposed in figure 1.

Another issue is that purely counting whether a scenario is met or not does not give an accurate view of the system. Future scenarios are inexact in nature and hence it is desirable to evaluate how close a configuration is to meeting a future scenario. It may be impossible to meet a future scenario given the available hardware but some configurations will come closer to meeting requirements than others. Similarly, many configurations may be able to accommodate a future scenario but some may do it in a more flexible way than others allowing for a second tier of upgrades. Flexibility measures need to be applied to scenarios as well as the current system. This leads to the conclusion that the cost function presented in equation (13) should be applied to scenarios also.

The method chosen to incorporate scenarios into the task allocation tool is to search for a global configuration that will simultaneously meet requirements of several systems, one of which is the current baseline system and the others are future scenarios. If different systems contain different tasks and messages, only relevant parts of the global configuration are applied to that system. A new cost value incorporating scenarios can then be calculated using

$$\text{TOTAL_COST} = \frac{W_1 \text{COST}_1 + \dots + W_N \text{COST}_n}{\sum_{i=1}^N W_i} \quad (15)$$

where COST_i is cost of system i obtained from equation (13) and N is the number of systems. The weightings, W_1, \dots, W_N are associated with the particular system requirements becoming a reality. This results in a high weighting being used for the current system with lower weightings for scenarios.

To clarify the method, consider the following example. The baseline requirements for a system contains 30 tasks. Two future scenarios are to be considered when selecting a baseline configuration for these requirements. The first scenario only increases worst case execution times of some of the thirty tasks. The second scenario requires an additional two tasks as well as modifying some execution times. The set of configurations to be used in the search space will contain allocation and priority information for all 32 tasks and any messages required. To evaluate the cost of using a particular configuration for the baseline requirements and first scenario, information for the appropriate subset of 30 tasks will be selected from the configuration. To evaluate the cost of the second scenario, all of the configuration will be used. At a particular point in the search the cost of using a configuration on the baseline is 0.05, the first scenario has a cost of 0.08 and the second scenario has a cost of 0.14. Giving the baseline requirement the highest weighting, the total cost may be calculated as

$$\text{TOTAL_COST} = \frac{300 * 0.05 + 50 * 0.08 + 30 * 0.14}{380} = 0.061$$

Note that in this example, the cost of change component in equation (12) would not be used since we are creating a baseline specification. This component is only used when a baseline configuration already exists to be used as a starting point.

The weightings for scenarios could be based on probability of occurring or how far into the future they are likely to occur. A requirement derived from a phased development would be weighted higher than a desired but non-essential

feature which is unlikely to be implemented due to lack of available resources in the foreseeable future. Future work may look into using different weightings for different systems depending on the circumstances surrounding the particular system requirements. However, as has already been acknowledged, there is some difficulty choosing the weightings for a single cost function and selecting these for many cost functions would require a more systematic method. The evaluation in this paper uses a single weighting value for all scenarios which is low relative to the weighting for the current system. It is intended to demonstrate whether the method is successful in improving changeability and future work may be able to optimise it by placing more consideration into choosing the search parameters.

This method elegantly allows tasks not present in the current system to be configured using the existing heuristic search without requiring a nested search. The number of cost function evaluations which includes scheduling and scalability calculations increases linearly with the number of systems. Since some moves in the search will now be devoted to configuring tasks not present in the current system requirements, it may be necessary to increase the number of iterations performed.

7 Evaluation

The evaluation approach was to take a set of system requirements and produce a number of baseline configurations with and without scenarios. The flexibility of each baseline was measured by attempting to upgrade to a number of different scenarios and measuring the cost of upgrade using equation (12).

A number of experiments have been performed. Here, two smaller examples are presented to demonstrate the method. Two experiments were performed with two different sets of baseline system requirements. In the first experiment, the hardware was highly utilised and the upgrade scenarios represented a small system upgrade by increasing the worst case execution of the tasks in the range of [20%, 30%). In the second experiment, the initial system had a much lower utilisation and the scenarios added a number of tasks and messages as well as modifying some execution times. The different styles of these experiments are meant to represent realistic changes that might be introduced by a small patch and a more major upgrade such as a mid-life update. In each experiment 4 sets of 3 scenarios were produced. Of these, 3 sets were used as *training* scenarios to be used for generating baseline configurations. These represent future changes that the baseline may need to cope with. The different sets of training scenarios are used to create different baselines which anticipate different changes. The fourth set was used as *testing* scenarios to evaluate the flexibility of the different baselines configurations produced. These represent the actual upgrade that occurs. These changes in the testing set vary in how much they differ from the changes anticipated by the training sets.

The aim of the evaluation is two-fold:

- Validate the technique by ensuring that scenarios improve flexibility when handling anticipated changes.
- Investigate whether scenarios improve general system flexibility for handling unanticipated changes or whether using incorrect scenarios worsens the cost of upgrade.

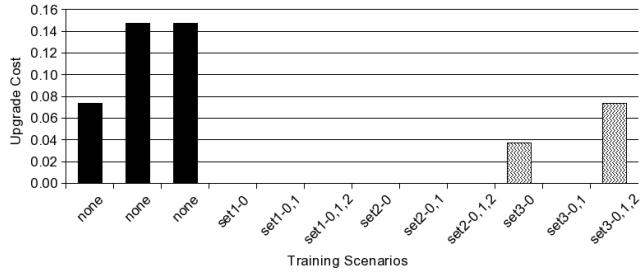


Figure 4: Experiment 1 results of upgrading to first test scenario.

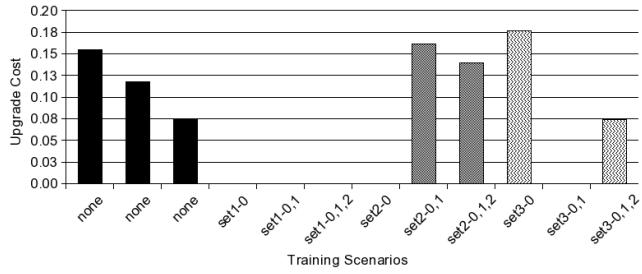


Figure 5: Experiment 1 results of upgrading to second test scenario.

7.1 Experiment 1 - Small patch

The first experiment took a baseline system of 11 tasks to be distributed over 3 processors. Each set of training scenarios modified the worst case execution time for 3 of the tasks, with a different 3 tasks chosen for each set. For a particular set, the same 3 tasks were modified in each scenario but by different amounts. Which task was modified can have a differing effect on the effort required to meet the scenario depending on how close the task is to missing its deadline. The test set was created by creating scenarios which modified the same 3 tasks as the first set of training scenarios but by different amounts. Therefore, baselines created using scenarios from the first set of scenarios should perform better in handling the changes when the upgrade test is performed. On the other hand, baseline configurations created from training sets 2 and 3 do not anticipate the changes correctly so the cost of changing these

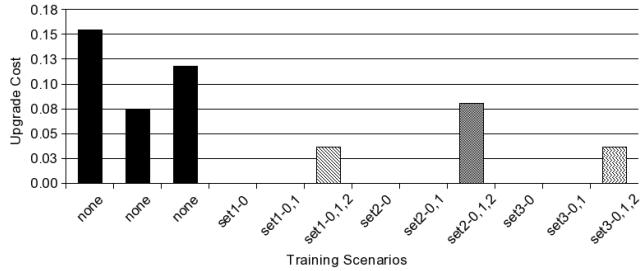


Figure 6: Experiment 1 results of upgrading to third test scenario.

baselines to meet the test scenarios should be higher. All scheduling analysis used distributed fixed priority scheduling[19] though the framework has been designed to allow different analyses to be used as required.

In total 12 configurations were used as baselines for upgrading to each of the 3 test scenarios. 3 baselines were produced without using scenarios. Any difference in how these perform is due to random factors in the method and so can be used to show how repeatable the method is. For each set of 3 training scenarios, a baseline configuration was created using just the first training scenario, two training scenarios and then finally for all 3. Training the baseline to meet more scenarios increases the time required to create the baseline in the hope that the baseline would be more flexible.

The results of the first experiment are shown in figures 4, 5 and 6. Each figure shows the cost of upgrade to a different test scenario from each of the baselines produced. The baselines produced with no scenarios are marked as *none*. The baseline created with a single scenario from a training set is labelled as *setx-0* where x denotes the training set. Baselines created using 2 or 3 scenarios are denoted as *setx-0,1* and *setx-0,1,2* respectively.

Each of the figures show the results of upgrading to a different test scenario. The results show that with all 3 test scenarios, use of the first set of training scenarios, which anticipated the changes correctly, greatly reduced the upgrade cost. There is a large variation in using baselines created from the other scenarios. Figure 4 suggests that using scenarios does improve the overall flexibility. However, figure 5 shows that, on occasion, it is possible for the baseline configurations trained with unanticipated scenarios to perform worse than the best configuration produced without scenarios.

The results do not show a benefit from using multiple scenarios. However, the multiple scenarios chosen all concentrated on a similar part of the system. In these experiments, due to the size of the systems, only one part of the system was chosen to be changed so that proportion of the system which was modified was realistic. In larger systems, it would be expected that the requirements would specify that two or three different subsystems should be optimised for modifiability. In this case, an increased benefit from multiple scenarios is expected.

7.2 Experiment 2 - Major upgrade

The second experiment was performed in a similar manner to the first with 3 sets of training scenarios and a set of test scenarios. In this case, the baseline system only contained 7 tasks distributed over 3 processors, allowing more room for expansion. For each scenario, 5 tasks and 3 messages were added as well as some execution time changes. Within each set of scenarios, the same tasks and messages were added but with different execution time changes. The tasks and messages added differed between each set of training scenarios. The test set added the same tasks and messages as the first set of training scenarios but with differing execution times. Therefore, as in the first experiment, we expected the baselines produced with the first set of training scenarios to perform best with respect to minimising the cost of the upgrade.

Figures 7, 8, 9 show the results for experiment 2. The baselines are labelled in the same way as the first experiment. In general they confirm the pattern found in the first experiment. When the changes are correctly anticipated, the

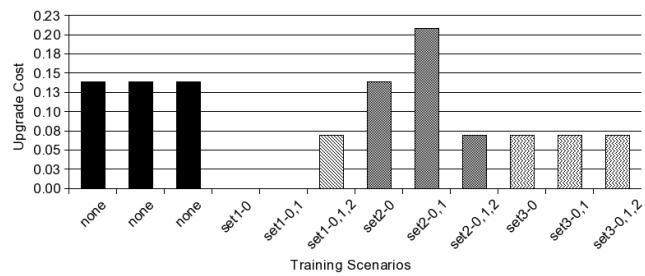


Figure 7: Experiment 2 results of upgrading to first test scenario.

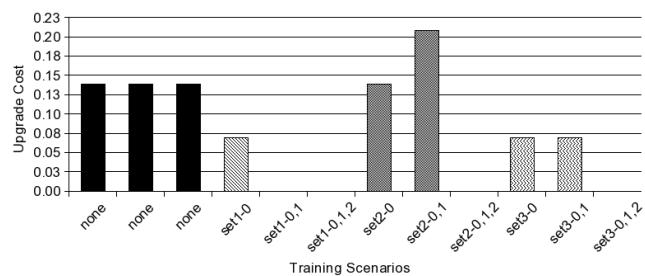


Figure 8: Experiment 2 results of upgrading to second test scenario.

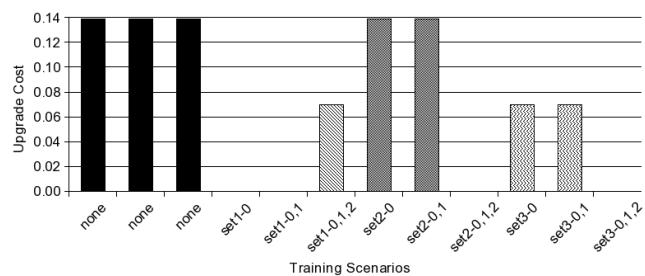


Figure 9: Experiment 2 results of upgrading to third test scenario.

scenarios clearly improve the cost of upgrade. However, with the unanticipated changes the results are more variable. In some cases, the baselines trained with incorrect scenarios performed better than those with correct ones. However, in the majority of cases, the baselines created with scenarios for both correctly anticipated and incorrect changes perform better than those without.

7.3 Further Experiments - Showing The Effects Of Scalability

Further measurements were made based on this second experiment. Some baselines were produced without using the general task and message sensitivity cost components described in equations (7) and (9). Not using these components has a number of implications. Firstly, it is more difficult for the search to differ between unschedulable solutions as these components calculate how much execution times need to reduce to make an unschedulable meet its timing requirements. For schedulable systems, the search is unable to differ in quality between two schedulable systems. This makes it more difficult for the search to converge and becomes more of a random search. The advantage of not using sensitivity analysis is that the current method is very computationally intensive. Maximum task execution times are determined using a binary search and the whole system has to be scheduled multiple times for each task to calculate these values.

Our experiments show that the cost of change from baselines produced with no sensitivity analysis and no scenarios approximately doubled to 0.28 over the baselines produced using no scenarios but with sensitivity analysis. The time required to produce the baselines decreased to a time of less than 30 seconds compared to several minutes for the comparative baselines produced with sensitivity analysis. As the search space is exponential in the number of tasks, scalability of the method must be considered. To deal with large systems, further work will look at methods which mix searching with and without scalability analysis to meet a compromise between performance and quality of solution. Also, it may be possible to improve performance of the scheduling analysis at the expense of some fidelity.

8 Conclusions and Future Work

This work has brought together the fields of scenario based analysis and automated evolutionary design. Scenario based analysis is usually performed as a series of meetings and manual changes to the architecture. However, as long as quantitative measures for the quality attributes can be defined, traditional methods can be used to elicit the scenarios but then applied to an automated tradeoff environment. It is hard to predict future changes exactly so it is important to be able to handle variations on the anticipated changes.

The space of possible design solutions must also be well defined. The work presented here assumes some high level architecture decisions have already been made in order to reduce the design space to an allocation and scheduling problem.

The method presented increases the flexibility of a system design by attempting to meet future requirements defined in scenarios at the time of designing

the baseline solution. This allows scalability to be targeted at relevant parts of the system and traded off with scalability of other subsystems. A simple metric has been devised for measuring the cost of upgrade by counting changes in allocation and priority ordering. More investigation is needed to make this more accurately correspond with actual cost of change. When the changes made match the anticipated changes, the method successfully reduces the cost of upgrade, often to requiring no changes to the original baseline. Creating baselines with unanticipated changes often also helps with coping with change. However, the results are more variable and occasionally produce worse results than not using scenarios due to the flexibility being targeted at the wrong areas of the system. A large number of weightings and parameters are included within the cost function so that the method can be tailored to different problems.

There are a wide variety of ways in which the future research can be developed. A few are listed below:

1. Definition and prioritisation of scenarios, not just to make a more flexible solution but also to improve the scalability and effectiveness of the search.
2. Taking full consideration of problem specific knowledge improve the efficiency and effectiveness of the search. This includes systematic methods for selecting the large number of search parameters and weightings.
3. How the effect of changes can be contained differently for specific parts of the design. For example, some sub-systems being more constrained than others.
4. Whether the search algorithm can be parallelised.
5. Applying the approach to other problem domains and properties, e.g. power consumption / dissipation and space.
6. How different computational models can be used to balance speed and accuracy, possibly changing the model for different stages of the search.

9 Acknowledgements

This work was funded by the Dependable Computing Systems Centre (DCSC), a joint venture between BAE Systems, the University of York and the University of Newcastle.

References

- [1] <http://www.rl.af.mil/tech/programs/MoBIES/>.
- [2] ARINC. *ARINC 653: Avionics Application Software Standard Interface (Draft 15)*. Airlines Electronic Engineering Committee (AEEC), June 17th, 1996.
- [3] J. Axelsson. Three search strategies for architecture synthesis and partitioning of real-time systems. Technical Report R-96-32, Department of Computer and Information Science, Linkoeping University Sweden, 1996.

- [4] I. Bate and N. Audsley. Flexible design of complex high-integrity systems using trade offs. In *8th IEEE International Symposium on High Assurance Systems Engineering*, pages 22 – 31, 2004.
- [5] I. Bate and P. Emberson. Design for flexible and scalable avionics systems. In *Proceedings of IEEE Aerospace Conference*, 2005.
- [6] I. Bate and T. Kelly. Architectural considerations in the certification of modular systems. *Reliability Engineering and System Safety*, 81:303–324, 2003.
- [7] A. Burns, S. Punnekkat, L. Stringini, and D. Wright. Probabilistic scheduling guarantees for fault-tolerant real-time systems. In *Proceedings of the 7th IEEE International Working Conference on Dependable Computing for Critical Applications*, pages 361 – 378, 1999.
- [8] C. A. C. Coello. A comprehensive survey of evolutionary-based multi-objective optimization techniques. *Knowledge and Information Systems*, 1(3):129–156, 1999.
- [9] R. Dick and N. Jha. Mocsyn: Multiobjective corebased single-chip system synthesis, 1999.
- [10] R. A. Edwards. ASAAC phase I harmonized concept summary. In *Proceedings ERA Avionics Conference and Exhibition*, London, UK, 1994.
- [11] G. Fohler and C. Koza. Heuristic scheduling for distributed real-time systems. Technical Report Research Report No. 6, Institut fur technische Informatik, Technische Universtate Wien, Austria, 1989.
- [12] L. L. Jilani, J. Desharnais, and A. Mili. Defining and applying measures of distance between specifications. *Software Engineering*, 27(8):673–703, 2001.
- [13] R. Kazman, L. J. Bass, M. Webb, and G. D. Abowd. SAAM: A method for analyzing the properties of software architectures. In *International Conference on Software Engineering*, pages 81–90, 1994.
- [14] R. Kazman, M. Klein, and P. Clements. Evaluating software architectures for real-time systems. *Ann. Softw. Eng.*, 7(1-4):71–93, 1999.
- [15] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science, Number 4598, 13 May 1983*, 220, 4598:671–680, 1983.
- [16] M. Nicholson. *Selecting a Topology for Safety-Critical Real-Time Control Systems*. PhD thesis, Department of Computer Science, University of York, 1998.
- [17] K. Sandström and C. Norström. Managing complex temporal requirements in real-time control systems. In *Proceedings of 19th IEEE Conference on Engineering of Computer-Based Systems*, pages 103–109, 2002.
- [18] I. Sommerville. *Software Engineering*. Addison Wesley, 7th edition, 2004.
- [19] K. Tindell, A. Burns, and A. J. Wellings. Analysis of hard real-time communications. *Real-Time Systems*, 9(2):147–171, 1995.

Reconfigurable Hardware Network Packet Scanning Structure

K. C. S. Cheng and M. Fleury

Department of Electronic Systems Engineering

University of Essex

{kcsche,fleum}@essex.ac.uk

Abstract

Denial-of-service attacks are an increasing problem in today's networks. Embedded security systems are required as purely software defences are unable to cope with packet rates on high-speed networks. Reconfigurable logic is well suited to the changing nature of the threat. This paper introduces a packet-scanning structure that can be applied to a range of development boards and target systems. Multiple clock domains allow the network interface to be decoupled from the security logic. The design has been accomplished through hardware compilation on a platform FPGA, a method appropriate to quickly matching changing threats. Automatic re-timing offers a way to optimize clock widths. Results show that this is a scalable solution.

1. Introduction

Embedded network security systems are on the increase as purely software approaches cannot cope with existing packet throughputs, let alone high-performance LANs such as Gb Ethernet. For example, Sourcefire Inc., well known for Snort --- a rule-based software search engine, have turned to an intrusion detection system (IDS) using up to ten G5 PowerPC processors [1] aimed at fibre-optic line-rates of 2--8 Gbps. SourceFire prefer not to use an ASIC, because these are not adaptable to new exploits. In this paper, we consider a look-up-table-based (SRAM) FPGA [2], which is reconfigurable but supports greater throughput than a RISC, as it does not suffer from the fetch-execute bottleneck and can process multiple parallel streams according to need. As a comparison, the AES encryption algorithm runs at 1.5 Gbps on a Pentium 4 (3.2 GHz) [3], at 12.2 Gbps on a Virtex XCV1000 FPGA [4], and 25.6 Gbps on an Amphion at 200 MHz clock speed [5]. Hardware security devices also bring reduced vulnerability to attacks against the device itself (from software afflictions such as virus, worms, and executable content) and can act as an insulating layer around a PC and its data.

The packet scanner considered herein provides passive defence at a home or campus PC. It could also be deployed as a boundary device, though with the growth in campus and corporate VLANs, active intrusion detection is preferred to firewall protection. Packet scanning also has a role in protection of routers in the network core, for example against Border Gateway Protocol (BGP) exploits. Packet scanning also differs from firewall protection [6] in that it can act against the packet payload and that stateful filters can be implemented, *i.e.* filters that are responsive to patterns of activity over time. Providing a timely response to threats is an important feature of an embedded security system, which is not simply a function of which hardware is applied to the task, but also depends on the ability to write/design the appropriate response. In turn, this depends on the design environment and a packet scanner system structure that will enable easy modification.

Packet scanners work in reactive mode, acting to thwart newly discovered exploits. Therefore, a hardware system should also be able to quickly adjust its response. Reconfigurable hardware in the form of a SRAM FPGA is well suited to this task but only if it is also possible to quickly re-program the array. Fortunately, from the point that a netlist of logic components and their interconnections is available, the process of place-and-route is largely automated,

unless optimised designs are required. There are three higher-level ways to arrive at a netlist: 1) by means of a hardware description language (HDL) [7]; 2) through a silicon compiler [8]; or 3) using a hardware compiler [9]. HDLs have the disadvantage that compilation times for large designs are lengthy [10], slowing down design iterations, though the range of associated tools allows low-level optimisations. A silicon compiler gives a succinct circuit description, often in an existing software language, *e.g.* [11]. A hardware compiler, as used in this design, converts a program or more accurately an algorithm into hardware. Hence, a hardware compiler exists at a higher level of abstraction allowing faster production of attack detection routines at a cost in control of the form of the output circuit. Since platform-FPGAs have become available from the two main manufacturers, Xilinx and Altera, gate (or rather slice) usage has become less critical, and certainly is not an issue for packet scanning routines (refer to Section 6).

Threat response time can also be improved if a pre-existing structure exists into which a scanning routine can be slotted. Device driver libraries have become available, such as Celoxica's PAL/PSL library (refer to Section 3.2), to allow an FPGA to interface to Ethernet and PCI busses, as well as standard computer peripherals such as RS-232, PS-2 and VGA. Network device interfaces are clearly of particular importance for a packet scanner. Beyond that a buffering structure is needed that will allow a set of scanning routines to work in parallel on one packet, while allowing arriving packets to be stored in a custom data-structure. Access to the buffer must be regulated in hardware to prevent over-write. This is a different problem to software concurrency control through software semaphores or monitors, as those solutions assume virtual concurrency simulated by an operating system scheduler. In this paper, we consider the design of a generic buffering structure that will meet the needs of a variety of scanning routines.

Suspicious content is identified in a number of ways. Signatures (unique byte sequences) can be matched against a packet's content (either header or payload) or conversely a hash of successive segments of a packet's content is matched against a database of such hashes. The signatures or hashes can be stored either in external SRAM banks, in block RAM on the FPGA [12] as conventional arrays or as Content-Addressable Memory (CAM). We have examined on-chip FPGA CAMs but a memory hierarchy, combining block RAM and SRAM [13], is also possible. Particularly in denial-of-service attacks, it is possible for an exploit to extend over a number of packets, requiring stateful filter structures such as counters and timers. Again, these can be pre-designed components of a generic structure. There are a variety of evaluation and development boards, such as the RC200/300 range from Celoxica Ltd, Tarari's content processor, Xilinx's ML300, Digilent Inc.'s XUP V2P, along with production boards [14], which implies that a structure that will also extend to future boards should be sought, not least because it will allow exchange of blocking routines or filters.

The remainder of this paper is organized as follows. Section 2 is a short review of other embedded systems for network security. Section 3 describes our FPGA design environment. We have used hardware compilation rather than a traditional hardware description language (HDL) such as VHDL or Verilog. Hardware compilation may allow a more direct translation from software algorithm description to hardware circuitry, which is more appropriate when a rapid response to a network threat is required. Section 4 is analysis of the type of denial-of-service attacks that embedded network systems are suited to. Section 5 is an analysis of our prototype design and Section 6 gives some preliminary results. Finally, Section 7 draws some conclusions and considers future research.

2. Related work

One category of software packet scanner, Dragon, Bro [15] and Snort [16], rely on exact string matching to locate offending packets. In an early Snort v. 1.6.3 implementation, Boyer-Moore algorithm was applied sequentially to each string, leading to an inefficient search [17].

(In the Boyer-Moore algorithm, a window is passed over the data, and a match is sought by searching backwards within the window.) In fact, some hackers purportedly sent worst-case data to befuddle search engines. Since then there have been improvements [18] including the use of the Aho-Corasick algorithm to search simultaneously for multiple strings. However, as tests confirm [19], most software IDS are insufficient at Gb Ethernet rates. In fact, Snort was originally developed for “small, lightly-utilized networks” [16], but unlike some other commercial products appears in open source form.

There are at least five approaches to hardware string matching [20]: 1) String matching as in the software approach; 2) non-deterministic finite automata (NFA); 3) comparators; 4) hashing, which involves approximate matching and, hence, can lead to false positives; 5) combinations of the others. The following designs were all implemented on Virtex FPGA. In [21] using approach 1), the well-known Knuth-Morris-Pratt (KMP) algorithm is employed to search a character at a time, matching by comparators. The main advantage of the KMP is said to be the ability to scale more readily, *i.e.* apply multiple rules by repeated application of the KMP, *i.e.* through pipelining. Approach 2) is applied in [22] to regular expressions. In [23], approach 3 is applied. Characters from a single packet are fanned out to a set of comparators, which is similar to the operation of a CAM. In [12], approach 4) is applied. A Bloom filter is employed to match multiple strings at the same time. For each string, multiple hash functions are applied, each function outputting one value from a finite set of values. The resulting bit pattern vector acts as the search string. Bloom filter matching only identifies candidate threats, implying that a further exact match is required if false matches are to be avoided. In [12], multiple Bloom filters are applied in parallel to the same packet. This implies that variable length IP packets must be buffered. Thus, higher memory usage is a disadvantage of this approach. A similar comment applies to the packet-wise parallelism applied in [23], in which a dispatcher places an arriving packet in one of four implemented content-scanners (regular expressions). On the other hand, logic usage grows according to the number of characters in character-based approaches, whereas this is not the case for hashing methods.

Choice of search algorithm is to some degree interchangeable, whereas it is less easy to alter the system architecture. In our initial work we have employed a flexible approach, able to adapt to a variety of algorithms and adjust its pace to the packet arrival rate. This implies packet-oriented rather than character-oriented parallelism, as only then can packets be buffered if there is a delay in processing a previous packet. A single packet can be scanned for multiple rules or multiple packets can be processed in parallel for different (or the same) rules.

3. Development environment

As outlined in Section 1, we seek a design environment that can enable a rapid response to new threats.

3.1 Hardware compilation

Handel-C [24] is a hardware compiler, which attempts to model a programming language in hardware, and outputs a netlist compatible with FPGA place-and-route tools. Software approaches to hardware [25] allow software to be readily ported to hardware or software to be synthesized from existing hardware designs. Based on ANSI-C, Handel-C adds extra features required for hardware development. These include flexible data widths, parallel processing and communication by channels between parallel threads. In Handel-C:

- Within a single clock domain, execution is clock synchronous.
- Assignment statements each require 1 clock cycle.

- Expression evaluation takes zero clock cycles, but results in propagation delay through corresponding combinational logic. Complex expressions will lead to long propagation delay, lowering the maximum clock speed. This will reduce the overall speed of the circuit.

Though the Handel-C model is clock synchronous, the channel primitive allows synchronized communication between parallel processes by means of a rendezvous. The channel allows the designer to neglect detailed timing issues when first preparing a design.

By means of multiple “main” blocks with associated clock statements, Handel-C supports multiple clock domains in the Xilinx Virtex series (from Virtex II). Communication across domains is through a shared buffer, which feature is built in to the design of Section 4. The channel is the only way for processes to communicate between two clock domains. In some designs, the application logic is slower than memory access. Therefore, data are assembled in several cycles [26] while an application completes. The clock speed of the I/O libraries in this design is restricted to 100 MHz and below, while the simpler application logic can run at faster speeds given a decoupled design.

Refinement of Handel-C programs is normally based on trial and error, as, though heuristics exist, there appears to be no direct relationship between making a change and a resulting improvement to the clock rate. As placement takes place automatically, propagation delays are not visible.

However, as hardware compilation is software oriented it allows a more direct translation of algorithms into hardware. Given that network threats can arise quickly, this programmatic way of design may allow quicker responses, provided there is a generic structure available.

3.2 Integrated Development Environment (IDE)

The Celoxica DK IDE, built around Handel-C, has the “look-and-feel” of MS Visual Studio. It incorporates a clock-accurate simulator. In DK, the user can choose to run simulation without any hardware, or generate the output in a standard Electronic Design Interchange Format (EDIF) netlist. (RTL VHDL output is also possible.) The ability to automatically re-time designs (introduce registers to decouple logic thus optimising timing) is an interesting improvement (see Section 3.1) supported in DK version 3. In some cases, timed logic can replace channels. We experimented with re-timing in our implementation.

The main feature utilised in the packet scanner design are the device driver libraries. The Platform Developer’s Kit (PDK) consists of three elements: the Data Stream Manager (DSM), the Platform Abstraction Layer (PAL) and the Platform Support Library (PSL) [27]. Only the latter two are used in the design. PAL is a thin wrapper layer around PSL, and hence we found no speed advantage from using PSL directly. PSL is a device specific layer, which is extensible. Figure 1 shows the relationship between the libraries.

3.3 Development board

The RC200 development board from Celoxica Ltd. was used to develop the structure. The principal features employed in the design were: Xilinx XC2V1000-4 Virtex-II FPGA [28]; 2 banks of ZBT SRAM providing a total of 4 MB; a CPLD for configuration/reconfiguration; an Ethernet MAC/PHY with 10/100 base-T socket; parallel port for bit-file download; and RS-232 serial port.

The RC300 board, which was released after this work commenced, provides two Gb Ethernet interfaces, making it more suitable for applications of this sort, as the packet stream can be released onto the Ethernet segment, rather than unrealistically outputting to a terminal window on the PC via the RS-232 interface. The Tarari Content Processor [29], incorporating

three Virtex II FPGAs, is custom-designed for packet content scanning. It uses a PCI bus rather than an Ethernet interface and dedicated FPGAs for input and output, allowing high throughput. However, it requires a double-width 64-bit, 66 MHz PCI bus, normally only present on multiprocessor PC servers.

The Virtex II FPGA, if used without modification, may not be ideal for security applications as the Xilinx's Jbits software tool [30] allows selective examination of the reconfiguration bitstream through the JTAG interface. However, Xilinx now provides bitstream encryption [31], though reconfiguration latency is increased. A triple-DES algorithm is applied, while two keys are stored in a small, battery-powered portion of on-chip memory. However, readback of the bitstream is not as potent a threat to packet scanners as it is to FPGA cryptographic devices for which keys or possibly algorithms are at risk. The main threat to scanners is probably a commercial one, as an FPGA is a standard part. An extensive survey of reverse-engineering attacks on FPGAs appears in [32].

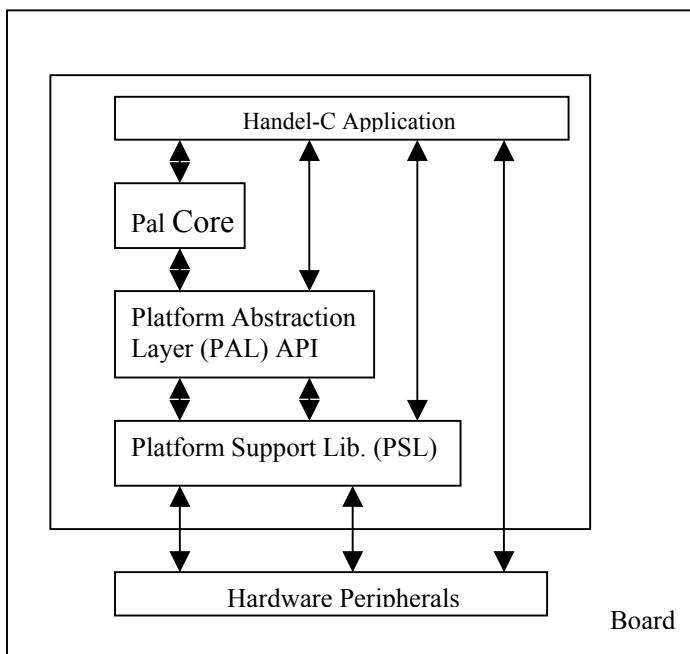


Figure 1: Relationship between application code and device driver libraries (after [27])

4. Fragmentation attack

Of the many different areas in network security suitable for a hardware solution, Internet Protocol (IP) (the main packet routing protocol) fragmentation was chosen as an example for these reasons: it is a network layer issue, which is simpler to tackle than complex transport protocols; the next generation IP version 6 (IPv6) also supports IP fragmentation; and in [33] the authors discovered that around 0.5% of the total traffic are fragmented packets. Although the relative volume of fragmented traffic is not high (though in absolute terms is considerable), it is quite common to have fragmented packets flowing around the networks.

As the maximum transport unit (MTU) can vary across a network path due to buffer sizes or data-link layer protocols, fragmentation allows a packet to be broken up into packets that fit within an MTU. The IP fragmentation mechanism is recursively applied at routers, with packet reassembly normally taking place at the end node. In [34] it is suggested that the disadvantages of IP fragmentation outweigh its advantages, and MTU discovery is an alternative. In [35], the authors offer some alternative remedies, as in some circumstances fragmentation can improve network performance. In security terms, IP fragmentation seems

to offer no advantages and only acts as a complication to other packet filtering. As transport-layer protocol headers are only contained in the first fragment (except in the aberration discussed in Section 4.1) packet filters may only process the first fragment and route the rest (assuming that as they cannot be reassembled they will do no harm). Other packet filters cache recent first fragments and the decision applied and re-apply the decision to succeeding fragments. This diversity and complication offers a threat to the successful application of a security policy. However, as fragmentation is widely deployed fragmentation attacks remain a threat.

The basic rules for IP fragmentation are:

1. All fragments must use the identification number of the original packet.
2. Each fragment must specify its offset in the original un-fragmented packet.
3. Each fragment must carry the length of the data carried in the fragment (minimum 8 B).
4. Each fragment must know whether there are more fragments after it.

A router accomplishes rules 1—3 by setting fields in the IP header. Rule 4 is accomplished by setting (or unsetting) a ‘more fragments’ flag within the flags field. The setting of individual flags is not reported by the `libpcap` library, which on Linux systems underlies some IDS, preventing setting rules to detect this type of exploit.

There are various kinds of IP fragmentation exploits; for a list, refer to [36]. However, many of these exploits first appeared some time ago and most operating systems and firewall software have addressed them with patches and upgrades. However, as fragmentation is widely deployed fragmentation attacks remain a threat, with the Rose attack [37] emerging in early 2004. Furthermore, the choice of IP fragmentation prevention is only as an example to demonstrate the idea of using reconfigurable hardware for network security.

4.1 Tiny Overlapping Fragment Attack

The Tiny Overlapping Fragment Attack is a combination of the “Tiny Fragment Attack” and the “Overlapping Fragment Attack” [38]. The target of these attacks is mainly Internet firewalls and the aim is to bypass firewall filtering.

In the Tiny Fragment Attack, the first fragment contains only the first eight bytes of the IP payload. In the case of TCP, this is actually the source and destination port numbers. The rest of the TCP header (most importantly the TCP flags field) will be stored in the second fragment. As a result, the firewall will not be able to test the TCP flags, and a harmful Telnet session could be established.

The Overlapping Fragment Attack exploits flaws in the reassembly algorithms. Two fragments are generated with overlapping offsets. The first one is legitimate, while the second one contains malicious information. Since the firewall only checks the first fragment, the two fragments will arrive at the destination. The first one will, however, be overwritten by the second one during reassemble to produce a malicious packet.

The Tiny Overlapping Fragment Attack is an enhanced version of the two attacks mentioned above. It consists of sending three fragments. However, the attack has a simple countermeasure in the form of two rules catching fragment offsets of zero or one [39], which is easily implemented in hardware. Strangely, this implies that one point where a hardware device would be placed would be to protect a firewall.

4.2 The Rose Attack

The Rose Attack's idea is very simple, the first fragment and the last fragment of a very large packet (64 KB) are sent, but not the middle fragments. The fragment buffer in the IP stack is held open for a certain period of time (That time for Microsoft Windows XP is 1 minute and Debian Linux is 30 s). If there are enough fragment pairs to fill the fragment buffer, no more fragmented packets are accepted. The attack is made effective by sending SYN packets, used to make an initial connection. This fragmentation exploit is one of a number which are denial of service attack, as they work to disable the re-assembler by causing it to reserve too much memory or computation time in the expectation of further fragments that never arrive. As some firewalls reassemble fragments there is again a threat to firewalls.

Alarmingly the author of this attack [37] describes how to set up a number of machines to generate a stream of fragments, spoofing addresses to prevent disablement of the offenders. The following effects of this exploit are stated:

- It causes the CPU to spike, thus exhausting processor resources.
- Legitimate fragmented packets are dropped intermittently.
- The machine under attack no longer accepts legitimate fragmented packets (unfragmented packets do not experience adverse effects) until the fragmentation ‘time exceeded’ timers expire.
- Buffer overflow can occur on intermediate routers, *i.e.* packets are dropped at high packet rates if there are not sufficient buffers allocated.

5. Packet Scanner Framework

This aim of the packet scanner structure is to simplify the development of packet inspection processes on an FPGA. It is achieved by the separation of the network interface from the packet operations. The packet scanner has been implemented using Handel-C on the RC200 development board (Xilinx Virtex-II XC2V1000 FPGA), as described in Sections 3.1—3.2. A functional block diagram of the structure is shown in Figure 2.

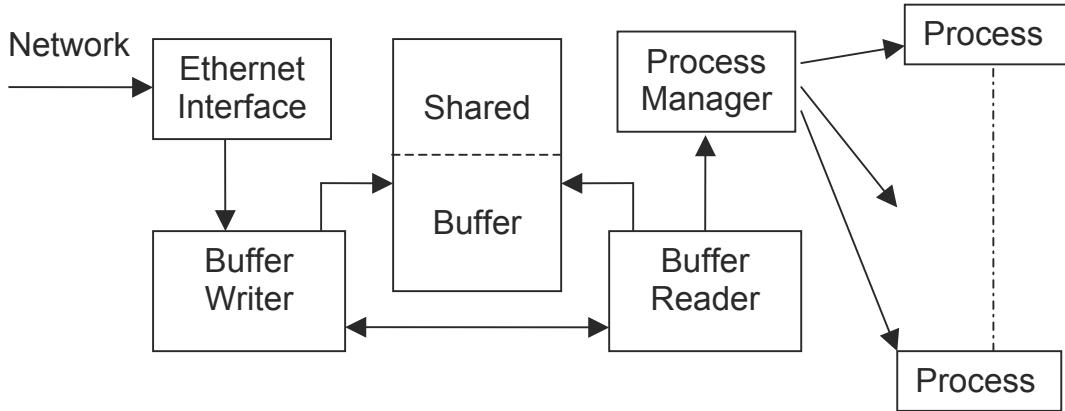


Figure 2: Functional Block Diagram of the Structure

A received packet comes in from the left. The Ethernet Interface is implemented by means of the PSL library (Section 3.2). Packets are read from the network and passed to the “Writer”, which then writes the packet (IP header in our example) into the 2×256 bit shared buffer. The shared buffer is implemented using Virtex-II dual-port block RAMs. The following code shows the structure of the shared buffer.

```

// Structure of the Multi-ported RAM
mpram SharedBuffer
{
    rom unsigned 256 Read[1];      // Read Only Port
    wom unsigned 8 Write[32];     // Write Only Port
};

mpram SharedBuffer Queue[MAX_QUEUE_SIZE] with {block = "BlockRAM"};

```

On the other side, the “Reader” reads the IP header from the shared buffer and passes it to the “Process Manager”. A Handel-C channel (Section 3.1) is used together with the buffer to implement a safe FIFO queue. Since the “Reader” needs only one clock cycle to copy the data from the buffer, there will be not any mutual exclusion problem in the design. (*i.e.* the “Writer” will not write into the location the “Reader” is currently reading.) By using a channel, exclusion problems are essentially packaged in the channel construct. The alternative to a channel, Handel-C’s semaphore, does not work across different clock domains. The implementation of the semaphore, unlike the channel, described in earlier research papers [40], also appears opaque, perhaps via Handel-C’s priority alternation construct or through a priority buffer or through a semaphore management process. In [40], a channel is implemented by two handshaking lines, ‘ready’ and ‘transfer’. The ‘ready’ line is asserted, through an OR gate, whenever any statement is ready to input and similarly the ‘transfer’ is asserted, through an OR gate, whenever a statement (in another process) is ready to output. The ‘ready’ and ‘transfer’ signals are ANDed together to create a ‘reg_load’ signal, which enables loading of a register to complete the transfer (and resets the channel for communication). The compiler checks that there is only one input and output statement that can communicate over a channel at any one time. Through use of handshaking, a channel is able to communicate across clock domains.

Reading a packet is illustrated in the following code. The code illustrates data width control, use of pointers in macro-calls, and the `par` construct, allowing nested parallelism. With the `par` constructs removed porting from ‘C’ and vice versa is a simple matter. `EthernetRead` is a macro written by us to call PSL library functions. Apart from macros each assignment takes one clock cycle to execute (though this may be in parallel) and all other statements take zero clock cycles. Notice the `delay` statements after each `if`. If omitted these may reduce the clock speed dramatically, as the logic depth increases. (This is reminiscent of the parallel language `occam`, from which Handel-C semantics are derived and which required a `skip` statement in place of the `delay` statement in Handel-C.)

```

void ReadPacket()
{
    unsigned 1 Error, Done;
    unsigned 16 Type;
    unsigned 48 Dest, Src;
    unsigned 11 Length;
    unsigned 5 Counter;
    unsigned 8 Data;
    unsigned 8 Temp[20];

    par
    {
        // Attempt to Read Packet
        EthernetReadBegin(&Type, &Dest, &Src, &Length, &Error);
        Counter = 0;
        Done = 0;
    }

    // If read was successful, read the packet data (IP Header)
    if ( Error == 0 )
    {
        // Process Ethernet Type 0x0800 only
        EthernetRead(&Data, &Error);
        if ( Data == 0x08 )
        {

```

```

EthernetRead(&Data, &Error);
if ( Data == 0x00 )
{
    do
    {
        EthernetRead(&Data, &Error);

        par
        {
            // Store the 20 Byte IP Header
            WriteBuffer1(Data, Counter);
            Done = ( Counter == 19 );
            Counter++;
        }
    } while ( !Done );
}

par
{
    EthernetReadEnd( &Error );
    SignalReader1();
}
}
else
{
    delay;
}
}

```

The “Process Manager” controls the different processes working on the header. Here is a code segment of an example process which checks for fragmentation threats. The code executes in one clock cycle, as the ‘if’ statements are all in parallel (and the nested `par` statements also all operate in parallel).

```

void FragmentProcess()
{
    par
    {
        // Rule 1
        if ( (header.ip_p == 6) && (header.ip_off == 0) && (header.ip_len < 40) )
            Pro1 = 1;                                // Rule 1 is matched
        else
            delay;

        // Rule 2
        if ( (header.ip_p == 6) && (header.ip_off == 1) )
            Pro2 = 1;                                // Rule 2 is matched
        else
            delay;

        // Rule 3
        if ( (header.ip_off != 0) || header.ip_flg )
        {
            par
            {
                Count[sec]++;
                TotalHit++;                         // Update Number of hits in this second
                Pro3 = 1;                            // Update Total hits
                if (TotalHit > MAX_ALLOWED)
                    Drop++;                         // Rule 3 is matched
                else
                    delay;
            }
        }
        else
            delay;
    }
}

```

The advantages of this design are as follows. Firstly, parallelism can be achieved between processes and within a process, and therefore throughput of the system is increased. Secondly, process design and implementation is independent of the network interface and buffer

management, which can be different on every development board. Moreover, thirdly, the use of a shared buffer not only allows pipelined reading from and writing to buffers, but it also provides another option to increase the throughput of the system by dividing the FPGA into two clock domains. The block diagram is shown in Figure 3. The reason for having two clock domains is because the network interface circuitry restricts the clock speed of the network interface. However, processes do not have such restriction since they are ‘pure logic’. By splitting the FPGA into two clock domains, throughput is further increased.

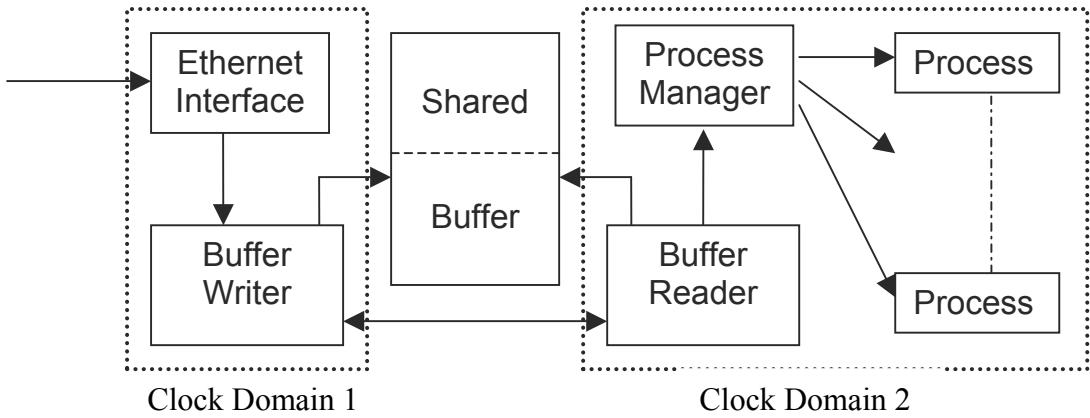


Figure 3: Structure divided into two clock domains

In the event of dual Ethernet interfaces, three clock domains are more appropriate. As previously remarked, on the RC300 board it was necessary to output test results elsewhere, in our case via an RS232 serial link to the PC system. This also requires three clock domains and the revised design shown in Figure 4.

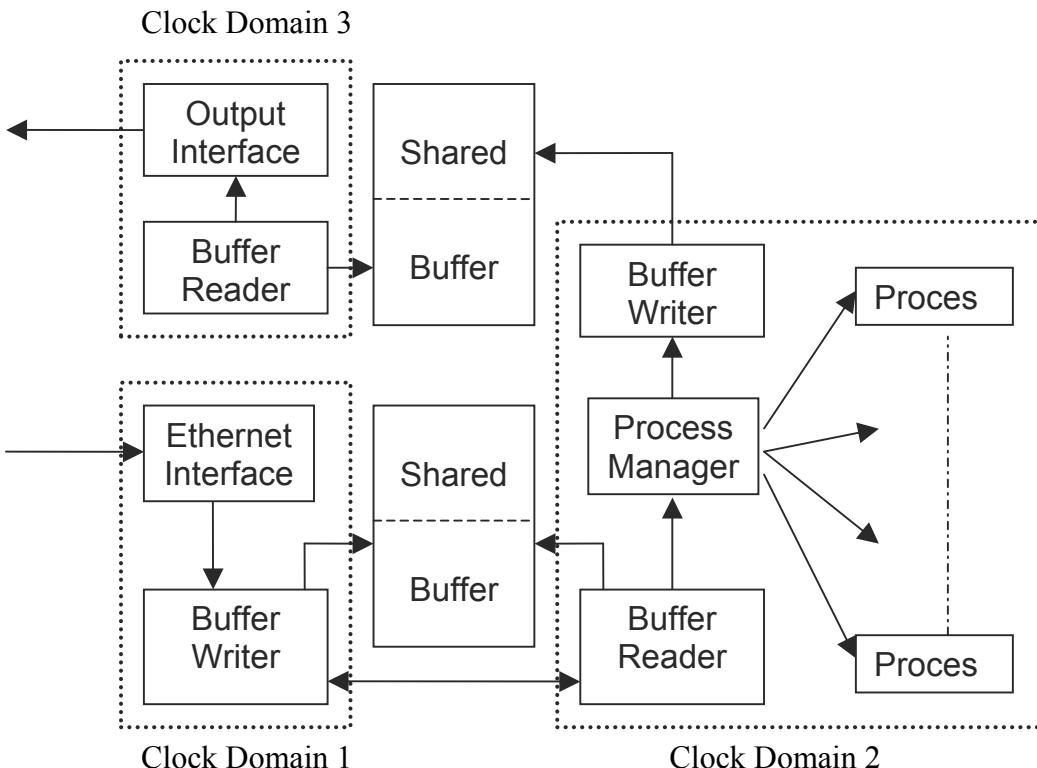


Figure 4: Structure divided into three clock domains

6. Results

In order to test the structure of Figure 2, with a single clock domain, an example of a process was needed. In this case, a single process tackling the IP fragmentation threat was implemented. It checks for certain patterns inside the header and counts the number of occurrences of fragmented packets in a period of time. This process takes one clock cycle to finish its operation.

Since the whole structure is implemented using the Handel-C PSL library, there are some constraints that must be matched. The fastest achievable clock rate for the Ethernet interface library code is 100 MHz. However, the maximum frequency achievable on RC200 is 300 MHz. The “Ethernet Interface” is potentially the bottleneck of the system, not only because there is a restriction of clock frequency but also because the number of clock cycles is also proportional to the size of the packet.

The resource usage of the structure is listed in Table 1. It runs at approximately 50 MHz, as buffering and calling the library code reduces the speed. All the processes used are identical, which is the *FragmentProcess()* of Section 5.

Number of Virtex-II slices		
No Process	1 Process	10 Processes
501 out of 5,120 (9%)	564 out of 5,120 (11%)	581 out of 5,120 (11%)

Table 1: Resource usage of the one clock domain structure

As only 11% of the Virtex-II device was needed for 10 processes, the indication is that it is possible to implement many more processes on the FPGA, which is obviously desirable in order to reduce the cost of deployment.

We also implemented the three-clock domain of Figure 4. To implement multiple clock domains, each domain is assigned a clock and a “main” function. The following code is the top-level control structure for the three domains:

```
// Clock Domain 1
#define RC200_TARGET_CLOCK_RATE 50000000          // 50 MHz
void main( void )
{
    // Run Ethernet in parallel with other code
    par
    {
        // Runs the device management tasks for the Ethernet interface
        EthernetRun( ClockRate1, 0x12345678dead );
        InitBuffer1();           // Initialize Shared Buffer 1

        seq
        {
            // Specifies initialization settings for Ethernet interface.
            EthernetEnable( RC200EthernetModeDefault );
            // Run the ReadPacket macro forever
            while ( 1 )
                ReadPacket();
        }
    }
    // Clock Domain 2
    #define RC200_TARGET_CLOCK_RATE 100000000         // 100 MHz
    void main( void )
    {
        par
        {
            RealTimeClock();      // Run the Real Time Clock
            InitBuffer2();         // Initialize Shared Buffer 2
            seq
```

```

    {
        InitCounter();
        // Run the ReadBuffer1 macro forever
        while ( 1 )
            ReadBuffer1();
    }
}

// Clock Domain 3
#define RC200_TARGET_CLOCK_RATE 50000000          // 50 MHz
void main( void )
{
    // Run the RS232 in parallel with other code
    par
    {
        // Run RS232 controller
        RS232Init(RC200RS232_115200Baud, RC200RS232ParityNone,
                    RC200RS232FlowControlNone, ClockRate3);
        seq
        {
            // Run the ReadPacket macro forever
            while ( 1 )
                ReadBuffer2();
        }
    }
}

```

In the implementation, clock domain 1 and 3 easily run at about 50 MHz. However, the ‘Buffer Writer’ in clock domain 1 takes at least forty clock cycles (number of bytes extracted for the IP header, 2 cycles per-byte) to load the buffer, whereas the ‘Buffer Reader’ takes one cycle in domain 2. Only one process has been implemented, which has two functions: 1) check for a “Tiny-Overlapping Fragment Attack” (Section 4.1) and 2) Count the number of fragmented packet received in the last 60 seconds. This process takes one cycle to operate (as mentioned in Section 5). The place-and-route algorithm depends on an initial estimate of the clock width (as specified within the Handel-C code) to reach its actual clock width.

Automatic retiming (Section 3.2) was found to offer a small reduction in the clock width for clock domain 2, traded-off against a small increase in slice usage. Table 2 presents the results, showing an example of retiming. Note that Table 2 is a snapshot, as optimizations to the code may result in small variations in clock speed. When retiming was applied (after compilation but before output of the final netlist and place-and-route), the clock speed increased from 89.0 MHz to 91.4 MHz. However, DK’s technology mapping (at the same stage in processing) also produced an improvement from 79.3 MHz to 89.1 MHz. Technology mapping can be enabled once the specific device is input.

Technology Mapper	Retiming	Virtex-II slices	Min. Clock width (domain 2 only)
No	No	692 out of 5,120 (13%)	12.610 ns
Yes	No	698 out of 5,120 (13%)	11.236 ns
Yes	Yes	739 out of 5,120 (14%)	10.936 ns

Table 2: Resource usage of the three-clock domain structure

6.1 Discussion

Partition of the design into separate clock designs did bring gains in relative clock speed in this implementation, as clock domain 2 now is approximately 100 MHz, whereas the same code ran at 50 MHz in the one clock domain structure. Partitioning is a useful strategy in terms of generic designs with differing I/O interfaces and interface code. Currently, though packet throughput would be around 100 M packet/s with a single process, the input interface reduces this by a considerable amount. Referring to the packet read code in Section 5, there is a minimum of 70 cycles to begin packet processing, while each byte read takes 2 cycles, with a further cycle to store in the shared buffer. Two of the bytes are from the Ethernet frame,

while the remaining 20 transferred to the buffer form the IP header. Finally, to complete takes 7 cycles. In total, this is 141 cycles at 50 MHz or 2,820 ns. All routers support at least 576 byte packets (which is between 42 byte TCP/IP minimal packets and 1500 maximal Ethernet frames). Thus the maximum throughput is 1.634 Gbit/s, assuming a continuous flow of packets, which easily copes with up-and-coming Gb Ethernet Metropolitan network backbones.

7. Conclusion

This paper has identified an area, network security, in which embedded systems will increasingly be deployed. The paper has also described appropriate design methods intended for rapid development of a design. If a pre-existing structure exists then hardware compilation is a way of quickly creating a process that will respond to a new threat from tainted packets. The structure designed in this paper has three clock domains, de-coupling the network interface from the application logic, and the input from output channels.

The paper represents preliminary results. Currently, the shared buffer has been deployed against packet headers, which are fixed in width. A weakness of shared buffer approaches (see Section 2) in general is that if the data payload is searched then the buffer slots must be of variable width. Dividing the payload into equal-sized chunks may solve this problem but the approach has still to be verified. Clock timings will be the main determinant of relative performance in comparison with the Snort-like software approach described in Section 1. A testing structure has already been constructed so that a stream of packets can be directed towards Snort and towards a packet scanner. Of course, this is on an isolated network. In fact, testing is not simple, as the software access to the raw socket interface is required to modify packet headers. Regulating access to a buffer by limiting the reader to a single clock cycle has a potential to increase the clock width if the reader is augmented in any way. Automatic re-timing offers some help in that respect. Minimizing the application clock domain clock width and the running speed is required for long running processes, so as to match packet arrival times. Hence, design of embedded systems for network security is a challenging task, which brings with it a range of new problems.

References

- [1] D. Storey, ‘Sourcefire 3D Suite Introducing IS5800’, presented at Sourcefire Seminar, London , April 2005.
- [2] S. D. Brown, R. J. Francis, J. Rose and Z. G. Vranesic, ‘Field-Programmable Gate Arrays’, Kluwer, Boston, MA, 1992.
- [3] H. Lipmaa, ‘Fast Software Implementation of AES’, 2002, <http://www.tcs.hut.fi/~helger/aes/rijndael.html>
- [4] K. Gaj and P. Chodowiec, ‘Fast Implementation and Fair Comparison of the Final Candidates for Advanced Encryption Standard using Field Programmable Gate Arrays’, CT-RSA 2001, N. Naccache (ed.), pp. 84-99, LNCS #2020.
- [5] Amphion, ‘High Performance AES Cores’, product description from <http://www.amphion.com/cs5240.html>
- [6] J. Moscola, J. Lockwood, R. P. Loui, and M. Pachos, ‘Implementation of a Content-Scanning Module for an Internet Firewall’, IEEE Workshop for Custom Computing Machines, 2003
- [7] K. C. Chang, Digital Systems Design with VHDL and synthesis, IEEE Computer Society, Piscataway, NJ, 1999
- [8] B. M. Pangrle and D. D. Gajski, ‘Design Tools for Intelligent Silicon Compilation’, IEEE Transactions on Computer-Aided Design, 6(6):1098-1112, 1987
- [9] H. Styles and W. Luk: Exploiting Program Branch Probabilities in Hardware Compilation. IEEE Trans. Computers, 53(11): 1408-1419, 2004
- [10] J. McCormack *et al.*, ‘Implementing the Neon: A 256-bit Graphics Accelerator’, IEEE Micro, 19(2):58-69, 1999
- [11] P. Hilfinger, ‘A High-level Language and Silicon Compiler for Digital Signal Processing’,

- [12] IEEE Custom Integrated Circuit Conference, pp. 213-216, 1985
- [13] S. Dharmapurikar, P. Krishnamurthy, T. S. Sproull, and J. W. Lockwood, ‘Deep Packet Inspection using Parallel Bloom Filters’, IEEE Micro, 24(1):52-61, 2004
- [14] D. V. Schuehler, J. Moscola, and J. W. Lockwood, ‘Architecture for a Hardware-based, TCP/IP Content-Processing System’, IEEE Micro, 24(1):62-69, 2004
- [15] J. W. Lockwood, N. Naufel, J. S. Turner, and D. E. Taylor, ‘Reprogrammable Network Packet Processing on the Field Programmable Port Extender (FPX)’, ACM Int. Symposium on Field Programmable Gate Arrays, 87-93, 2001
- [16] V. Paxson, ‘Bro: A System for Detecting Network Intruders in Real-Time’, 7th USENIX Security Symposium, 1998
- [17] M. Roesch, ‘Snort---A Lightweight Intrusion Detection for Networks’, 13th Systems Administration Conference, (LISA 99), pp. 229-238, 1999
- [18] C. J. Coit, S. Staniford, and J. McAlerney, ‘Towards Faster String Matching for Intrusion Detection’, 2nd DARPA Information Survivability Conference, pp. 367-373, 2001.
- [19] M. Norton and D. Roelker, ‘SNORT 2.0: High Performance Multi-rule Inspection Engine’, White Paper, Sourcefire Inc., Columbia, MD, April, 2004.
- [20] B. Yocum, R. Birdsall, and D. Poletti-Metzel, ‘Gigabit Intrusion Detection Systems’, Network World, 4 Nov. 2002,
<http://www.nwfusion.com/reviews/2002/1104rev.html>
- [21] G. Navarro and M. Raffinot, ‘Flexible Pattern Matching in Strings’, Cambridge University Press, Cambridge, UK, 2002
- [22] Z. K. Baker and V. K. Prasanna, ‘Time and Area Efficient Pattern Matching on FPGAs’, ACM Int. Symposium on Field Programmable Gate Arrays, pp. 223-252, 2003
- [23] R. Sidhu and V. K. Prasanna, ‘Fast Regular Expression Matching using FPGAs’, IEEE Symposium on Field-Programmable Custom Computing Machines, 2001
- [24] I. Soannis and D. Pnevmatikatos, ‘Fast, Large-Scale String Match for a 10 Gbps FPGA-based Network Intrusion Detection System’, 13th Int. Conf. On Field Programmable Logic and Applications, 2003.
- [25] M. Bowen, ‘Handel-C Language Reference Manual’, Celoxica Ltd, Abingdon, UK, 1998
- [26] M. Fleury, R. P. Self, and A. C. Downton, ‘Hardware Compilation for Software Engineers; an ATM Example’, IEE Proc. on Software, 148(1):31-42, 2001
- [27] M. Fleury, R. P. Self, and A. C. Downton, ‘Multi-spectral Image Processing on a Platform FPGA Engine’, Military and Aeronautics Programmable Device (MAPLD), 2005.
- [28] PAL (Platform Abstraction Layer User) Manual, Celoxica Ltd., Abingdon, UK, 2002
- [29] Virtex-II Platform FPGA Handbook, Xilinx Inc., San Jose, CA, 2001
- [30] Tarari . 1098 Technology Place, San Diego, CA, ‘Accelerating Content Processing’, White paper, 14 pages, 2002
- [31] S. A. Guccione, D. Levi, and P. Sundarajan, ‘Jbits: A Java-based Interface to FPGA Hardware’, 2nd MAPLD Conf. 1999,
<http://www.io.com/~guccione/Papers/Papers.html>
- [32] R. C. Pang *et al.*, ‘Nonvolatile/Battery-backed Key in PLD’, US Patent No. 6366117
- [33] T. Wollinger, J. Guajardo and C Paar, ‘Security on FPGAs: State-of-the-Art Implementations and Attacks’, ACM Transactions on Embedded Computing Systems, 3(3):534-574, 2004
- [34] C. Shannon, D. Moore, K. C. Claffy, ‘Beyond Folklore: Observations on Fragmented Traffic’, IEEE/ACM Transactions on Networks, 10(6): 709-720, 2002.
- [35] C. A. Kent, J. C. Mogul, ‘Fragmentation Considered Harmful’, SIGCOMM '87, 17(5) 1987
- [36] P. Chandranmenon and G. Varghese, ‘Reconsidering Fragmentation and Reassembly’, ACM Symp. On Principles of Distributed Computing, pp. 21-29, 1998
- [37] J. Anderson, ‘An Analysis of Fragmentation Attacks’, March 2001,
<http://www.ouah.org/fragma.html>
- [38] W. K. Hollis, ‘IPv4 fragmentation --> The Rose Attack’, 30 March 2004,
<http://seclists.org/lists/bugtraq/2004/Mar/0351.html>
- [39] G. Ziembra, D. Reed and P. Traina, ‘RFC 1858 – Security Considerations for IP Fragment Filtering’, October 1995
- [40] I. Miller, ‘Protection Against a Variant of the Tiny Fragment Attack’, RFC 3128, June 2001
- [41] I. Page, ‘Constructing Hardware-Software from a Single Description’, Journal of VLSI Signal Processing, 12:87-107, 1996

How to Manage Determinism and Caches in Embedded Systems

Richard G. Scottow and Klaus D. McDonald-Maier

University of Kent, Department of Electronics, Canterbury, Kent CT2 7NT, United Kingdom
{rgs3, K.D.Maier}@kent.ac.uk

Abstract - The purpose of the presented research is to design measurement tools to quantify a system's degree of determinism. The tools measure the execution times of real-time embedded tasks to assess the impact of enabling caches in real-time embedded systems. It is crucial for successful embedded real-time systems development, as caches will continue to be unused until developers can effectively manage the increased non-determinism. The presented measurement tools support development of novel algorithms to determine the optimal cache configuration for a given system. This paper presents results and tools that measure and analyse task execution times.

Keywords: Determinism, cache, measurement, embedded systems.

1. Introduction

Processor and memory speeds have developed at different paces for computer systems [1]. Research consequently has focused on techniques to reduce the processor idle time caused by the slower memory access time. One such technique is the caching of memory accesses.

A cache memory is able to supply the required information to the processor faster than main memory. A cache achieves this by being positioned closer to the processor, generally on the processor die, to minimise logic and signal delay. Advanced memory systems such as fast SRAM based storage cells have helped to further reduce delay. After the processor has initially requested information from main memory, it is then delivered to the processor and also stored in the cache according to a predefined policy. Any further requests for the same information can be retrieved from the faster cache instead of the main memory, thus improving the performance of general purpose computers.

Processor and memory speeds in embedded systems have recently started to diverge similarly to desktop

and server computers [1]. System manufacturers have responded by adding caches to their systems. However, designers of time-critical embedded systems, such as automotive engine controllers, can not easily take advantage of these caches as they introduce variable execution times, i.e. the delivery of the requested information to the processor becomes non-deterministic.

When caching is enabled for the whole system, the performance of the cache and therefore the software's execution time will depend on the following factors:

1. The software that has been previously executed and its location in memory. This will determine what information is available in the cache at a given instance in time.
2. Cache architecture factors such as the size, associativity, number of lines and line size.
3. Other cache features such as locking information into the cache, non-cacheable areas, replacement policies and write strategies are typical options, and often baffle designers.
4. How the software has been written, compiled and linked.

Many approaches to achieve high levels of determinism rely upon the architecture's constant processor-memory fetch time. This determinism is critical for real-time systems where the correctness of the system depends not only on the logical result of computation, but also on the time at which the results are produced [2]. In hard real-time systems earliest and latest deadlines must always be satisfied and in soft real-time systems only a certain percentage of these deadlines should be reached [3]. Presently caches are turned off when deterministic operation is required in these systems, thus avoiding these non-deterministic effects. However, these approaches can not be sustained for current and future embedded processors with ever increasing memory requests that the memory simply can not deliver. If system designers are to achieve the necessary performance, they need to obtain the relevant information affecting determinism which relates to their specific application

in order that informed decisions can be made. Optimisation of the numerous cache parameters that affect determinism for a particular software and hardware match is highly desirable.

2. Related Work

Early cache research has provided statistics such as ‘doubling the associativity decreases the miss rate by about 20%’ [4] and manufacturers have released hit / miss rate statistics as shown in Fig 1. [5, 6]. Such results are generalised and use cache simulators and software with no direct relevance to the system development tasks encountered by designers of real-time systems. Designers instead require actual measured figures from their target application and system to aid their decisions. Execution time is the best metric and is the key in real-time systems development [3], so this research is focused on measuring task execution time.

Performance related cache research has analysed the effect of varying cache configurations with SPEC benchmarks [7] and multimedia applications [8] by collecting memory traces to characterise memory usage. Other research has been conducted to statically analyse software with various methods and to conduct static cache simulations to calculate Worst Case Execution Time (WCET) [3] for overall programs,

methods to calculate the worst case cache related pre-emption delay (CRPD) and cache interferences has been avoided using cache partitioning [3]. One prior work removes paths not known to produce the WCET combined with instrumented measurement of small software blocks [9].

The disadvantage of these methods are that as most embedded systems make complex and sometimes asynchronous interactions with the external environment, it is impossible to fully replicate the system behaviour in a model and make offline measurements. Typically, hard real-time systems cannot be simply paused during the middle of execution while measurements are taken, as they often control mechanical parts requiring controlled shutdown. Hence this problem requires an online measurement method using the entire system comprising its hardware and software.

The presented method instruments the overall program with software around tasks where a record of the current time is desired and therefore directly measures the execution time of tasks in a system. By determining the variation in task execution time, the influence of caching on all selected real-time tasks can be analysed.

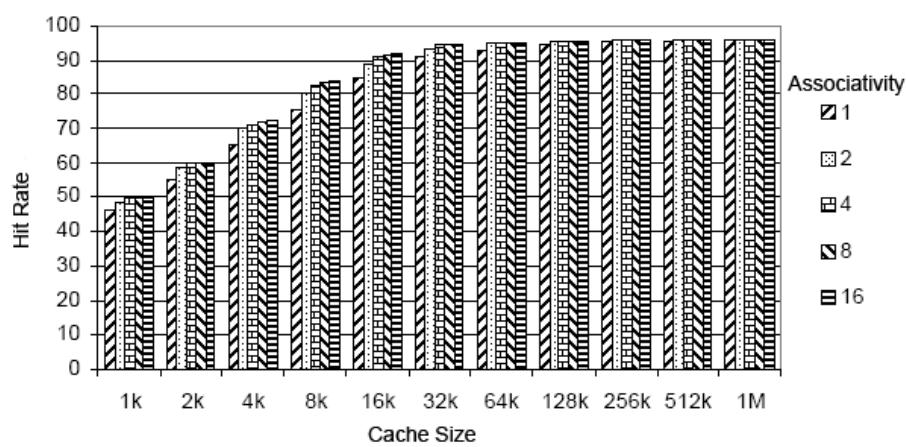


Figure 1: Cache hit rate while varying associativity and cache size [6].

3. Measuring Determinism

Two suitable online methods are:

1. Software instrumentation – Functions have instrumentation code inserted before and after their calls to record the value of a system timer. This has the following advantages:
 - a. It is processor independent and the instrumentation can be designed to require only changes to a device adaptation layer which provides the software with access to the system timer.
 - b. Real system operation is captured as the measurement code is added to the overall software.A disadvantage is that the instrumentation and storage of data will effect cache performance; However placing the instrumentation at a higher level of abstraction than functions, at the task level in a scheduler, would mitigate the effect compared with placement at lower code levels.
2. Hardware – Logic analyzers or similar can be used in conjunction with on-chip development support to trace and capture the internal bus transactions of a processor non-intrusively. This method also captures the operation of the real system and does not affect cache performance. It is however processor dependent.

When measuring execution time it should be considered whether an accurate variation in execution time will be found and measured, due to decisions in the software the shortest and longest path might not always be executed. This consideration could be reduced by testing the system in the typical states and in between states of program execution. Further analysis with profiling software, such as those available from Lauterbach [10] could provide much fine grained detail for particular scenarios to test the actual system.

A test bench as shown in Fig. 2 has been developed and is intended for generic operation using different processor models or actual silicon implementations of processors. It consists of a measurement library which provides the ability to initialize measurement at the start before the program core is executed, store the required data during measurement and finally to access the measurement data at the end of the program.

The measurement data consists of three fields:

1. Task identification (the address of the function).
2. Measurement type (start or end of task).
3. Timestamp.

The measurement library can easily be modified for example if a processor's development interface is used instead.

Tools have also been designed; an instrumentation tool to select the tasks to be instrumented and another tool to produce the execution time graphs for single tasks and those graphs where comparisons can be made between all the running tasks in the system.

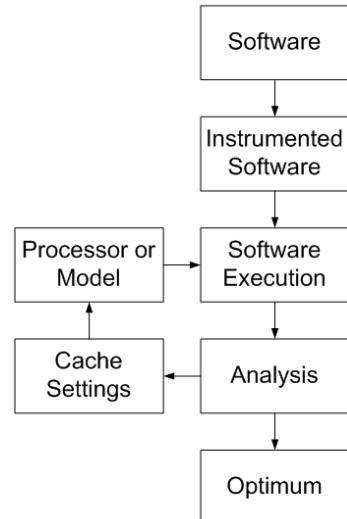


Figure 2: Measurement Overview

The instrumentation tool analyses a source file for functions that can have instrumentation code placed around and lists them, so that tasks to be measured can be selected. After selection the source file contains the appropriate calls to the recording method, `record(taskid,type)` around the selected tasks. This sequence is shown in Fig 3.

4. Results and Discussion

In order to demonstrate the methods and tools, a program with three basic tasks have been instrumented and executed on a SystemC PPC750 model [11] with twelve different cache configurations. The model uses SystemC [12] to realize a timed simulator and therefore an accurate system timer is available. The program, which has been compiled using GCC v2.95.3, has a static scheduler consisting of three tasks containing repeated division in task one, multiplication in task two, and addition in task three.

In these results the measurement data is stored in an unused part of system memory using the measurement library method `record` during execution and at the end of measurement the `record_end` method writes the results to a file.

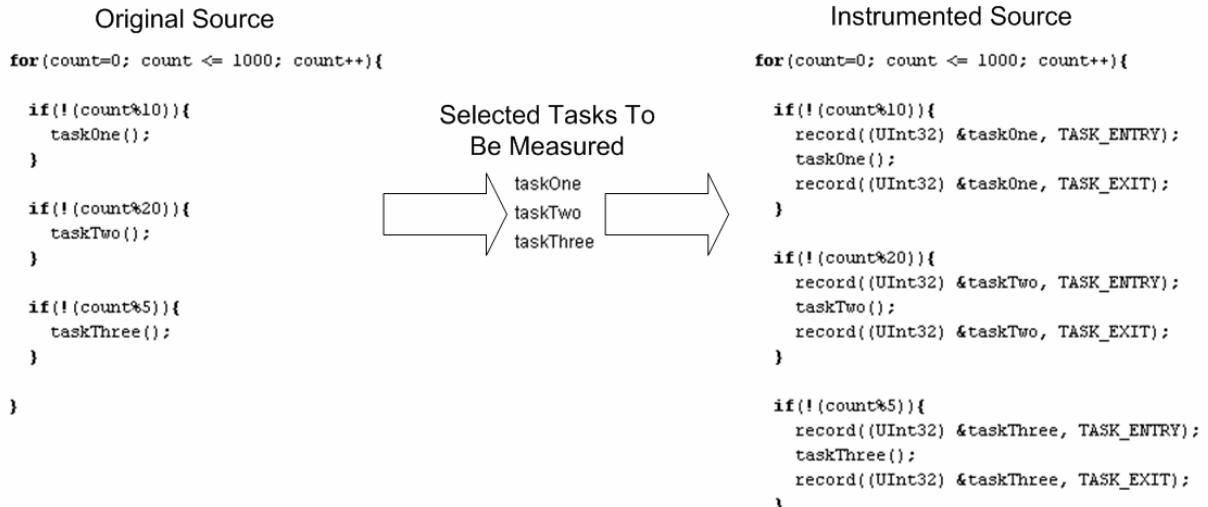


Figure 3: The instrumentation tool to analyse the source, select and instrument task

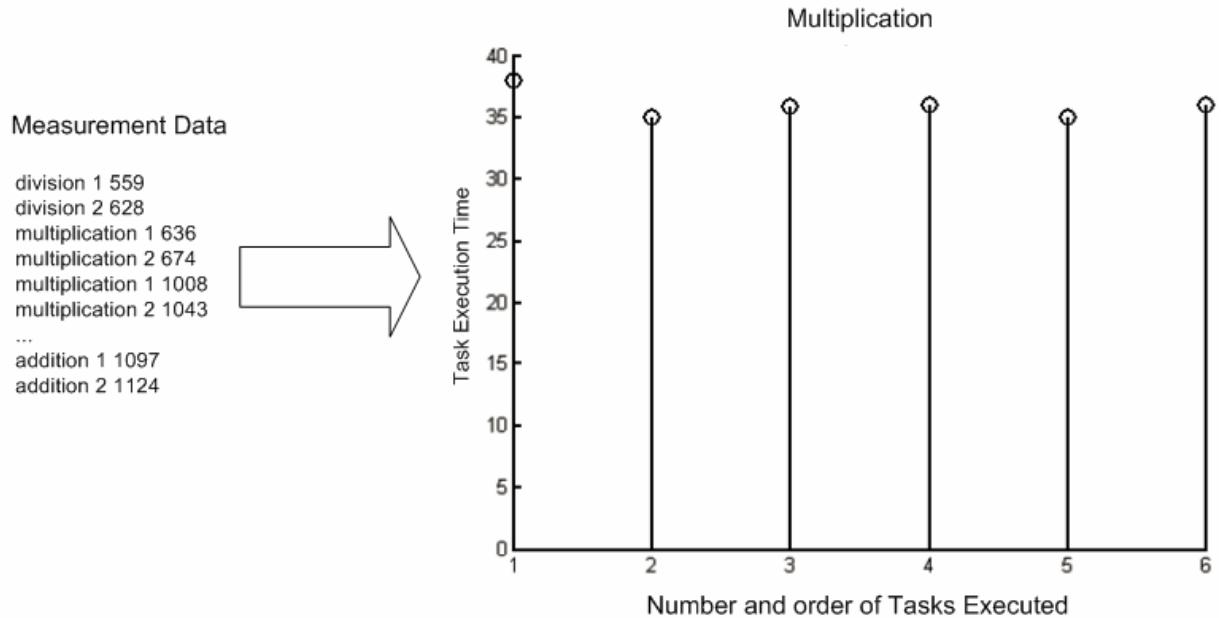


Figure 4: The viewer tool produces execution time graphs from measurement data

The tool to produce execution time graphs from the measurement data is similar to Fig 4 which is for the multiplication task under a single cache setting. It is produced from the measurement data stored in the format: taskid, type and timer value. This is useful because the execution time from the measured tasks is shown in order of execution and task statistics can also be displayed.

The graphs for comparing all the cache configurations are shown in Fig. 5. The execution times have been normalised to cache size 32KB, 8-way set associative and 128 lines. For each cache setting the average

execution times have been determined for program tasks and the maximum and minimum execution times measured shown as limits with \pm and \mp respectively.

The results show that for this task set while the cache size is reduced from 32KB down to 8KB the degree of variation in average execution time increases. The increase is not large, but will be limited by the minimalistic nature of the applied tasks (e.g. repeated division, multiplication and addition). The variation of the maximum execution time measured for the tasks changes significantly and increases in variation

as the cache size is reduced. Furthermore with all three tasks the minimum execution time measured is constant and very close to the average execution time.

These initial results demonstrate the level of non-determinism introduced by a cache. Clearly this can

be analysed using the presented approach. Further work will refine these techniques and allow their application to more complex and representative embedded software, such as industry standard benchmarks and task suites including scheduling so that the optimum cache settings can be found.

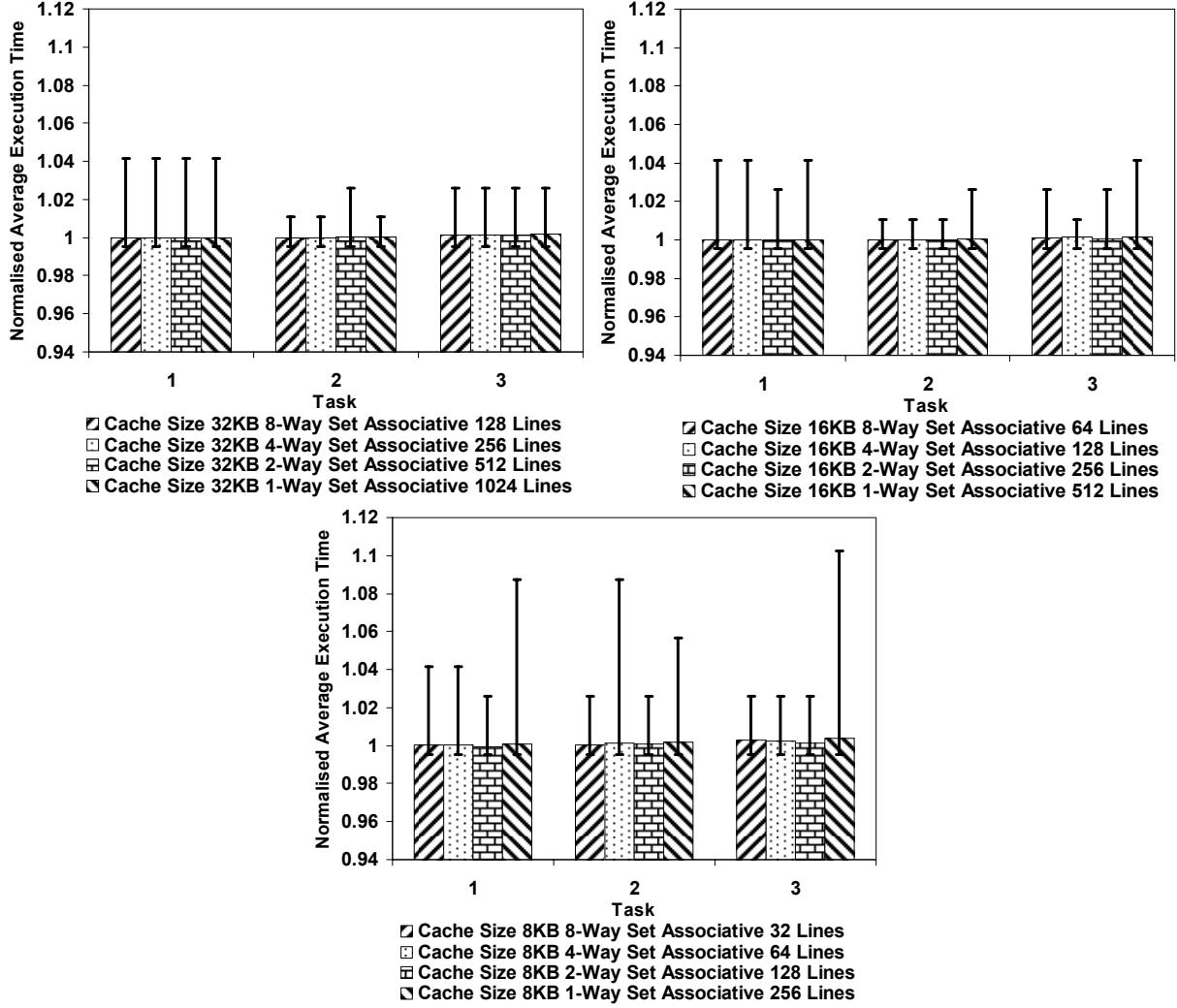


Figure 5: Variation of execution time for three tasks running on the PPC750 configured with different cache settings, normalised to cache size 32KB 8-way set associative 128 lines.

5. Conclusion and Future Work

Using the developed instrumentation and analysis tools, variation in the tasks' execution times can be determined for each of a processor's cache configurations. This therefore allows the analysis between the effect of cache settings and execution time. These tools are particularly suitable because they are processor independent, extendable, straightforward to use and would directly aid embedded real-time system design.

Systems are becoming increasingly complex; frameworks for systems, one such in development for automotive is AUTOSAR [13]. This will provide the ability for system designers to integrate software modules from other manufacturers. This means that the system designers will be required to integrate software into their design and have no influence on a number of the factors influencing determinism. With measurement of execution time system designers can be sure how the overall system behaves.

Combined with information such as task deadlines, a further tool would allow the developer to choose the optimum cache settings for the given application. Measurement code will be further investigated to assess its impact on the system's determinism. This will include comparisons with non-intrusive tracing using existing device specific on-chip development support, such as an IEEE-ISTO NEXUS standard based emulator [14].

6. Acknowledgements

This research is supported by the Engineering Physical Sciences Research Council (EPSRC) and Delphi Diesel Systems. We are especially grateful for the assistance from Stuart Jobbins and the Delphi Diesel Systems Software Development Team.

7. References

- [1] Semiconductor Industry Association (SIA) : International Technology Roadmap for Semiconductors, <http://public.itrs.net/>.
- [2] J. Stankovic : Real-Time Computing, Byte, 1992, pp. 155-160.
- [3] F. Sebek : Cache Memories and Real-Time Systems, Mälardalen University Technical Report 01/37, 2001.
- [4] M. D. Hill : Aspects of Cache Memory and Instruction Buffer Performance, University of California at Berkeley Technical Report CSD-87-381, 1987.
- [5] P. Genua : A Cache Primer – AN2663, Freescale Semiconductor, rev 1, 2004.
- [6] R.G. Scottow, K.D. McDonald-Maier, Measuring Determinism in Real-Time Embedded Systems using Cached Processors, ESA, Las Vegas, 2005
- [7] J.F. Cantin, M.D. Hill : Cache Performance for SPEC CPU2000 Benchmarks, v3.0, 2003, <http://www.cs.wisc.edu/multifacet/misc/spec2000cache-data/>
- [8] Z. Xu, S. Sohoni, R. Min and Y. Hu : An Analysis of Cache Performance of Multimedia Applications, IEEE Transactions on Computers, vol. 53, no. 1, 2004, pp. 20-38.
- [9] G. Bernat, A. Colin, S. Petters : pWCET: a Tool for Probabilistic Worst-Case Execution Time Analysis of Real-Time Systems, WCET, Portugal, 2003.
- [10] Lauterbach : TRACE32, <http://www.lauterbach.com>.
- [11] G. Mouchard : Microlib PowerPC 750 (G3) Simulator, v1.0.4, <http://www.microlib.org/projects/ppc750sim>.
- [12] Open SystemC Initiative (OSCI) : SystemC, <http://www.systemc.org/>.
- [13] AUTOSAR : Automotive Open System Architecture, <http://www.autosar.org/>.
- [14] IEEE-ISTO 5001™ : The Nexus 5001™ Forum Standard for a Global Embedded Processor Debug Interface, 2003.

Developing reliable embedded systems using a pattern-based code generation tool: A case study

Chisanga Mwelwa¹, Michael J. Pont¹ and David Ward²

¹*Embedded Systems Laboratory,
University of Leicester, University Road, Leicester, LE1 7RH, UK*

²*MIRA Ltd,
Watling Street, Nuneaton, Warwickshire CV10 0TU, UK*

Abstract

Automated code generation has developed over the last half century from techniques based on assembly language through high-level programming languages to those based on modelling languages (such as UML). We have previously argued that the use of design patterns to support automated code generation represents a logical next step in this evolutionary process. To support this claim, a prototype pattern-based code generation tool has been developed in the Embedded Systems Laboratory. In this paper, we describe the tool and illustrate its effectiveness by applying it in a non-trivial case study.

Acknowledgements

This work is supported by the UK Government (EPSRC Industrial CASE award) and by MIRA Ltd. Work on this paper was completed while MJP was on Study Leave from the University of Leicester.

1. Introduction

It is estimated that nearly a third of the cost of developing luxury passenger cars is spent on electronic and software systems such as airbags, electronic brakes and cruise-control systems [Bouyssounouse et al., 2005; Mullerburg, 1999]. These devices are expected to play a key part in achieving the main transport policy goal of the European Union (EU), which is to reduce fatalities on European roads by 50% by the year 2010. To achieve this goal, the EU believes that vehicle safety should be enhanced using appropriate technologies in order to design safer and more intelligent vehicles [EU-eSafety-Working-Group, 2003].

As embedded designs become more widespread and complex, and take on an increasing role in system safety, support for the developers of these systems is clearly required [Camposano et al., 1996]. Automated code generation holds the promise of reducing the time and effort required to implement such safety-critical systems, and eliminating many of the errors introduced in this stage of development [Whalen et al., 1999]. Industries such as aerospace and automotive have made extensive use of automatic code generation tools aimed at control and signal processing systems [Marsh, 2003; Schatz et al., 2003; O'Halloran, 2000]. These tools are used to model a system's design using the "Unified Modelling Language" (UML) and then to generate code from the design model. Hundreds of thousands of cars now rely on code generated using this type of approach [Marsh, 2003].

While the use of very high-level languages can facilitate rapid code generation, and reduce basic coding errors [Audsley et al., 2003; Whalen et al., 1999], such approaches do not directly support the developer as he or she makes important design decisions. More simply, high-level languages may help developers implement a particular design but they do not help developers identify an appropriate design to implement.

In order to support such design decisions, many people have proposed the use of "design patterns" [Damasevieius et al., 2003; Fowler, 2003; Rising, 2001; Douglass, 1999; Beck et al., 1996; Cline, 1996; Riehle et al., 1996; Gamma et al., 1995; Cunningham et al., 1987]. The development of pattern-based design techniques has become an important area of research in the software engineering community. Gradually, the focus has shifted from the use, assessment and refinement of individual patterns, to the creation of complete pattern languages, in areas such as telecommunication systems and applications with hardware constraints [Noble et al., 2001; Rising, 2001].

We have previously argued that use of appropriate “design patterns” can assist in the creation of reliable embedded systems [Pont et al., 2004; Pont, 2001]. Over the last decade we have assembled a collection of patterns to support developers working in this area. The particular focus of this work has been on “time-triggered” (TT) systems [e.g. see: Kopetz, 1997]. We have now described more than seventy patterns which we will refer to in this paper as the “PTTES[†] collection” [Pont et al., 2005; Pont et al., 2004; Pont et al., 2003; Pont et al., 2003; Pont et al., 2003; Key et al., 2003; Mwelwa et al., 2003; Pont, 2001; Pont, 2000; Pont, 2000; Pont et al., 1999].

Initially, work on patterns focused on a manual approach: as such, it is (sometimes implicitly) assumed that a developer will browse a catalogue, choose appropriate patterns and – possibly using some code examples or hardware schematics as a starting point – assemble a system. When working in this way, some users find it difficult to make the leap from the pattern description to a particular implementation (even though most patterns include example code). Other pattern users may have less trouble translating the pattern into code, but they still find this to be a tedious and time-consuming process [Lucredio et al., 2003; Budinsky et al., 1996].

In recent years various researchers have attempted to enhance the application of design patterns by developing pattern-based CASE tools [Lucredio et al., 2003; Cinneide, 2000; Florijn et al., 1997; Budinsky et al., 1996; Meijers, 1996; Alencar et al., 1996]. Despite the fact some interesting results have been obtained from these attempts, there is no widely used pattern-based tool. We have recently described a prototype pattern-based tool – “PTTES Builder” – which is intended to support pattern-based automatic code generation using the PTTES collection [Mwelwa et al., 2004(b); Mwelwa et al., 2004(a); Mwelwa et al., 2003]. In this paper our aim is to illustrate the potential of pattern-based automatic code generation by demonstrating how PTTES Builder can be used to develop code for a non-trivial embedded system (a cruise-controller for a passenger car).

The paper is organised as follows: Section 2 provides an overview of the PTTES collection and Section 3 briefly describes PTTES Builder. Section 4 gives an overview of the cruise control system (CCS) implemented using PTTES Builder and Section 5 describes the development of the CCS. Section 6 concludes the paper.

[†] PTTES = “Patterns for Time-Triggered Embedded Systems”

2. Patterns for Time-Triggered Embedded Systems (PTTES)

Since 1996, we have been assembling a collection of patterns to support the development of software for systems using a time-triggered co-operative (TTC) architecture. These patterns are today referred to as the Patterns for Time-Triggered Embedded Systems (PTTES) collection [Pont, 2001]. At present there are more than seventy patterns in the collection [Pont et al., 2005; Pont et al., 2004; Pont et al., 2003; Pont et al., 2003; Pont et al., 2003; Key et al., 2003; Mwelwa et al., 2003; Pont, 2001; Pont, 2000; Pont et al., 1999].

Some of the patterns currently included in the PTTES collection are[‡]:

- Processor patterns (e.g. STANDARD 8051, SMALL 8051 and EXTENDED 8051) supporting selection of an appropriate processor with CPU performance levels, memory, port pins, etc, which match the needs of the application.
- Oscillator patterns (e.g. CRYSTAL OSCILLATOR and CERAMIC RESONATOR) allowing an appropriate choice of oscillator type, and oscillator frequency to be made, taking into account system performance (and, hence, task duration), power-supply requirements, and other relevant factors.
- Various schedulers. For example the “Shared-Clock” schedulers (e.g. SCC SCHEDULER, SCI SCHEDULER (DATA), SCI SCHEDULER (TICK), SCU SCHEDULER (LOCAL), SCU SCHEDULER (RS-232) and SCU SCHEDULER (RS-485)) describe how to schedule tasks on multiple processors, while still maintaining a time-triggered system architecture. Using one of these schedulers as a foundation, the pattern LONG TASK describes how to migrate longer tasks onto another processor without compromising the basic time-triggered architecture.
- LOOP TIMEOUT and HARDWARE TIMEOUT describe the design of timeout mechanisms, which may be used to ensure that tasks complete within their allotted time.
- MULTI-STAGE TASK illustrates how to split up a long, infrequently triggered task into a short task, which will be called more frequently. PC LINK (RS232) and LCD CHARACTER PANEL both implement this architecture.
- HYBRID SCHEDULER describes a scheduler that has most of the desirable features of the (pure) co-operative scheduler, but allows a single long (pre-emptible) task to be executed.

3. PTTES Builder

The present version of PTTES Builder supports eleven design patterns from the PTTES collection: STANDARD 8051; EXTENDED 8051; CRYSTAL OSCILLATOR; CO-OPERATIVE SCHEDULER; PORT WRAPPER; HEARTBEAT LED; ONE-SHOT ADC; PID CONTROLLER; PULSE COUNT; RC RESET and PC LINK [Pont, 2001; Mwelwa et al., 2003]. These patterns are stored in the tool’s repository in the form of a pattern description and one or more (partial) “Pattern Implementation Examples” (PIEs): see [Pont et al., 2005].

[‡] Please note that the names used here match those used in the original PTTES collection [Pont, 2001] and do not reflect the changes in the structure and naming of the collection which have been made more recently (see: [Pont et al., 2005]).

Very briefly, the user interacts with the tool via a GUI-based “wizard”. This wizard guides the user in the selection and implementation of patterns from the repository. The code generator is “fed” PIEs associated with the selected patterns: it “fills in the gaps” in the PIEs (according to the user’s requirements) and “glues” the completed PIEs together, in order to create a code framework for the project.

Further information about PTTES Builder can be found in the following papers: [Mwelwa et al., submitted; Mwelwa et al., 2004(b); Mwelwa et al., 2004(a); Mwelwa et al., 2003].

4. Case study: Developing a cruise control system

In this paper our aim is to illustrate the potential of pattern-based automatic code generation by demonstrating how PTTES Builder can be used to develop code (and some of the hardware design) for a non-trivial embedded system. The example chosen is a cruise-control system model (CCS) for a passenger car. The example employs a Hardware-in-the-Loop test-bed developed in our lab [Ayavoo et al., 2005].

A CCS is intended to provide the driver of a car with an option of maintaining the vehicle at a desired speed without further intervention. Such a CCS will typically have the following features:

- 1) An ON / OFF button to enable / disable the system.
- 2) An interface through which the driver can change the set speed while cruising.
- 3) Switches on the accelerator and brake pedals that can be used to disengage the CCS and return control to the driver.

For the purpose of this paper, the specification of the CCS was simplified such that the vehicle was assumed to be always in “cruise” mode. While in cruise mode, a “speed dial” was available to allow the driver to dynamically change the car speed. The tasks used to implement the CCS are illustrated in Table 1. This system was to be implemented on an Infineon C515C microcontroller.

A computational model was used to represent the car environment in which the CCS would operate. The core of the car model was a simplified physical model based on Newton’s laws of motion. This model had one input (current throttle setting) and one output (a train of pulses representing the speed of the car). The car model was implemented on a desktop PC running the DOS operating system [Ayavoo et al., 2005]. Figure 1 illustrates the CCS setup.

Table 1: CCS tasks

Task Names	Task Description	Task Period (ms)
LED Flash Update	Flashing LED to indicate that the board is working	1000
Compute Car Speed	Computes the car speed obtained from car model	50
	Calculates and sends the required throttle to be applied to the car model	
Compute Throttle		50
Get Ref Speed	Gets the desired speed from the driver	1000

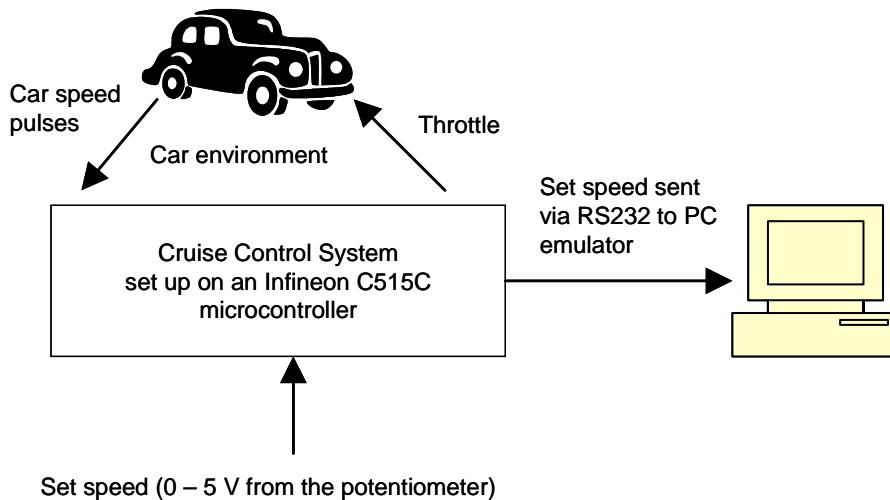


Figure 1 An overview of the CCS testbed ([Ayavoo et al., 2005])

5. Development of the CCS

We assume that the project begins with the development of a suitable hardware platform followed by the software implementation. The following sub sections give a step-by-step account of the development process, beginning with the hardware design.

a) The hardware platform

As noted in Section 3, the CCS was to be implemented using an Infineon C515C processor. The Infineon C515C is an EXTENDED 8051 processor which has 3 timers, a UART and an analogue-to-digital converter (ADC), among other hardware features.

The other hardware issues to be considered at this stage are the design of suitable oscillator and reset circuits. The development board used in this project includes these features and - in this project - the only configuration required on the board was the setting of the oscillator frequency (10 MHz).

Figure 2 and Figure 3 illustrate the selection of the hardware platform using the tool.

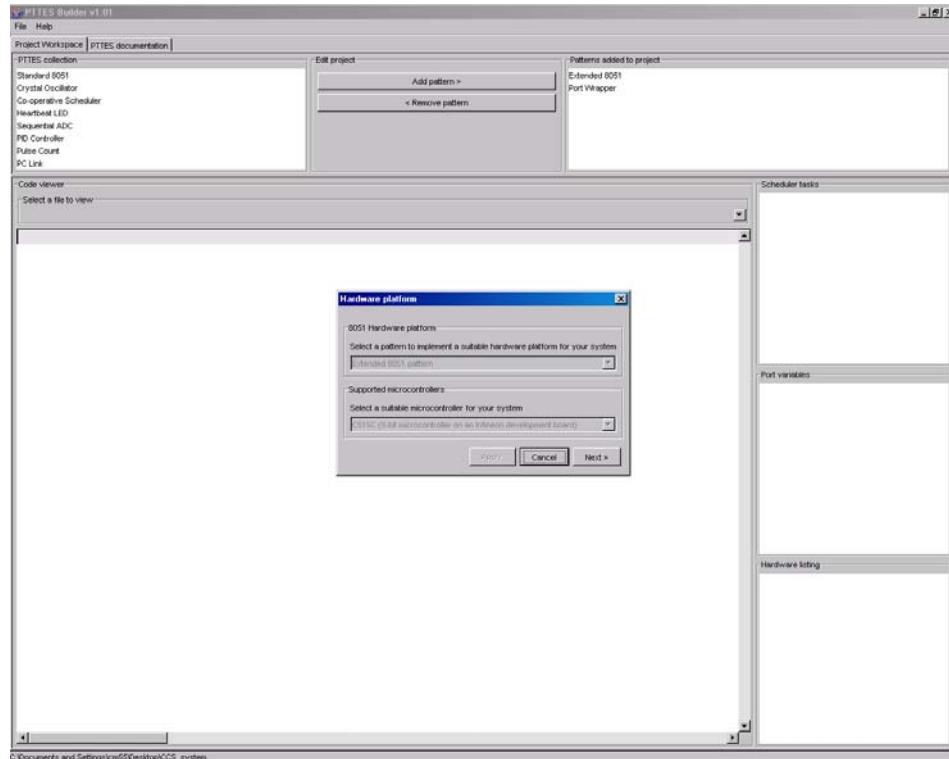


Figure 2: The first step in implementing a system using the tool, is the selection of an appropriate hardware platform, the EXTENDED 8051 was selected

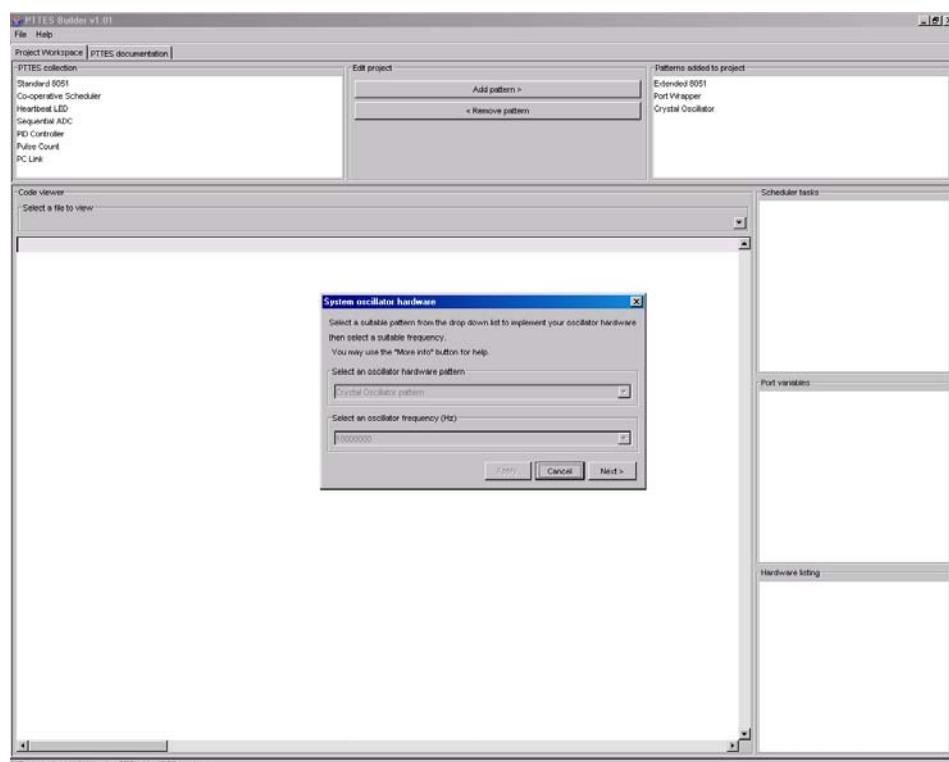


Figure 3: Based on the microcontroller selected, an appropriate oscillator frequency is set (10 MHz in this case)

b) Scheduler

At the heart of most computer systems is some form of scheduler. An embedded system is no different: it requires a scheduling system to organise and prioritise the execution of tasks.

The PTTES collection describes a range of scheduling algorithms [Pont, 2001]. At present, PTTES Builder only supports CO-OPERATIVE SCHEDULER. This scheduler is suitable for the CCS and was therefore used; Figure 4 shows PTTES Builder being used to select and configure the CO-OPERATIVE SCHEDULER implementation; Timer 2 was selected as the source of system ticks and a 1 ms timer overflow was set.

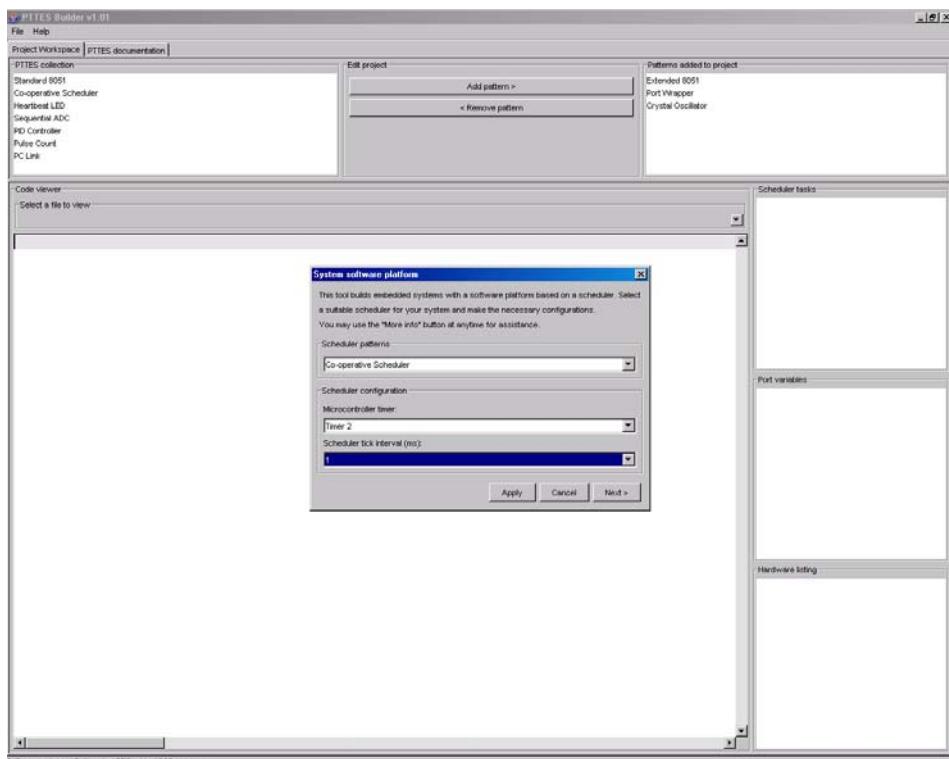


Figure 4: Selecting the COOPERATIVE SCHEDULER and configuring it

c) Tasks

We consider the implementation of the CCS tasks (see: Table 1) in this section.

LED_Flash_Update

The task `LED_Flash_Update` is implemented using the pattern HEARTBEAT LED [Mwelwa et al., 2003]. This is a simple pattern that is used to control a single flashing LED: provided the LED is

flashing (at 0.5 Hz, 50% duty cycle), all is well. Figure 5 shows how this pattern is implemented using PTES Builder.

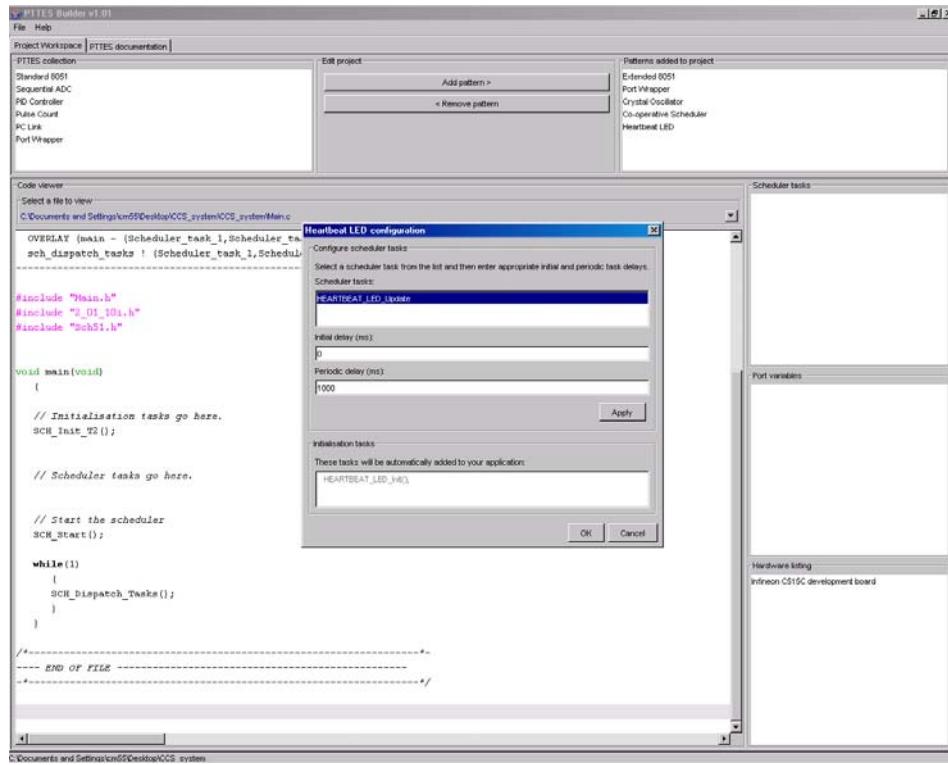


Figure 5: HEARTBEAT LED is configured to an initial delay of 1 ms and a periodic delay of 1000 ms

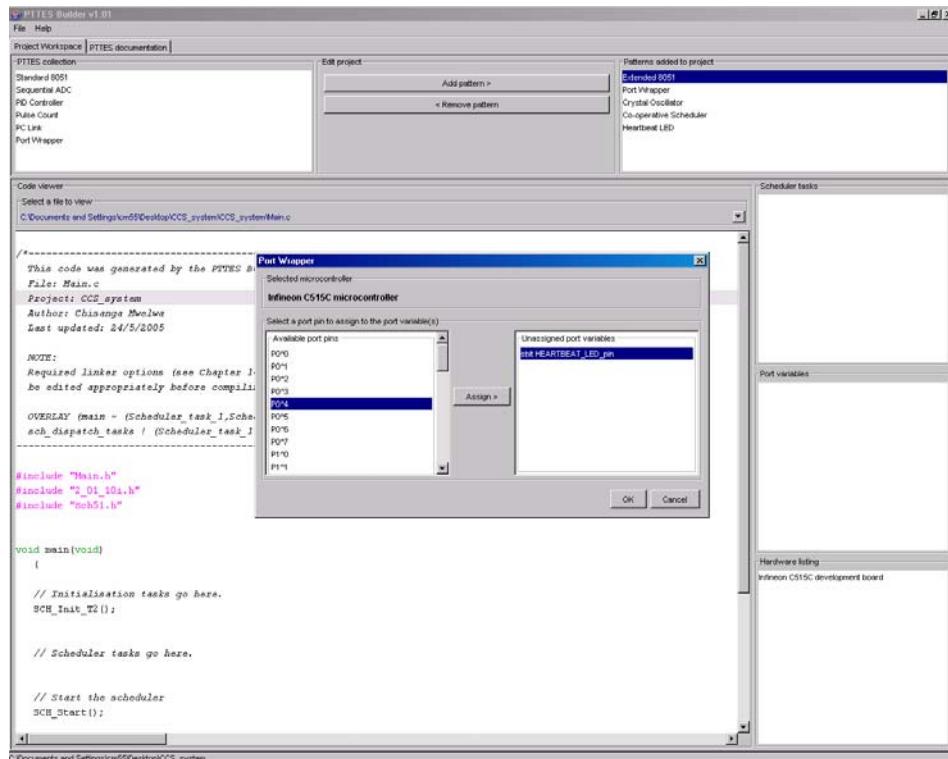


Figure 6: PORT WRAPPER is associated with every pattern that interfaces a microcontroller port(s)/pin(s). Here it is used to select a port pin on which to flash the HEARTBEAT LED task

Sens_Compute_Speed

The task Sens_Compute_Speed uses an on board timer / counter (Timer 0) to count the number of pulses that have been sent out from the car model. This pulse rate (assumed to arise from an optical or magnetic pulse encoder in a real vehicle) provides an indication of the current speed of the car. The raw pulse count is then scaled to obtain the actual car speed. This task is scheduled every 50 ms.

To implement this task, the pattern HARDWARE PULSE COUNT is used [Pont, 2001]. This implements a task capable of counting pulses received by the microcontroller from an external peripheral.

Figure 7 shows the configuration of this pattern.

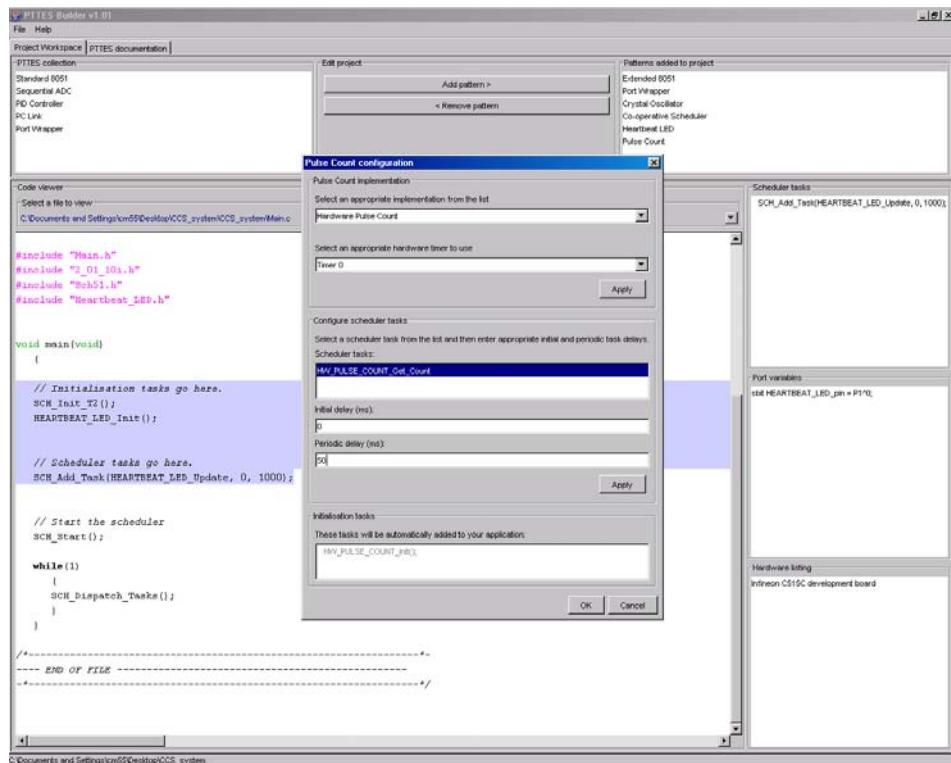


Figure 7: HARDWARE PULSE COUNT is configured and added to the project

Compute_Throttle

The task Compute_Throttle uses information about the car's current speed and the set (required) speed and – through a PID control algorithm [Ogata, 2002] - calculates the required throttle position. The throttle position is then scaled to an 8-bit value and sent to the “car”. This task is scheduled every 50 ms.

To implement this task, the pattern PID CONTROLLER was employed [Pont, 2001]. Please note that, as illustrated in Figure 8, PTTES Builder, in addition to configuring the PID CONTROLLER tasks, also allows the user to set the PID algorithm parameters (PID_KP = 0.005, PID_KI = 0.0000, PID_KD = 0.01, PID_WINDUP_PROTECTION = 1, PID_MAX = 1, PID_MIN = 0 and SAMPLE_RATE = 20).

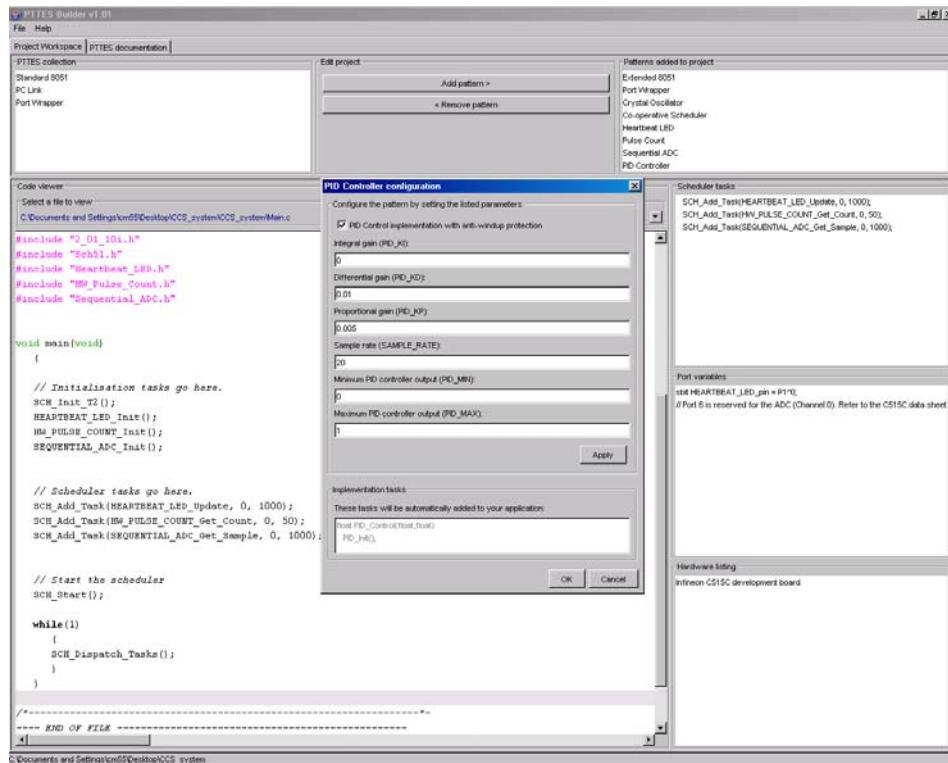


Figure 8: PID CONTROLLER is implemented

Act_Get_Ref_Speed

The task `Act_Get_Ref_Speed` uses the C515C's on-chip ADC to read in the voltage from a potentiometer (i.e. Channel 0 on Port 6, Pin 0). This value is then scaled to represent the set speed. This task is scheduled every 1000 ms.

As shown in Figure 9, this task was implemented using the pattern ONE-SHOT ADC [Pont, 2001].

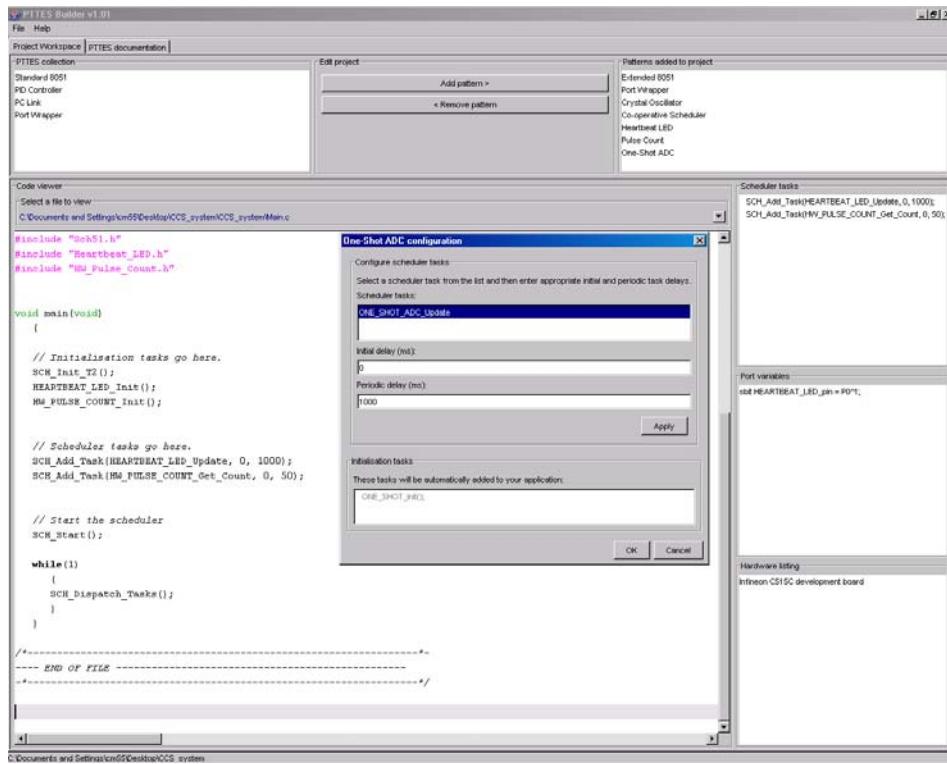


Figure 9: Implementing ONE-SHOT ADC for the configuration of the on board ADC

d) Completing the development

Figure 10 shows the *Main.c* file generated as selected patterns are configured and added to the project.

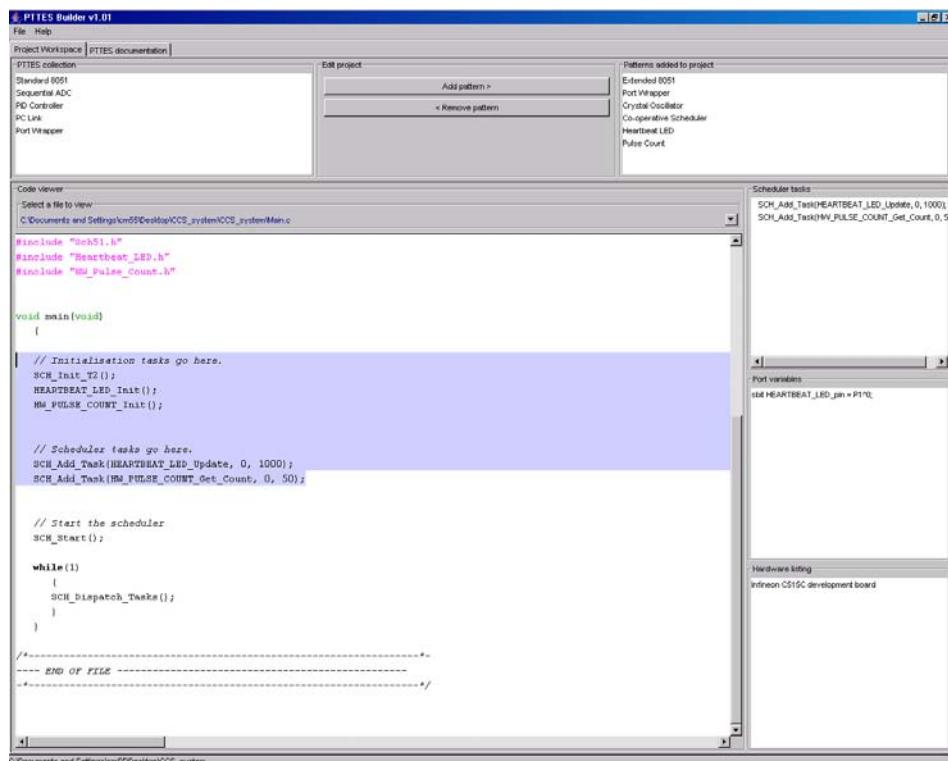


Figure 10: Main.c file generated as patterns are added to the project

In order to view the set speed, the C515C's UART was linked to a PC running "HyperTerminal" (see Figure 1). To implement this behaviour, the pattern PC LINK (RS232) was used [Pont, 2001]. Figure 11 shows PTTES Builder implementing and adding this pattern to the project.

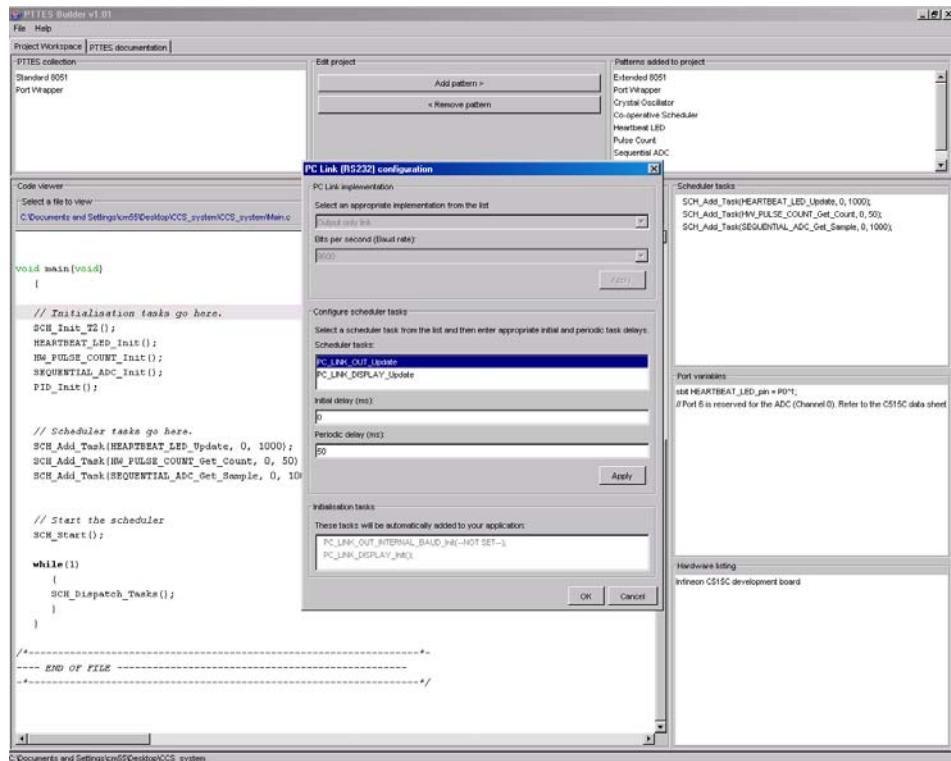


Figure 11: PC LINK configuring the on board UART that is used to link the CCS with a PC at 9600 baud

e) Observations

It should be noted that the present tool does not allow the user to edit the generated code: any editing has to be done in another suitable development environment. In our case we used the Keil IDE[§] to edit and compile the generated code.

Apart from adding patterns to a project, PTTES Builder also allows patterns to be removed from an existing project without affecting an application's architecture. For instance, if the user no longer needs the HEARTBEAT LED, the tool can be used to remove this pattern from the project leaving behind the project's other patterns intact.

[§] Keil Software: <http://www.keil.com/>

f) Results

Figure 12 shows a graph of the recorded set speed (i.e. desired speed) from the CCS over a period of approximately 105 seconds. During this period the set speed was initially 30 m/s (at $t = 1$ s) and then 45 m/s (at $t = 25$ s). The graph shows that the system developed in this case study adapted to the desired speed as required.

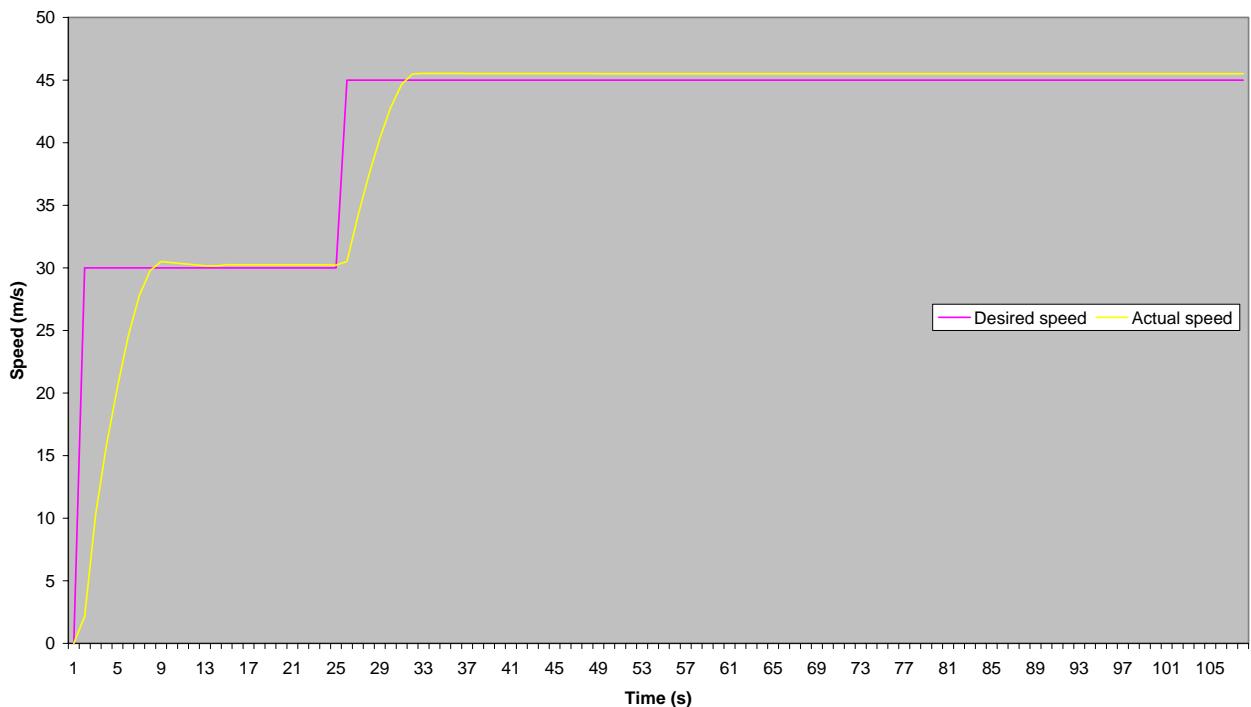


Figure 12 Output speed from the CCS over time (in response to user-initiated changes in set speed)

6. Discussion and conclusions

This paper has provided an overview of a technique for developing reliable embedded systems using a pattern-based code generation tool.

In a related study we have shown that by using this approach to embedded systems development, we can reduce the effort required to develop embedded software [Mwelwa et al., submitted]. This study also suggests that the reliability of the resulting system may improve when applying such an approach.

7. References

- Alencar, P., Cowman, D., Lichtner, K., Lucena, C. and Nova, L., "Tool support for formal design patterns," *University of Waterloo, Waterloo, Ontario, Canada*, Technical Report No. CS-9536, 1996.
- Audsley, N., Bate, I. and Crook-Dawkins, S., "Automatic code generation for airborne systems," *Proceedings of IEEE Aerospace Conference 8th - 15th March*, Big Sky Montana, 2003.
- Ayavoo, D., Pont, M.J., Fang, J., Short, M. and Parker, S., "A 'Hardware-in-the Loop' testbed representing the operation of a cruise-control system in a passenger car," *Proceedings of the 2nd UK Embedded Forum, 20th October*, Birmingham, UK, 2005.
- Beck, K., Crocker, R., Meszaros, G., Coplien, J.O., Dominick, L., Paulisch, F. and Vlissides, J., "Industrial experience with design patterns," *18th International Conference on Software Engineering (ICSE), March 25 - 29*, Berlin, GERMANY, IEEE Computer, 1996.
- Bouyssounouse, B. and Sifakis, J., *Current Design Practice and Needs in Selected Industrial Sectors*, In: Bouyssounouse, B. and Sifakis, J. (Eds.) *Embedded Systems Design: The ARTIST Roadmap for Research and Development*, pp. 15-38, Springer-Verlag GmbH, 2005.
- Budinsky, F., Finnie, M., Vlissides, J. and Yu, P., "Automatic code generation from design patterns," *IBM Systems*, Vol. 35, (2), pp. 151-171, 1996.
- Camposano, R. and Wilberg, J., "Embedded system design," *Design Automation for Embedded Systems*, Vol. 1, (1-2), pp. 5, 1996.
- Cinneide, M.O., *Automated Application of Design Patterns: A Refactoring Approach*, Doctoral Thesis, Department of Computer Science, Trinity College, University of Dublin, Dublin, Ireland, 2000.
- Cline, M., "The pros and cons of adopting and applying design patterns in the real world," *Communications of the ACM*, Vol. 39, (10), pp. 47-59, 1996.
- Cunningham, W. and Beck, K., "Using pattern languages for object-oriented programs," *Presented to the OOPSLA'87 workshop on the Specification and Design for Object-Oriented Programming*, Orlando, Florida, USA, 1987.
- Damasevicius, R., Majauskas, G. and Stuikys, V., "Application of design patterns for hardware design," *Proceedings of the 40th Conference on Design Automation, Session: Design Analysis Techniques*, Anaheim, CA, USA, pp. 48 - 53, 2003.
- Douglass, B.P., *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks and Patterns*, Addison Wesley, 1999.
- EU-eSafety-Working-Group, "eSafety: the use of information and communication technology (ICT) for road safety," *Communication from the Commission to the Council and the European Parliament on Information and Communications Technologies for Safe and Intelligent Vehicles, Brussels*, Technical Report No. COM (2003) 542 Final, 2003.

Florijn, G., Meijers, M. and Winsen, P., "Tool support for object-oriented patterns," *ECOOP'97*, Finland, 1997.

Fowler, M., "Patterns," *IEEE Software*, Vol. 20, (2), pp. 57, 2003.

Gamma, E., Helm, R., Johnson, R. and Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

Key, S., Pont, M. and Edwards, S., *Implementing low -cost TTCS systems using assembly language*, In: Henney, K. and Schutz, D. (Eds.) Proceedings of the 8th European Conference on Pattern Languages of Programs (EuroPLoP 2003), pp. 667-690, Universitätsverlag Konstanz, Irsee, Germany, 2003.

Kopetz, H., *Real-time Systems: Design Principles for Distributed Embedded Applications*, Kluwer Academic, 1997.

Lucredio, D., Alvaro, A., Almeida, E. and Prado, A., "MVCASE tool - working with design patterns," *Proceedings of the 3rd Latin American Conference on Pattern Languages of Programming (SugarLoafPLoP 2003)*, Porto de Galinhas, pp. 261-275, 2003.

Marsh, P., "Models of control," *IEE Electronics Systems and Software*, Vol. 1, (6), pp. 16-19, 2003.

Meijers, M., *Tool Support for Object-Oriented Design Patterns*, Doctoral Thesis, Department of Computer Science, Utrecht University, Utrecht, 1996.

Mullerburg, M., "Software intensive embedded systems," *Information and Software Technology*, Vol. 41, pp. 979-984, 1999.

Mwelwa, C., Ayavoo, D., Pont, M.J. and Ward, D., "Supporting pattern-based development of reliable embedded systems with software tools: Challenges and solutions," submitted.

Mwelwa, C. and Pont, M.J., "Two simple patterns to support the development of reliable embedded systems," *2nd Nordic Conference on Pattern Languages of Programming (VikingPLoP 2003)*, Bergen, Norway, 2003.

Mwelwa, C., Pont, M.J. and Ward, D., "Towards a CASE tool to support the development of reliable embedded systems using design patterns," *Proceedings of the 1st International Workshop on Quality of Service in Component-Based Software Engineering*, Toulouse, France, CEPADUES-EDITIONS, pp. 67-80, 2003.

Mwelwa, C., Pont, M.J. and Ward, D., "Using patterns to support the development and maintenance of software for reliable embedded systems: A case study," *Proceedings of the IEE/ACM Postgraduate Seminar on "Systems-on-Chip" Design, Test and Technology*, Loughborough, UK, IEE, 2004(a).

Mwelwa, C., Pont, M.J. and Ward, D., *Code generation supported by a pattern-based design methodology*, In: Koelmans, A., Bystrov, A. and Pont, M.J. (Eds.) Proceedings of the 1st UK Embedded Forum, pp. 36-55, University of Newcastle upon Tyne, Birmingham, UK, 2004(b).

Noble, J. and Weir, C., *Small Memory Software*, Addison Wesley, 2001.

Ogata, K., *Modern Control Engineering*, Prentice-Hall, 2002.

O'Halloran, C., "Issues for the automatic generation of safety critical software," *The 15th IEEE International Conference on Automated Software Engineering*, Grenoble, France, 2000.

Pont, M.J., "Can patterns increase the reliability of embedded hardware-software co-designs?," *IEE Colloquium on Hardware-Software Co-Design*, Savoy Place, London, IEE Colloquium Digests, 2000.

Pont, M.J., *Designing and implementing reliable embedded systems using patterns*, In: Dyson, P. and Devos, M. (Eds.) Proceedings of the 4th European Conference on Pattern Languages of Programming and Computing (EuroPLoP 1999), Universittsverlag Konstanz, 2000.

Pont, M.J., *Patterns for Time-Triggered Embedded Systems*, Addison-Wesley, 2001.

Pont, M.J. and Banner, M.P., "Designing embedded systems using patterns: A case study," *Journal of Systems and Software*, Vol. 71, (3), pp. 201-213, 2004.

Pont, M.J., Kurian, S., Maaita, A. and Ong, R., "Restructuring a pattern language for reliable embedded systems," *Embedded Systems Laboratory, University of Leicester, Leicester, UK*, Technical Report No. Embedded Systems Laboratory, Technical Report No. 2005-01, 2005.

Pont, M.J., Li, Y., Parikh, C.R. and Wong, C.P., "The design of embedded systems using software patterns," *Proceedings of Condition Monitoring*, Swansea, UK, pp. 221-236, 1999.

Pont, M.J. and Mwelwa, C., "Developing reliable embedded systems using 8051 and ARM processors: Towards a new pattern language," *2nd Nordic Conference on Pattern Languages of Programming (VikingPLoP 2003)*, Bergen, Norway, 2003.

Pont, M.J., Norman, A.J., Mwelwa, C. and Edwards, T., *Prototyping time-triggered embedded systems using PC hardware*, In: Henney, K. and Schutz, D. (Eds.) Proceedings of the 8th European Conference on Pattern Languages of Programs (EuroPLoP 2003), pp. 691-716, Universitsverlag Konstanz, Irsee, Germany, 2003.

Pont, M.J. and Ong, H.L.R., "Using watchdog timers to improve the reliability of TTCS embedded systems," *Proceedings of the 1st Nordic Conference on Pattern Languages of Programs*, pp. 159-200, 2003.

Riehle, D. and Zullighoven, H., "Understanding and using patterns in software development," *Theory and practice of object systems (TAPOS)*, Vol. 2, (1), pp. 3-13, 1996.

Rising, L. (Ed.) *Design Patterns in Communications Software*, Oxford University Press, New York, USA, 2001.

Schatz, B., Hain, T., Houdek, F., Prenninger, W., Rappl, M., Romberg, J., Slotosch, O., Strecker, M. and Wisspeintner, A., "CASE tools for embedded systems," *Technical University of Munich, Munich*, Technical Report No. TUM-I0309, 2003.

Whalen, M.W. and Heimdahl, M.P.E., "On the requirements of high-integrity code generation," *Proceedings of the 4th High Assurance in Systems Engineering Workshop*, Washington DC, 1999.

Mining for Pattern Implementation Examples

Susan Kurian and Michael J. Pont

*Embedded Systems Laboratory,
University of Leicester, University Road, Leicester LE1 7RH, UK*

Abstract

In the pilot study reported in this paper, an attempt was made to understand the challenges involved in developing techniques that can be used to support software maintenance in embedded systems created using design patterns. Specifically, this project explored techniques that can be used to identify particular “Pattern Implementation Examples” (PIEs) in source code that has been created from a pattern library and subsequently undergone manual modifications. The results reported illustrate that this approach has considerable potential.

Acknowledgements

This work is supported by an ORS award to Susan Kurian from the UK Government (Department for Education and Skills) and by the University of Leicester. Work on this paper was completed while Michael J. Pont was on Study Leave from the University of Leicester.

1. Introduction

Most current work on software design patterns was inspired by the work of Christopher Alexander and his colleagues (Alexander et al., 1977; Alexander, 1979). Cunningham and Beck used some of Alexander's techniques as the basis for a small pattern language intended to provide guidance to novice Smalltalk programmers (Cunningham and Beck, 1987). This work was subsequently built upon by Erich Gamma and his colleagues who, in 1994, published an influential book on general-purpose object-oriented software design patterns (Gamma et al., 1995). Since 1994 the development of pattern-based design techniques has become an important area of research in the software engineering community. Gradually, the focus has shifted from the use, assessment and refinement of individual patterns, to the creation of complete pattern languages, in areas including telecommunication systems (see, for example: Rising, 2001) and systems with hardware constraints (Noble and Weir, 2001).

We have previously argued (Pont, 2001; Pont and Banner, 2004) that use of appropriate “design patterns” can assist in the creation of reliable embedded systems. Our research in this area has resulted in the assembly of a collection of more than seventy patterns, most of which are catalogued in the work “Patterns for Time-Triggered Embedded Systems” (Pont, 2001): together these patterns will be referred to in this paper as the “PTTES collection”.

All of our previous work with patterns has focused on software creation rather than on software maintenance. For example, recent studies in our laboratory have shown that it is possible to create code automatically – and extremely efficiently from a pattern-based design (e.g. Mwelwa et al., 2004a; Mwelwa et al., 2004b). However, the costs of software creation are dwarfed – in many projects – by the costs of software maintenance (e.g. Bennett, 2005). This has encouraged some researchers to consider ways in which design patterns might be applied during software maintenance. For example, Keller et al. (1999) have argued that we need to build reverse engineering systems that go beyond capturing the abstract representations of source code. Specifically, Keller et al. suggest that design patterns form the basis of many key elements that contribute to a system’s design, and they argue that extraction of these design elements from code projects should provide a better understanding of the system itself.

In keeping with this argument, the study reported in this paper describes an attempt to develop techniques which can be used to support the maintenance of embedded systems created using patterns. The focus of this study was on techniques which may assist in the identification of

“pattern implementation examples” in source code that has been created from the PTTEs pattern library and subsequently undergone manual modifications.

This paper is organized as follows. Section 2 introduces the concept of a Pattern Implementation Example (PIE). It also describes pattern mining in the context of PIEs. Section 3 explains in brief the problem of extracting software design patterns in embedded systems. Section 4 presents the actual case study. Section 5 presents the results of this project. The final section is a discussion analysing the results.

2. PIEs vs patterns

In early papers, “pattern mining” was a term used to describe the process of extracting design patterns from multiple code examples, in order to build pattern collections (Yoder et al., 1998). The patterns identified would then undergo a process of review and refinement (usually in “PLoP” conferences (Yoder et al., 1998). This review process relied heavily on inputs from domain experts.

Our situation is somewhat different. We have a collection of code from a single project. We assume that this code was originally produced either manually (through use of the PTTEs book), or automatically (through the use of a suitable CASE tool). We also assume that the code will have undergone manual modifications, in order to adapt it – as necessary – for use in a specific project.

Our initial goal (and the one we focus on in this paper) is to identify the patterns which have been used to create the code collection which we are analysing. In these circumstances, we are not “mining” for patterns in the conventional sense. Instead we are trying to find a particular “pattern implementation example” (PIE), which will be a modified version of a code example we already possess.

3. Embedded design patterns vs the “Gang of Four” patterns

As noted in Section 2, there are significant differences between the traditional process of pattern mining and the process of identifying PIEs in source which we wish to perform. Despite these differences, it may seem that many of the tools used for pattern mining could also be used to identify PIEs. However, the use of these tools is not as straightforward as it may appear, at least not for the embedded patterns with which we are concerned in the present study.

Most research in pattern mining has been done on the “Gang-of-Four” patterns: that is the pattern collection compiled by Erich Gamma and colleagues (Gamma *et al.*, 1995). For example, Keller et al. (1999) and Balanyi (2003) describe their reverse-engineering environments (based on C++

source code) that are intended for use with this collection.

When developing embedded systems, C++ is rarely used, and the C language is a much more popular choice (the reasons for this are discussed by Pont, 2003). Unfortunately, despite many advantages in an embedded setting, C lacks the support for object-oriented design that is assumed in previous tools developed for pattern mining.

4. The case study

This study was designed to identify implementations of some scheduler patterns suitable for use in embedded systems. Our aim in this study was to achieve this – as far as possible - using simple “off the shelf” tools.

4.1 The patterns

We aimed to identify three patterns in this pilot study:

- TTC-SL SCHEDULER
- TTC-ISR SCHEDULER
- TTC SCHEDULER

All three patterns are described in Kurian and Pont (2005).

4.2 Identifying the PIEs

The identifying characteristics of each PIE assumed in this study are summarised below.

A TTC-SL SCHEDULER PIE is characterised by:

- Presence of an infinite `while` loop in `main()`, in which tasks to be executed are called directly
- Absence of scheduler files and functions
- Absence of ISR functions (see below)

A TTC-ISR SCHEDULER PIE is characterised by:

- Presence of an infinite `while` loop in `main()`
- Presence of a function which is defined but not explicitly called (assumed to be the ISR associated with a timer).
- Absence of any scheduler files and functions

A TTC SCHEDULER PIE is characterised by:

- Presence of an infinite `while` loop in `main()`
- Presence of a primary scheduler file (with definitions of the core scheduler functions) and – possibly - a secondary scheduler file. (In some cases, a few of scheduler functions are defined in a second C-language source file.)
- Presence of calls to key scheduler functions in the file `main.C`
- Presence of a function which is defined but not explicitly called (assumed to be the scheduler ISR associated with a timer).

In the event that this final analysis program cannot match the source code to one of these lists, it is designated an unidentified architecture.

4.3 Tools used

The C-language source files from these projects were analysed in a systematic manner. Much of the code analysis involved processing the lines of source code as strings of characters. To do this, regular expressions that determined a particular character sequence needed to be constructed: Perl (Wall et al., 2000) was chosen as the implementation language.

For example, a function call can be defined as `/[\w]+\\(.*)\\.*\\;`; which represents a sequence of one or more “word” characters – `'[\w]+'`, followed by a ‘(‘, followed by a sequence of zero or more characters `'.'` which are finally terminated by the ‘)’ character.

Similarly PIE features like the names given to functions can be used to extract information regarding pattern specific constructs.

For example: scheduler functions have names with a “SCH_” prefix in the original pattern library. It is assumed - in this pilot study - that many developers will retain this prefix. A search for the “sch” character sequence then helps us to narrow into any possible scheduler function calls or definitions.

4.4 The data set

The methodology described above was tested on a number of code submissions made by university students.

In most projects, the students were given the original scheduler code in PTTEs, and asked to build

their exercise submissions on this basic framework. The exercises were usually of a two-week duration. The focus of the exercise was (typically) on building a working system, and the original scheduler code was not significantly modified. A total of 87 such projects were subjected to code analysis.

In another case, the code was taken from a larger design project, wherein the students were given the freedom to build their system as they wished. A total of 7 such projects that had C-source files submissions were used in this study.

The complete data set consisted of 94 projects.

5. Results

In Section 4.3, we described techniques adopted to analyse various aspects of the source files in a project. The programs created to test the above techniques were executed as a single batch file.

The results of execution are summarised in Table 1.

PIE used	Total number of samples	Correctly identified	Incorrectly identified	Notes
TTC-SL SCHEDULER	24	20	4	All incorrectly identified as TTC-ISR SCHEDULER
TTC-ISR SCHEDULER	35	34	1	All incorrectly identified as TTC-SL SCHEDULER
TTC SCHEDULER	35	29	6	All incorrectly identified as TTC-ISR SCHEDULER

Table 1: Results of the analysis

Though the percentage identification seems good, there are important factors that need to be taken into consideration while examining these results.

First, the identification process developed involved many code-style assumptions. In spite of this, the number of correctly identified architectures was high. This fact can be attributed to the fact that the original software that was modified by students was written based on strict style rules. This is characteristic of any software that was initially developed using tool support. In this experiment, the style rules were exploited in building the identification system. In a more sophisticated tool, it should be used at least as a validation technique while extracting design patterns.

Second, all of the analysed source code assumed that only the relevant project source code was present in any project folder. In the designed methodology, we analyse all source files in a folder, irrespective of whether they have been actually used in the project. Introducing extra source files can significantly alter correct identification of the micro-architectures involved. This is especially

the case with simple architectures. The relatively high number of wrongly identified TTC-ISR SCHEDULER architectures can be attributed to this flaw. It is important then that a complete extraction tool provides for some way of identifying the source files in a project.

6. Discussion and conclusions

This paper has presented the results from an initial attempt at identifying pattern implementation examples in source code. This represents the first stage in a new programme of work aimed at using design patterns to support the maintenance of reliable embedded systems.

There are many other aspects of design pattern that characterise it and give meaning to its use in a software project. These aspects too need to be used while extracting design patterns from source code. Future research aims at using these other aspects either to extract patterns or to validate the extraction process.

References

- Alexander, C. (1979) "The Timeless Way of Building", Oxford University Press, NY.
- Alexander, C., Ishikawa, S., Silverstein, M. with Jacobson, M. Fisksdahl-King, I., Angel, S. (1977) "A pattern language", Oxford University Press, NY.
- Balanyi, Z.; Ferenc, R.(2003) "Mining design patterns from C++ source code" Source:Proceedings International Conference on Software Maintenance ICSM 2003 Publisher:IEEE Comput. Soc
- Bennett, K., Gold, N. and Mohan, A. (2005) "Cut the biggest IT cost", The Computer Bulletin, 47(1): 20-21.
- Cunningham, W. and Beck, K. (1987) "Using pattern languages for object-oriented programs", Proceedings of OOPSLA'87, Orlando, Florida.
- Dominus, Mark-Jason (1998) "Perl: Not just for Web programming" Source: IEEE Software, v 15, n 1, Jan-Feb, 1998, p 69-74 Publisher: IEEE Comp Soc
- Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995) "Design patterns: Elements of reusable object-oriented software", Addison-Wesley, Reading, MA.
- Keller, Rudolf K.; Schauer, Reinhard; Robitaille, Sebastien; Page, Patrick (1999) "Pattern-based reverse-engineering of design components" Source: Proceedings - International Conference on Software Engineering, 1999, p 226-235, May 16-May 22 1999, Los Angeles, CA, USA Publisher: IEEE
- Kurian, S. and Pont, M.J. (2005) "Building reliable embedded systems using Abstract Patterns, Patterns, and Pattern Implementation Examples", paper presented at the 2nd UK Embedded Forum (Birmingham, UK, October 2005).
- Larry Wall, Tom Christiansen, and Randal L. Shwartz; with Stephen Potter.(2000) "Programming Perl", Cambridge: O'Reilly, 2000.
- Mwelwa, C., Pont, M.J. and Ward, D. (2004a) "Using patterns to support the development and maintenance of software for reliable embedded systems: A case study", Proceedings of the IEE / ACM Postgraduate Seminar on "System-On-Chip Design, Test and Technology",

Loughborough, UK, 15 September 2004. Published by IEE. ISBN: 0 86341 460 5 (ISSN: 0537-9989)

Mwelwa, C., Pont, M.J. and Ward, D. (2004b) "Code generation supported by a pattern-based design methodology". In: Koelmans, A., Bystrov, A. and Pont, M.J. (Eds.) Proceedings of the UK Embedded Forum 2004 (Birmingham, UK, October 2004). Published by University of Newcastle.

Noble, J. and Weir, C. (2001) "Small memory software", Addison Wesley, 2001

Ong, H.L.R and Pont, M.J. (2002) "The impact of instruction pointer corruption on program flow: a computational modelling study", Microprocessors and Microsystems, 25: 409-419

Pont, M.J. (2001) "Patterns for time-triggered embedded systems: Building reliable applications with the 8051 family of microcontrollers", ACM Press / Addison-Wesley, UK

Pont, M.J. (2003) "An object-oriented approach to software development for embedded systems implemented using C", Transactions of the Institute of Measurement and Control 25(3): 217-238.

Pont, M.J. and Banner, M.P. (2004) "Designing embedded systems using patterns: A case study", Journal of Systems and Software, 71(3): 201-213.

Pont, M.J. and Ong, H.L.R. (2003) "Using watchdog timers to improve the reliability of TTCS embedded systems", in Hruby, P. and Soessen, K. E. [Eds.] Proceedings of the First Nordic Conference on Pattern Languages of Programs, September, 2002 ("VikingPloP 2002"), pp.159-200. Published by Microsoft Business Solutions. ISBN: 87-7849-769-8.

Rising, L., [Ed.] (2001) "Design patterns in communications software", Oxford University Press.

Shaw, A.C. (2001) "Real-time systems and software", John Wiley & Sons, New York.

Yoder, J. W.; Foote, B.; Riehle, D.; Tilman, M.; Metadata and active objectmodels, in: Workshop Results Submission OOPSLA'98 Addendum, 1998.

On-chip Sub-Picosecond Phase Alignment

C. D'Alessandro, K. T. Gardiner, D. J. Kinniment, A. V. Yakovlev

October 26, 2005

Abstract

A novel approach to the problem of on-chip phase alignment of two signal paths consists in modifying the dynamic characteristics of inverter buffers placed along the lines. A Mutual Exclusion element (ME) is employed as a phase detector and the whole system is implemented in the fashion of an easy-to-implement Delay Locked Loop (DLL) with minimum use of analogue circuitry. The system proposed allows the two signal paths to be aligned on average to within a sub-picosecond time range. Simulation results are presented and a possible on-chip implementation is described.

1 Introduction

Reduction in chip size results in higher variability of delays between separate parts of the same system, due to the increased dependance on the process parameters. This unpredictability of delays can cause problems if the design requires two paths to be precisely aligned to each other; the increase in speed of circuits is reflected in shorter alignment ranges and reduced jitter rejection.

The phase alignment of two paths is implemented using variable delay lines on the two paths, where additional delay is introduced on either line until the two paths are exactly matched in delay. The requirement for exactly synchronized lines arises in various research areas; we focus in particular on the characterization of metastability and the Time-to-Digital conversion problem [2]. Modern approaches attempt to measure delays of the order of picoseconds and emerging technologies let the research community foresee the need for even shorter time measurements. Our system aids these activities by minimising the time difference between two signals introduced by jitter on the transmission lines. As an example, consider the case of the Time Difference Amplifier [1]; as the name implies, this device amplifies the time difference between two signals, but if the time difference is beyond its range, the device enters “saturation”, producing wrong results. This could happen if there is a constant time difference introduced by process variations or increases in temperature. Therefore the need arises for a device which would minimise this additional delay and allow the time measurement circuitry to detect only the useful measure.

Various techniques are employed to measure the jitter of signals and to measure the time difference between two signals. Popular approaches include the Delay Locked Loop [8] (DLL) and the Vernier Delay Line [5] (VDL). The DLL, very similar to a Phase Locked Loop (PLL), measures the phase of two signals and modifies the delay on the lines until the two signals are in phase. The VDL uses separate delay lines with different delay values; the propagation of the signals through the lines is continuously monitored using arbiters at each delay element and, from the time of arrival of the two signals to each arbiter, the control logic can infer an estimation of the time difference between the signals.

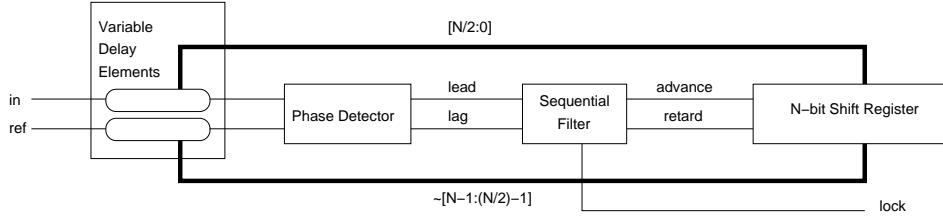


Figure 1: Block Diagram of overall system

Both approaches rely strongly on successful matching of the delay lines and, in the case of DLL using Dynamic Voltage Scaling (DVS) or Current-Starved Inverters (CSI), on the generation of stable voltage references. Although this is possible at the expense of design time and restrictions on layout and production, it can become unacceptable if the device needs to be repeated in various parts of a host circuit. Our approach consists in a DLL which uses a modified DVS approach, more robust and more easily replicated.

2 Overall System

A block diagram of the system is shown in Figure 1. The system comprises of a phase detector based upon a Mutual Exclusion element (ME) [7] and a control logic to perform event detection; a filter to determine the response of the device; a control logic for the delay lines and two variable delay lines controlled by the loop. The system uses a sequential filter instead of the popular charge-pump configuration, in order to avoid analogue circuitry. The next sub-sections will describe in more details the various parts of the system.

2.1 Phase Detector

The phase detector (PD) is build around an ME, whose circuit diagram and Signal Transition Graph (STG) are depicted in Figures 2 a) and b) [4]. Normally this device is used for arbitration of requests to access a resource by two competing blocks. Therefore, for the sake of consistency with existing literature, the inputs are “requests” and the outputs are “grants”.

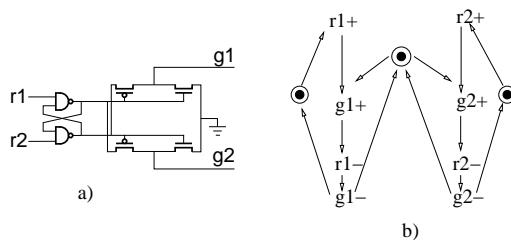


Figure 2: Mutual Exclusion element. a) Hardware implementation; b) STG

The “grant” signal corresponding to the “request” which first is asserted at the ME will be asserted. In normal conditions, the other request will remain “pending” (asserted) until the corresponding “grant” has been issued, as described by the STG in Figure 2 b). Note that if the request does not wait for the grant to be given, the

ME could not have enough time for the signal to be switched between the two grants. This could cause hazards at the output of the ME: if the two requests are “tied” to the same input with delay mismatches between the request generation and the input of the ME, the falling transitions of these inputs could be so close that the ME will not have enough time to correctly switch between the two requests. This could generate hazards as the outputs might or might not, according to the input delay, follow the normal behaviour of the ME described in the STG in Figure 2 b). To perform event detection, the ME is enclosed in a wrapper which produces an output when both signals have arrived, regardless of their behaviour. The implementation of this wrapper ensures that the inputs are not unbalanced by the introduction of gates at the input of the ME. A simple implementation of the event detector would have consisted of two flip-flops at the outputs of the ME clocked by the ANDing of the two inputs. However, this implementation introduces an imbalance due to the structure of AND gates; this imbalance could be resolved using an additional AND gate to restore the balance, but this approach could result in unacceptable results. Instead, event detection is performed at the output of the ME, where the matching requirements are less stringent.

The output of the PD indicates that one of the signals *leads* or *lags* the other. This type of PD is described in the literature of PLLs as a *lead/lag* phase detector and is used to compare a signal to a reference, usually the output of the Voltage Controlled Oscillator (VCO) of a PLL. To be consistent with the literature, one of the inputs of the PD will be the INPUT signal, while the other will be the REFERENCE signal. “Lead” and “lag” will therefore refer to the INPUT signal leading/lagging the REFERENCE.

2.2 Sequential Filter

As the PD produces a binary quantized output, the system is not able to indicate whether the two signals are “in lock” without additional logic. The implementation relies on a *sequential filter* to perform integration of the PD output and indicate whether the inputs are synchronized. This type of filter, reported and briefly described in [6], is slightly different from usual digital filters in that the output is not a linear combination of a number of previous inputs. Rather, the filter observes the input for a length of time and produces an output when a certain confidence limit on the inputs is established. Two examples are found in [3], an *N-before-M* filter and a *Random Walk* filter. The block diagrams of Figures 3 a) and b) illustrate the function of these filters.

An *N-before-M* filter is built using two counters which saturate at N and an additional counter whose saturation point is M . The two N counters integrate the occurrence of either a “lead” or “lag” output of the PD, while the M counter integrates the occurrence of any event. If either of the N counters is filled before the M counter, an *advance* or *retard* signal is generated, indicating that the INPUT either (respectively) lags or leads the REFERENCE (hence the name *N-before-M*). The counters are reset to zero when either of the counters is full. The *Random Walk* filter requires a simple up/down counter from 0 to $2N$. The counter is initialized at N and, upon reaching either 0 or $2N$, an advance/retard signal is generated and the counter is reset to N .

Analysis of sequential filters is based upon statistical methods, well described in literature and available to designers and a description can be found in [3]. We simply note here that, in the case of an *N-before-M* filter, the M counter can be considered an *event* counter, whilst the two N counters are selectively counting one of two possible events. If the occurrence of one event is dominating beyond a probabilistic limit, the inputs are violating the distribution expected for “in lock” behaviour and an update of the delay lines status is required. Similar reasoning can

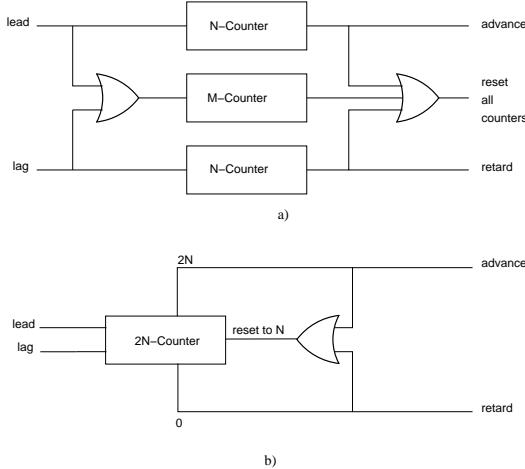


Figure 3: Examples of Sequential Filters; a) *N-before-M* filter, b) *Random Walk* filter

be followed for the *Random Walk* filter. Also note that N and M for the *N-before-M* filter must respect the inequality $N < M < 2N$.

The equations relating N and M or $2N$ to the probability functions are summarized in [3]. Define U_1 and U_{-1} as, respectively, the probabilities of a retard or an advance command; u_1 and u_{-1} as the probabilities of a phase lead or lag detected by the PD and T as the expected time required to produce an output. Then, for an *N-before-M* filter:

$$U_1 = \frac{\alpha}{\alpha+\beta}; U_{-1} = \frac{\beta}{\alpha+\beta}$$

$$\alpha = \sum_{i=N}^M \binom{i-1}{N-1} u_1^N \cdot u_{-1}^{i-N}; \beta = \sum_{i=N}^M \binom{i-1}{N-1} u_{-1}^N \cdot u_1^{i-N}$$

$$T = \frac{(1 - \alpha - \beta) \cdot M + \sum_{i=N}^M \binom{i-1}{N-1} [u_1^N \cdot u_{-1}^{i-N} + u_{-1}^N \cdot u_1^{i-N}] \cdot i}{\alpha + \beta}$$

Similar equations describe the *Random Walk* filter. This statistical response is advantageous in this case as the input to the phase detector is subject to jitter and noise.

The implementation employs an *N-before-M* filter and is such that the N and M values can be loaded in the counters at any time, in order to modify the dynamic behaviour of the loop.

2.3 Control Logic

The delay lines are controlled by a bi-directional shift register. One half of the register controls the delay line of the INPUT, while the second half the delay line of the REFERENCE; the lines controlling the delay elements of the REFERENCE are inverted. Consider the case where the INPUT is consistently leading the REFERENCE; in this case, the INPUT signal needs to be retarded and the shift register will “move” according to the design-dependent rules¹. When the maximum number

¹ The actual rule will depend on the design of the variable delay elements

of steps for the INPUT have been used, the REFERENCE will need to be advanced, and therefore the delay of the REFERENCE line needs to be reduced. Hence, in order to control the two delay lines using only one register the control lines for the two signals must be controlled in opposite ways.

2.4 Variable Delay Elements

The crucial part of the system is the variable delay element (VDE) employed on the input lines. In general, DVS, VDL or CSI techniques can be employed. Our approach employs a simple DVS technique which employs including an array of transistors above (or below) an inverter, as shown in Figure 4 (a). Note that only the falling edge of the input in Figure 4 (a) will be affected by the variable delay. This asymmetry caused by using a single array of transistors is introduced in order to simplify the PD design, as a single ME can only recognise rising edges of the inputs. It can be corrected by either introducing a matching array below the inverter or by adding an identical array of transistors to the normal inverter shown.

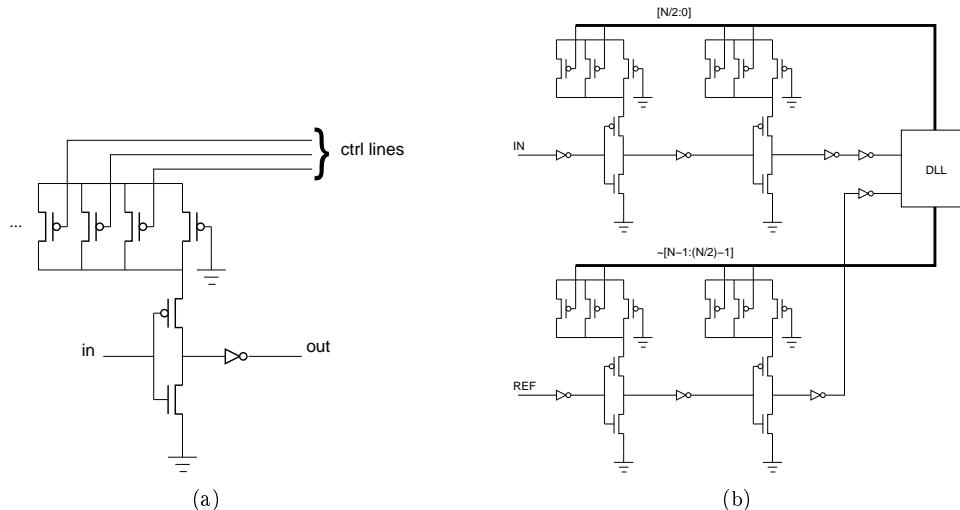


Figure 4: (a) Variable Delay Element based on simple DVS; (b) Variable Delay Elements configuration

The transistors are activated by the control lines; one transistor is always on in order to provide a minimum/maximum (according to the design) delay for the element (base transistor). Consider the case shown in Figure 4 (a): switching one transistor on, the equivalent resistance of the array of transistors will diminish, reducing the delay of the element. However, using several transistors of same size to be switched on or off causes non-linearity in the available delays, in a manner resembling parallel resistors. This can be alleviated primarily by using a wide base transistor and relatively smaller ones to be controlled. The width of the controllable transistors must be such that every contribution of the transistors will modify the delay as needed.

This approach is successful for a small number of transistors or when linearity is not a primary issue; if linearity becomes important, other techniques must be used. One approach consists in scaling the width of the controllable transistors; however, process variations could affect the linearity of the system, as the variations required are small compared to the size of the transistor themselves. Our solution employs

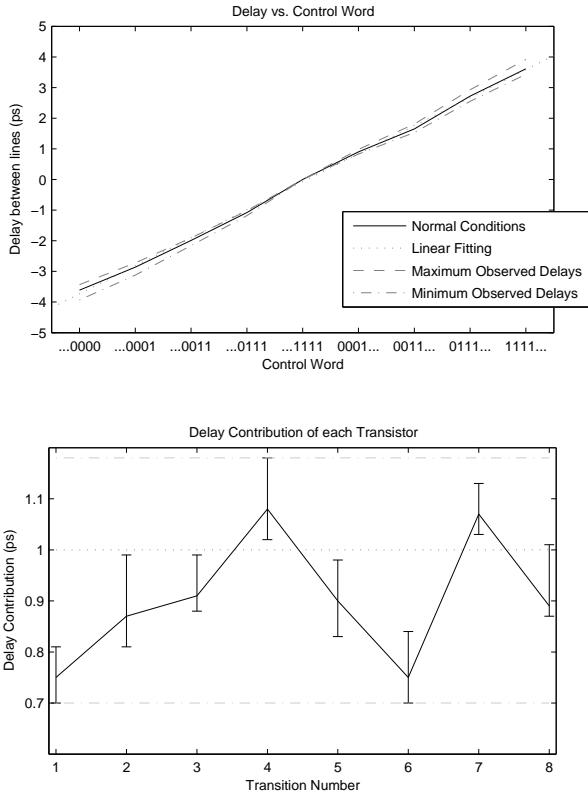


Figure 5: Delay between the two lines vs control word (top) and contribution of each transistor between control words (bottom). For the right-hand figure: transition 1 = 00000000 to 00000001; transition 2 = 00000001 to 00000011;...

several delay elements on the same line in order to improve linearity, as in Figure 4 (b). Note that the capacitative load introduced by the stage following the VDEs changes the delay contribution of each transistor; therefore, every VDE includes a buffer after the modified inverter to fix the capacitative load. Also note that, for the reasons explained above, only the rising edge of the input will be affected.

3 Results

The layout of a chip is under development and a chip should be produced in the summer of 2005. The results reported are based upon simulations performed using Cadence (c) on a $0.35\mu\text{m}$ process parameters. The variable delay lines consist of three-transistors arrays of P-type transistors above inverters. The control register is 8-bit wide and therefore there are four VDEs, two on the INPUT path and two on the REFERENCE path. The circuitry was designed so that the delay contribution by each transistor would be $1 \pm 0.3\text{ps}$.

Figure 5 shows the effect of the VDEs onto the two signal lines when tied together to the same clock; the control register is moved through all the possible values. Note that in the top figure, the x-axis corresponds to the thermometer code sequence, while in the bottom figure it indicates the transition between one control word and the next. The curves in the top figure indicate the behaviour at room temperature with voltage supply at 3.3V (“Normal Condition”), a linear fit of this

curve and the effects of combined temperature and voltage supply variation. The bottom figure indicates the contribution to the delay by each transistor switching, with the error bars referring to the effect on this contribution due to the same variations specified above. Note the high linearity of the results indicated in the top figure. Also note that the maximum difference in contribution, considering errors introduced by temperature and voltage supply variations, does not exceed the specified range: the absolute error with respect to the nominal contribution of 1ps has an upper bound is 0.18ps and a lower bound is 0.3ps. Different results could be obtained varying the transistors width and the capacitative load of the VDEs.

4 Conclusions

A novel on-chip phase alignment method has been presented. The implementation is attractive thanks to the limited analogue circuitry involved, the ease of design and the possibility to replicate the circuit in various parts of a host system, as the implementation does not depend strongly on process parameters matching. The results show good linearity and resilience in case of variations in temperature and voltage supply. As well as simulations, measurements on experimental DLLs built from individual VDEs and phase comparators have shown good controllability down to the picosecond level. Further studies are under way, in order to employ this approach with existing systems for the applications described in the Introduction.

References

- [1] A.M. Abas, A. Bystrov, D.J. Kinniment, O.V. Maeovsky, G. Russell, and A.V. Yakovlev. Time difference amplifier. *Electronics Letters*, 38(23):1437–1438, November 2002.
- [2] A. Bystrov, D.J. Kinniment, G. Russell, O.V. Maeovsky, and A.V. Yakovlev. On-chip structures for timing measurement and test. *Microprocessors and Microsystems*, 27:473–483, October 2003.
- [3] J.R. Cessna and D.M. Levy. Phase noise and transient times for a binary quantized digital phase-locked loop in white gaussian noise. *IEEE Transaction on Communications*, COM-20(2):94–104, April 1972.
- [4] J. Cortadella, A.V. Yakovlev, L. Lavagno, and P. Vanbeekbergen. Designing asynchronous circuits from behavioral specifications with internal conflicts. In *Proceedings. ASYNC'94*, pages 106–115. IEEE CS Press, November 1994.
- [5] Piotr Dudek and S. Szczepański. A high-resolution CMOS Time-to-Digital converter utilizing a Vernier delay line. *IEEE Transaction on Solid-State Circuits*, 35(2):240–247, February 2000.
- [6] W.C. Lindsey and C.M. Chie. A survey of digital phase-locked loops. *Proceedings of the IEEE*, 69(4):410–431, April 1981.
- [7] C. Molnar and I. Jones. Simple circuits that work for complicated reasons. In *Proceedings. Sixth International Symposium on Asynchronous Circuits and Systems*, volume 1. IEEE CS, April 2000.
- [8] M. Mota and J. Christiansen. A four-channel self-calibrating high-resolution time to digital converter. In *1998 IEEE International Conference on Electronics, Circuits and Systems*, volume 1, pages 409–412. IEEE, 1998.

Automatic conversion from 'single processor' to 'multi-processor' software architectures for embedded control systems

Peter J. Vidler and Michael J. Pont

*Embedded Systems Laboratory,
University of Leicester, University Road, Leicester LE1 7RH, UK*

Abstract

In this paper we consider the problem of converting between systems in which multiple tasks are executed on a single node and the equivalent design in which multiple nodes each execute a single task. The work presented here represents the first stage in a larger study in which we aim to develop tools which will automate such a conversion process. In the systems discussed in the present paper, the single-processor designs employ a time-triggered co-operative scheduler (or “cyclic executive”) and the multi-processor designs employ a “domino” scheduler. We describe the design of a prototype conversion tool. The results obtained when applying this tool in a case study are then reported.

Acknowledgements

This project is supported by the UK Government (EPSRC, DTA award). Work on this paper was completed while MJP was on Study Leave from the University of Leicester.

1. Introduction

Time triggered (Kopetz, 1997) embedded systems are those in which all tasks are executed periodically, usually under the control of a single hardware timer. Such systems are widely recognised as providing benefits to both reliability and safety (Allworth, 1981; MISRA, 1994; Nissanke, 1997; Pont, 2001; Storey, 1996) in some of the more safety-critical applications (such as those used in the automotive and aerospace industries). Time triggered systems that run multiple tasks require some form of scheduler to determine when each task should be executed. This is usually one of three main types: pre-emptive (Audsley, *et al.*, 1995; Moitra, 1986), co-operative (“non pre-emptive”) (Locke, 1992; Pont, 2001) or hybrid (Maaita and Pont, 2005; Pont, 2001).

In a co-operative scheduler, each task runs until completion and no task may interrupt (pre-empt) another. This provides the advantage of simplicity, amongst others (Bate, 2000), as the developer need not be concerned – for example – with issues that can arise from concurrent data access in a pre-emptive system. As such there is no need for locking mechanisms and consequently no chance of encountering priority inversion (Jie and Lee, 2003) and other such complications.

Such simplicity comes at a price: because no task can be interrupted, tasks that gather input cannot run while another task is being executed. Therefore if the system must be highly responsive to changes in input, it might not be possible to build it using a purely co-operative scheduler.

If the system does not meet the requirements of a co-operative scheduler, then a designer may opt to use either a hybrid or a pre-emptive scheduler (Bate, 1998; Locke, 1992). Unfortunately, neither of these solutions has the simplicity of the co-operative system nor are they immune to the problems of concurrent data access. Alternatively a multiple processor system can be employed, ideally using a co-operative scheduler on each processor. As long as careful thought is given to the necessary communication between processors, such a system can have most of the advantages of a co-operative scheduler at cost of increased complexity. This complexity can be further reduced by using a domino scheduler (Pont, 2001).

Domino schedulers work in systems where the data flow is all in one direction (see Figure 1) and the various tasks are all of very similar duration. For example, if Task A is collecting input, Task B is processing this data and Task C is outputting the data, then we can put each task on a separate processor and have them all running at the same time, with data passing from Task A to Task B (and on to Task C) at the beginning of every task cycle. Although this actually takes longer (in

terms of the time from input to output) than the single processor system (due to the extra time to communicate between processors), the resulting system can be much more responsive to changes in input. This is because Task A will be collecting input while the other two tasks are executing, as opposed to the co-operative single-processor system where it would have to wait for the other two tasks to complete before executing again. We therefore gain the responsiveness that can be lacking in the original co-operative system, without adding too much complexity.

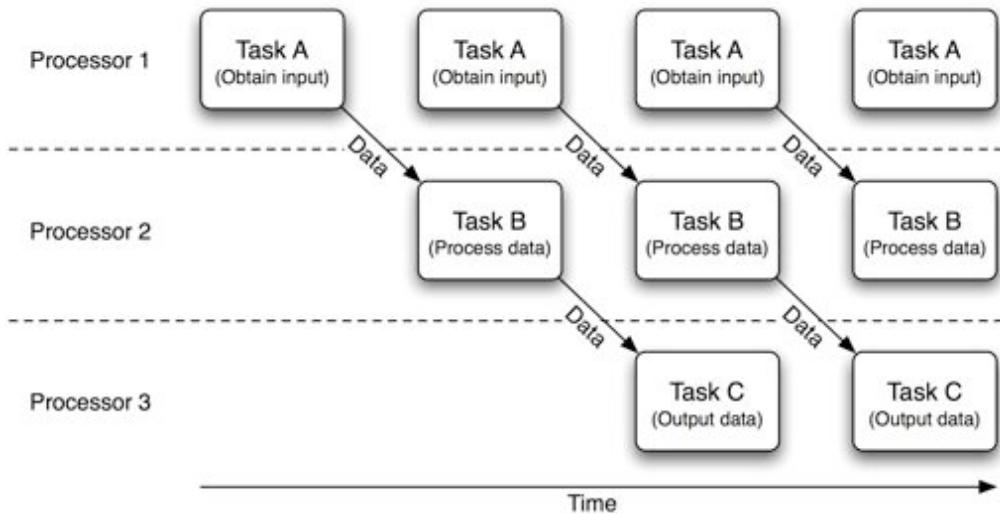


Figure 1: The communication between tasks on separate processors in a domino scheduler.

The focus of this paper is on the automatic conversion from source code written for a single-processor co-operatively scheduled architecture into a multiple-processor domino scheduled system. This will be discussed in greater detail in Section 3, while Section 2 contains details of previous work in this area. Sections 4 gives the details of a case study used to test the automatic conversion. Finally, in Section 5 we give some concluding remarks based on the results of the case study presented in the earlier sections.

2. Previous Work

There has been a great deal of previous work in the field of automatic parallelisation. However, most of this relates to Instruction Level Parallelism (ILP), which requires significant support from both software tools and hardware in order to be truly effective (Aditya, *et al.*, 2000; Aditya and Rau, 1999; Pillai and Jacome, 2003; Rajagopalan, *et al.*, 2001) and is also limited to the availability of ILP within the system and source code itself. The approach we are taking in this paper is that of task level parallelism, or splitting individual tasks off onto separate processors (or at least processor cores); as such, most techniques used in employing ILP are not highly relevant.

In terms of automatic code conversion tools, there are a few available. These include EXPRESSION (Halambi, *et al.*, 1999), Giotto (Henzinger, *et al.*, 2001; Jie and Lee, 2003), SEA and CHaRy (Rust, *et al.*, 2000) to name a few. Most of these tools work either by providing the tool directly with additional details about the system (usually through an external file written in a custom language (Halambi, *et al.*, 1999; Henzinger, *et al.*, 2001)), or revolve around high level (sometimes visual (Rust, *et al.*, 2000)) programming tools that are used to aid the programmer in modifying the system (as opposed to being truly automatic).

There are other tools that generate source code for embedded systems. These typically generate code from high-level models, usually in UML (Mellor, 1999), but can also generate code from representations such as patterns (Mwelwa, *et al.*, 2004). There is a degree of similarity between such systems and our own; they tend to employ a conversion from a higher-level representation into source code, while we convert one source code architecture directly into another.

3. Automatic Code Conversion

In this section we describe the approach used in this paper for automatic code conversion.

a) Where do we convert?

Before considering the details of the approach, we first need to identify where – in the program build process – we should perform the conversion. Figure 2 shows a typical build process, from source file inputs to binary output. Any one of the four major steps in Figure 2 could be used to perform the conversion; indeed, the usual approach when employing ILP is to do so in the compiler itself. The disadvantage of this approach is that the conversion process becomes highly dependent on the target platform. The converter would either have to be built in to the compiler for every desired target, or a new highly retargetable compiler would have to be created that could also handle the conversion.

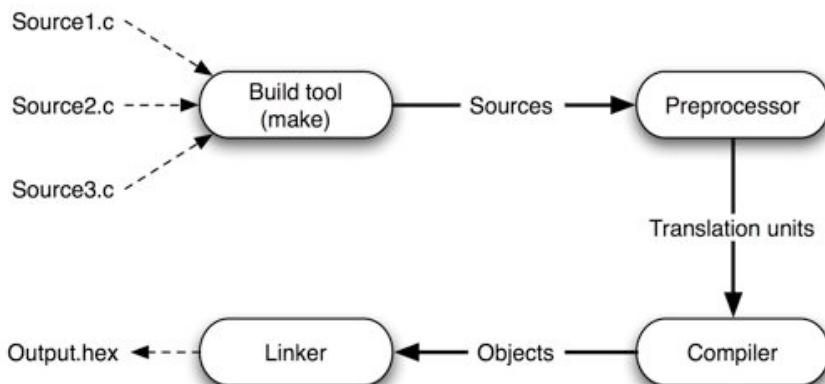


Figure 2: Steps of a fairly typical (single-processor target) compilation process.

For the developer of embedded systems, highly retargetable compilers are rarely employed, because of the reduction in code quality that is almost always the result of such flexibility (Hohenauer, *et al.*, 2004): this is often a concern, since CPU and memory resources can be very restricted in embedded designs. However, since companies make use of a range of different hardware platforms and, consequently, must maintain a number of different compilers, the prospect of including the converter in the compiler is also unattractive (as this is likely to add significantly to compiler costs).

If we rule out changes to the compiler, then our options are to change the pre-processor, the linker, or the build tool itself. As we are trying to achieve parallelism at the task level, we need access to all the tasks at the time of conversion. This is not possible with the pre-processor (which operates on a file-by-file basis): this limits our options to either the build tool or linker phases. Altering the linker has many of the same disadvantages as altering the compiler, which leads us to consider the build tool itself.

b) The conversion process

Having decided (largely through a process of elimination) to consider executing the conversion at the build tool phase, we need to address the conversion process itself.

In order to convert from a single processor, co-operatively scheduled system to a domino-scheduled architecture, we need to be able to identify all the tasks in the program and the communication between them.

In co-operative systems, this communication is usually in the form of global variable accesses. These are perfectly safe to use in such systems because there is no possibility of several tasks accessing the same variable at the same time (the tasks simply cannot execute at the same time).

If a system is suitable for conversion to a domino-scheduled architecture, then all data will flow between tasks in a single direction. In real terms this means that if (say) Task A writes to a global variable and Task B only reads from it, then we can say there will be data flowing from Task A to Task B. Knowing this, we can parse the source code for each task and determine which variables each task uses and whether read-only or read-write access is employed. We can then extract each task into a separate translation unit, add extra code to take care of the communication (replacing the global variable in each case) and compile each unit separately.

Figure 3 shows the new compilation process that results from these proposals.

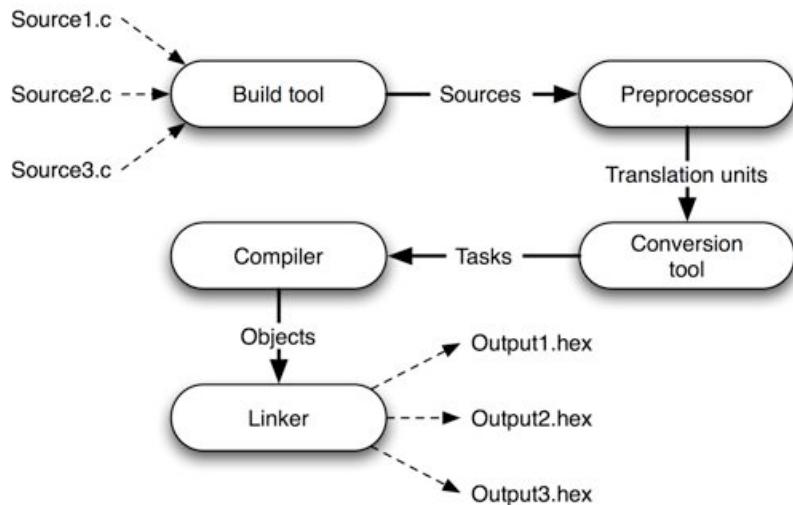


Figure 3: Steps of the new (multi-processor) compilation process.

Notice that the process in Figure 3 is slightly different to the one described earlier. Instead of directly replacing the build tool, we have used the build tool to pass each source file through the preprocessor before being passed on to the conversion tool. This saves us the effort of preprocessing the source files from within our conversion tool; it also allows us to take advantage of the interface between preprocessor and compiler (specifically the “#line” compiler directive) to help the compiler give correct line numbers for error messages.

Notice also that the communication between the conversion tool and the compiler is in “tasks”. These are actually separate translation units for each task, incorporating parts of one or more of the translation units that the preprocessor outputs. Likewise, each of the three output files are made up of compiled code from one or more of the three inputs (along with some extra scheduler code in each).

Finally, in order to make the code conversion as simple as possible in this first version, each task will be split into a separate processor. Therefore the output of the conversion process will be one translation unit for each task in the input source files.

4. Case Study

We used a simple case study to evaluate the first version of the conversion tool. The case study is described in this section.

a) Background

We have chosen a simple cruise control system (Ayavoo, *et al.*, 2005, 2004) as the main case study for this paper. This system was chosen because it can be implemented using three relatively simple tasks, which communicate through global variables of varying sizes. This allows us to test the automatically generated communication to ensure that it works even when different amounts of information are being sent at each stage.

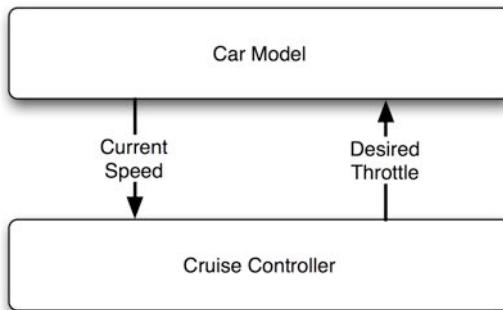


Figure 4: The Cruise Control System (CCS) used for the case study.

The cruise control system consists of two basic parts (shown in Figure 4), the car model and the cruise controller itself. The car model outputs the current speed of the car as a Pulse Rate Modulated (PRM) signal and receives as input the new throttle setting from the cruise controller. Therefore the cruise controller will have to receive input as a PRM signal, convert it to a meaningful value, carry out some form of PID control to arrive at a desired throttle setting and finally output the throttle to the car model.

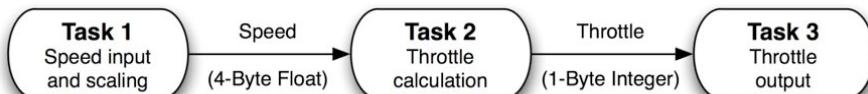


Figure 5: The three tasks of the CCS and the communication between them.

In choosing how to separate this system into tasks, we must remember that the intent here is to test the communication between processors. For this reason, we have chosen to partition the system into three separate tasks: current speed input (including scaling), throttle computation and throttle output. Figure 5 shows the three tasks and the data that must be passed between them.

b) Architecture

Although the design of the conversion tool is flexible enough to support different architectures, for the purposes of this paper we have settled on using standard 8051 processors daisy chained together with a parallel connection for communication (shown in Figure 6). The first processor in the chain will be the ‘master’ and will be the only processor to use a timer interrupt, which it will use to keep

track of tick intervals. Each subsequent processor will receive ticks via an external interrupt from its predecessor in the chain. This form of shared-clock architecture allows us to keep the processors synchronised, while also using only one interrupt per microcontroller (Pont, 2001).

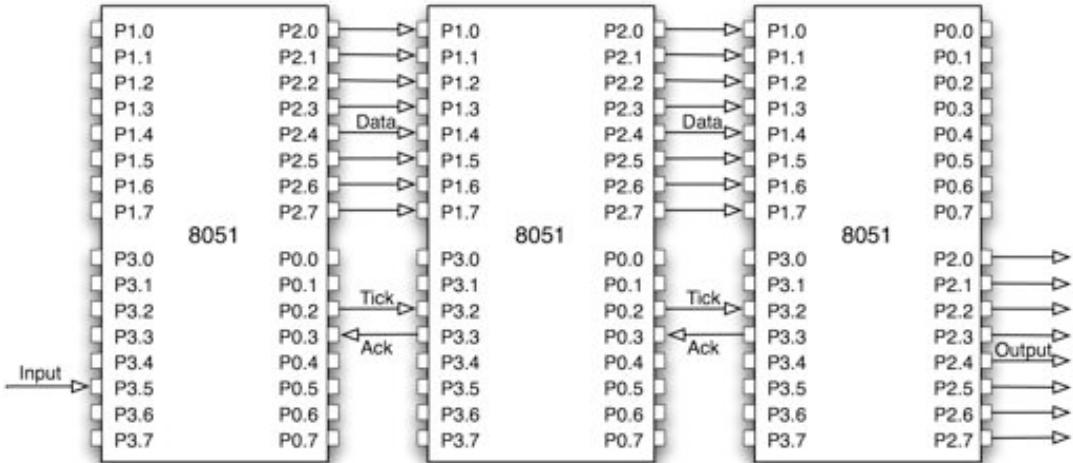


Figure 6: The multi-processor CCS architecture (some ports rearranged for clarity).

It is possible to implement this system using a single shared bus architecture, but that would suffer from the tick and acknowledge messages raising interrupts while the task is executing. This would prevent our system from being strictly co-operative on each processor, losing one of the biggest advantages of a domino scheduler. It is also possible to daisy chain the processors using multiple UARTs or CAN controllers (or other serial interface), but a full study of the implications of these approaches is beyond the scope of this paper.

When there is more than one byte of data to be transmitted in one go, the system first sets the output port to the first byte of data. It then sets the ‘Tick’ pin in order to raise the interrupt on the next processor, which receives the first byte and disables the interrupt so that further data can be received. The rest of the data is then sent by the first processor, which sends a new byte whenever the second processor changes the ‘Ack’ pin (and sets the data port to the inverse of the message, as a crude means of error checking).

During this communication, each processor must (at some point) wait for the other processor to send a message. For the purposes of this paper, each processor uses a watchdog timer to ensure that these delays do not become infinite. This solution is rather severe, causing the entire system and all three processors to reset in the case of a problem occurring; in a real system, a timeout in the wait loops and some form of more complete error handling would be advisable. It is quite possible to implement such a system and still use the current conversion tool.

c) Results

The purpose of this case study was to show that the automatic conversion process works correctly. In order to have some way of testing this, we had each of the cruise controllers deal with a sudden increase in the desired speed setting (from 30 mph to 60 mph, 30 seconds after the start of the simulation).

Figure 7 shows the results measured for the single processor cruise controller.

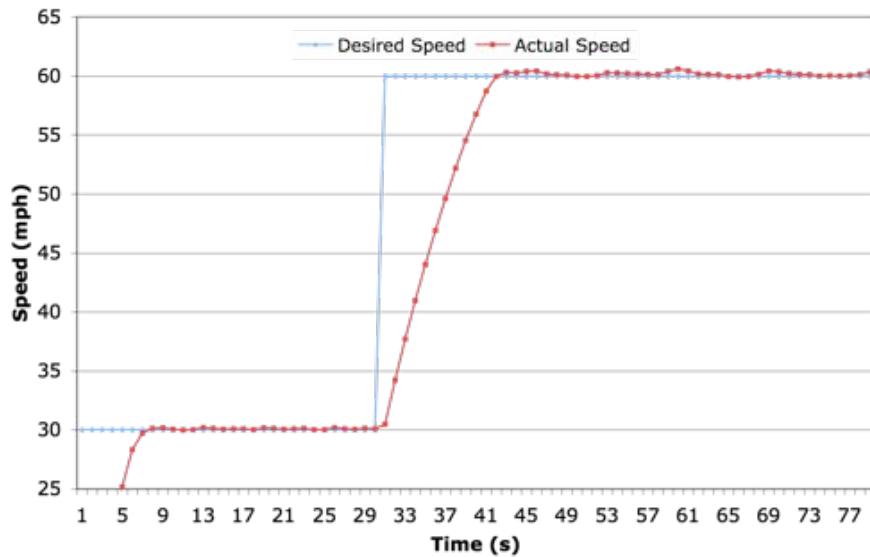


Figure 7: Desired and actual speeds for the single processor CCS.

Note that the actual speed follows the desired speed closely, with little fluctuation – well within the tolerances of the system. Figure 8 shows the corresponding throttle setting, which is the actual output of the cruise controller.

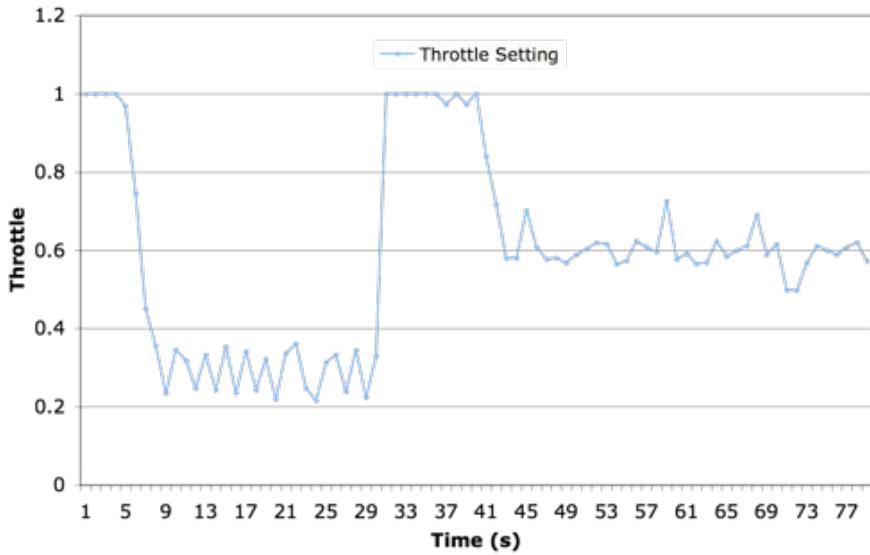


Figure 8: The throttle setting outputted by the single processor CCS.

Figure 9 shows the results for the multi-processors cruise controller.

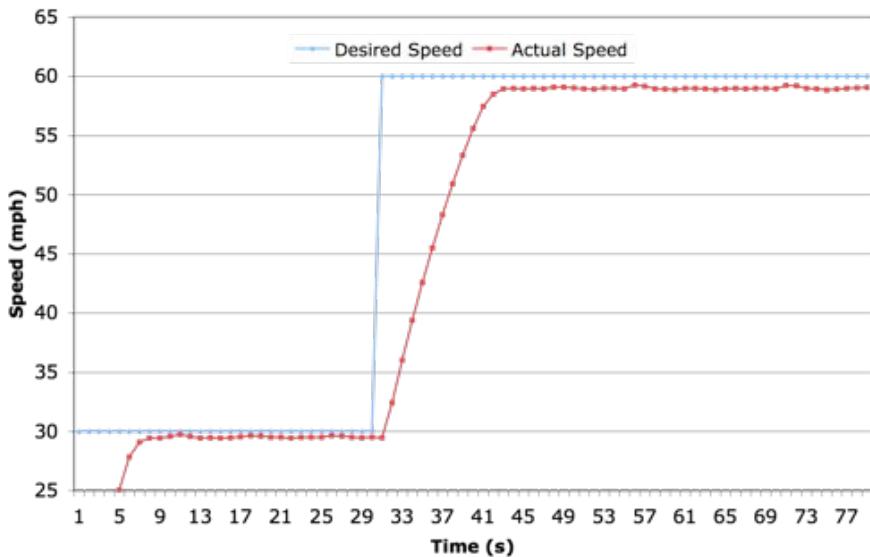


Figure 9: Desired and actual speeds for the multi-processor CCS.

Of particular note in this version is the fact that the car model never actually reaches the desired speed. This is most likely a result of the timing issue discussed previously, caused by the extra delay due to communication between processors. Despite such issues, the result is still well within the tolerances of the system. Figure 10 shows the corresponding throttle settings for the multi-processor CCS variant.

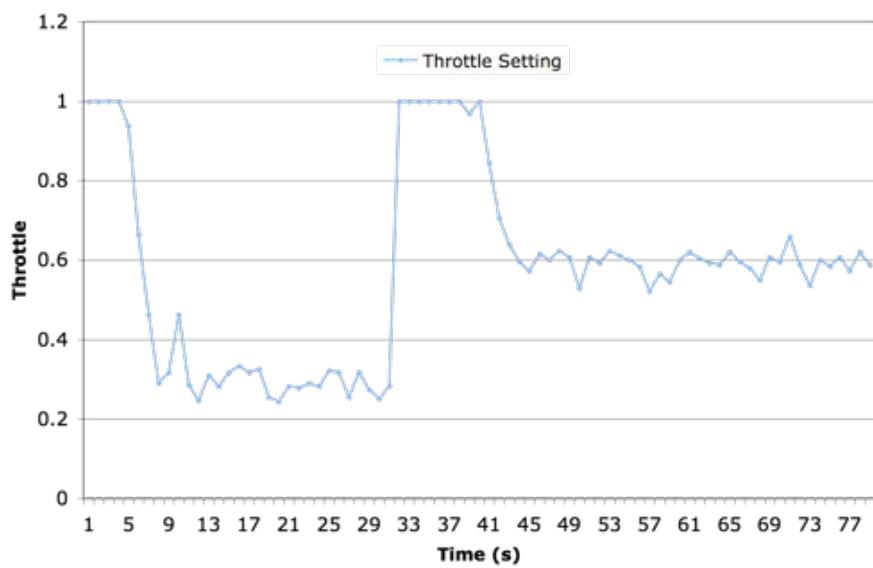


Figure 10: The throttle setting outputted by the multi-processor CCS.

d) Discussion

Although the match between the two systems is imperfect, it is clear that the conversion process has worked correctly.

The conversion has also highlighted some challenges that will need to be faced as this project develops. These relate, in particular, to the timing of the tasks on the two versions of the CCS. In this case, the same timing was used for both systems. This means that we cannot (and do not) expect to obtain an improvement in the system responsiveness as a result of the conversion. To obtain any such improvements will require “hand tuning” of the code.

The need for such hand tuning is highlighted in this example by the way the single-processor system calculates the current speed from the PRM (pulse streams) input. In this case, the PRM input signal is fed into one of the processors timer / counter input lines. The speed is then calculated based on the number of pulses that occur within the task period (originally 50 ms). Altering the task’s period clearly invalidates one of the assumptions upon which this calculation is based and therefore causes the system to produce an incorrect result (in this case calculating the speed as being significantly lower than it actually is).

If we wish to completely automate such a conversion, there are several possible solutions to this problem. One of the more practical would be to allow the programmer access to several constants representing the various parameters that can be changed by the conversion process. For example, we could have used a “`Task_period`” constant in the input task to make the calculation independent of the actual task period, which can no longer be known at the time the source code is written. This could be argued to be “good programming practice”, but it may require extra effort on the developer’s part when first writing the software.

Another aspect of this problem is the differences in the control performance evident in each implementation. For example, the car never reaches the desired speed in the multi-processor implementation (Figure 9). It may be possible to take the same “`Task_period`” approach to address this problem. However, the introduction of this (and related) constants into a control algorithm is beyond the scope of the present paper.

Overall, as this project develops, it is unlikely that it will be practical for the conversion tool to detect all timing-reliant calculations: it is therefore likely that hand tuning will be required in most cases, if optimum performance is to be obtained from the multi-processor implementation.

5. Conclusion

In this paper we have considered the problem of converting between systems in which multiple tasks are executed on a single node and the equivalent design in which multiple nodes each execute a single task. The work presented here represents the first stage in a larger study in which we aim to develop tools which will automate such a conversion process.

In the systems discussed in the present paper, the single-processor designs have been assumed to employ a time-triggered co-operative scheduler and the multi-processor designs have been assumed to employ a “domino” scheduler.

We have described the design of a prototype conversion tool, and reported the results obtained when this tool was applied in an initial case study. The results obtained from the conversion are encouraging.

The case study has also raised some issues about timing assumptions made in the initial system. These issues will be considered more fully as the project develops.

References

Aditya, S., Mahlke, S.A. and Rau, B.R., *Code size minimization and retargetable assembly for custom EPIC and VLIW instruction formats*, HPL-2000-141, HP Laboratories, Palo Alto, (2000).

Aditya, S. and Rau, B.R., *Automatic architecture synthesis and compiler retargeting for VLIW and EPIC processors*, HPL-1999-93, HP Laboratories, Palo Alto, (1999).

Allworth, S.T., *Introduction to real-time software design*, Macmillan, (1981).

Audsley, N.C., Burns, A., Davis, R.I., Tindell, K.W. and Wellings, A.J., *Fixed priority pre-emptive scheduling: an historical perspective*, Real-Time Systems, 8 (1995), pp. 173.

Ayavoo, D., Pont, M.J. and Parker, S., *A 'hardware-in-the-loop' testbed representing the operation of a cruise-control system in a passenger car*, 2nd UK Embedded Forum (submitted), Birmingham, UK, (2005).

Ayavoo, D., Pont, M.J. and Parker, S., *Using simulation to support the design of distributed embedded control systems: A case study*, in Koelmans, A., Bystrov, A. and Pont, M.J., eds., *Proceedings of the 1st UK Embedded Forum*, University of Newcastle upon Tyne, 2004, pp. 54-65.

Bate, I.J., *Introduction to scheduling and timing analysis, The Use of Ada in Real-Time Systems*, IEE Conference Publication 00/034, (2000).

Bate, I.J., *Scheduling and timing analysis for safety critical real-time systems*, PhD Thesis, University of York, UK (1998).

Halambi, A., Grun, P., Ganesh, V., Khare, A., Dutt, N. and Nicolau, A., *EXPRESSION: a language for architecture exploration through compiler/simulator retargetability*, IEEE Comput. Soc, Munich, Germany, (1999), pp. 485.

Henzinger, T.A., Benjamin, H. and Kirsch, C.M., *Embedded control systems development with Giotto*, SIGPLAN Notices, 36 (2001), pp. 64.

Hohenauer, M., Scharwaechter, H., Karuri, K., Wahlen, O., Kogel, T., Leupers, R., Ascheid, G., Meyr, H., Braun, G. and van Someren, H., *A methodology and tool suite for C compiler generation from ADL processor models*, IEEE Comput. Soc, Paris, France, (2004), pp. 1276.

Jie, L. and Lee, E.A., *Timed multitasking for real-time embedded software*, IEEE Control Systems Magazine, 23 (2003), pp. 65.

Kopetz, H., *Real-Time Systems: Design Principles for Distributed Embedded Applications*, Kluwer Academic, New York, (1997).

Locke, C.D., *Software architecture for hard real-time systems: Cyclic executives vs. Fixed priority executives*, The Journal of Real-Time Systems, 4 (1992), pp. 37-53.

Maaita, A. and Pont, M.J., *Using "planned pre-emption" to reduce levels of task jitter in a time-triggered hybrid scheduler*, 2nd UK Embedded Forum (submitted), Birmingham, UK, (2005).

Mellor, S.J., *Automatic code generation from UML models*, C++ Report, 11 (1999), pp. 28.

MISRA, *Development guidelines for vehicle-based software*, Motor Industry Software Reliability Report, (1994).

Moitra, A., *Scheduling of hard real-time systems*, Springer-Verlag, New Delhi, India, (1986), pp. 362.

Mwelwa, C., Pont, M.J. and Ward, D., *Code generation supported by a pattern-based design methodology*, in Koelmans, A., Bystrov, A. and Pont, M.J., eds., *Proceedings of the 1st UK Embedded Forum*, University of Newcastle upon Tyne, 2004, pp. 36-55.

Nissanke, N., *Realtime Systems*, Prentice-Hall, (1997).

Pillai, S. and Jacome, M.F., *Compiler-directed ILP extraction for clustered VLIW/EPIC machines: predication, speculation and modulo scheduling*, IEEE Comput. Soc, Munich, Germany, (2003), pp. 422.

Pont, M.J., *Patterns for Time-Triggered Embedded Systems*, Addison-Wesley, (2001).

Rajagopalan, S., Rajan, S.P., Malik, S., Rigo, S., Araujo, G. and Takayama, K., *A retargetable VLIW compiler framework for DSPs with instruction-level parallelism*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 20 (2001), pp. 1319.

Rust, C., Stappert, F., Altenbernd, P. and Tacken, J., *From high-level specifications down to software implementations of parallel embedded real-time systems*, IEEE Comput. Soc, Paris, France, (2000), pp. 686.

Storey, N., *Safety-Critical Computer Systems*, Addison-Wesley, (1996).

The PH Processor: A soft embedded core for use in university research and teaching

Zemian M. Hughes, Michael J. Pont and Royan H.L. Ong

*Embedded Systems Laboratory,
University of Leicester, University Road, Leicester LE1 7RH, UK.*

Abstract

For the developer of embedded systems, the division between “software” and “hardware” is becoming rather blurred. With the rapid expansion in the availability of FPGAs, design software (using VHDL and related languages) and various “soft” processor cores, it is now becoming possible to create “custom processors” which matches the needs of a particular application. In addition to have numerous commercial applications, such processors are core components in many university research and teaching programmes. In this paper we describe the “PH Processor”, which is freely available for use in university research and teaching.

Acknowledgements

The project described in this paper was supported by the UK Government (EPSRC, DTA award). The processor described in this paper is based on a design by David A. Patterson and John L. Hennessy which is presented in their book “Computer Organization and Design” (Patterson and Hennessy, 2004). We are grateful to Prof Patterson and Prof. Hennessy - and their publishers - for granting us permission to release details of what we refer to here as the “PH Processor”. Work on this paper was completed while MJP was on Study Leave from the University of Leicester.

1. Introduction

Over recent years, we have considered various ways in which time-triggered software architectures can be employed in embedded systems where reliability is a key design consideration (e.g. Pont, 2001; Pont, 2003; Pont and Banner, 2004). The techniques described in these studies have involved creating software for industry-standard hardware platforms, such as the 8051 microcontroller (Pont, 2001), ARM processor (Pont and Mwelwa, 2003) or PC platform (Pont et al., 2003).

Developing reliable applications using this approach can be effective, but there is a mismatch between generic processor architectures and time-triggered software designs. For example, most processors support a wide range of interrupts (e.g. Siemens, 1997; Infineon, 2000; Philips 2004), while the use of a (pure) time-triggered software architecture generally requires that only a single interrupt is active on each processor. This leads to design “guidelines”, such as the “one interrupt per microcontroller rule” (Pont, 2001). Such guidelines can be supported when appropriate tools are used for software creation (e.g. see Mwelwa et al., 2003; Mwelwa et al., 2004). However, it is still possible for changes to be made (for example, during software maintenance or upgrades) that lead to the creation of unreliable systems.

The present paper represents the first step in a new research programme in which we are exploring an alternative solution to this problem. Specifically, we are seeking to develop a novel processor, which is designed to support only time-triggered software. This approach has become possible since the advent of reduced cost of FPGA chips with increasing gate numbers (Gray, 2000). With VHDL and related hardware descriptive languages reaching higher levels of abstraction, the production of radiation hardened FPGA chips (e.g. Atmel AT40KEL040), and fault-tolerant techniques (e.g. Sinha et al., 2000; Hammarberg et al., 2003), such as triple-modular redundancy (Jasinski and Pedroni, 2004), FPGA-based designs are being used in safety-critical applications, such as those in the aerospace industry (Fernandez-Leon , 2002).

In the present paper we describe the first stage of this project in which we have assembled a conventional processor: this will be used as a platform to support our research in this area. Since 32-bit RISC processors are becoming more widely used within embedded systems, we felt that such an architecture would form an appropriate starting point. The specific platform chosen was what

we will refer to here as the “PH Processor”: we have based this design on that described by Patterson and Hennessy (2004)¹.

The paper is organised as follows. In Section 2 we outline the origins of the MIPS microprocessor upon which the PH Processor is based. Section 3 gives a brief overview of the PH Processor. Section 4 introduces the hardware platform upon which the PH Processor was tested. Section 5 has a focus on key implementation details. Section 6 then describes the process of compiling C programs and loading the executables on the hardware: this section also provides a summary of the features of the PH Processor debug application. Section 7 describes tests used on the core and our conclusions are presented in Section 8.

2. The origins of the PH Processor

The “Microprocessor without Interlocked Pipeline Stages” (MIPS) processor was the product of a team led by John Hennessy at Stanford University in 1981 (Stanford, 2003). The aim was to dramatically increase the speed of a processor through the use of deep instruction pipelining. Pipeline designs of that era required interlocks for multi-cycle instructions, in order to prevent the processor from loading new data whilst the current instruction was executing. The hardware required to set up these locks was generally large and complicated: this had a significant impact on the speed of such processors (Hennessy, 1982). Hennessy’s solution was to create a simple RISC instruction set by eliminating a number of complex instructions (such as multiply and divide) which took multiple clock cycles to execute, and – thereby - create an instruction set where all instructions take one clock cycle (Hennessy, 1981). In doing so the pipeline no longer required the complex interlock mechanisms and formed an efficient processor design which is still a feature of many modern MIPS and RISC processors, including those in Sony Playstations, PDAs and larger devices used for research in physics (MIPS Technologies, 2005).

¹ As noted, the “PH Processor” described in this paper is based on a design by David A. Patterson and John L. Hennessy (Patterson and Hennessy, 2004). We are grateful to Prof Patterson and Prof. Hennessy - and their publishers - for granting us permission to release details of this processor. The full PH Processor implementation (VHDL) is available for download from our website (<http://www.le.ac.uk/eg/embedded/PH.htm>).

3. The PH Processor

We describe the PH Processor in this section. Except where otherwise stated, much of the information presented here is adapted from Patterson and Hennessy (2004).

3.1 Overview

There are currently a range of MIPS processors available. Our intention was to create a cut-down version of a R2000 processor which would be compatible with the MIPS I Instruction Set Architecture (ISA: see Kane and Heinrich, 1992) and which excluded any patented instructions.

Our design follows the outline provided by Patterson and Hennessy (2004) and we therefore called it the “PH Processor”. Briefly, this is a 32-bit processor with 32 registers and a 5-stage pipeline. There are two separate ports to memory, one for instruction fetch and one for data access: this is often referred to as a “Harvard” architecture. The processor also includes the system coprocessor CP0, to support precise exceptions (see Section 5.9).

3.2 Instructions

There are three main instruction categories, R-type (register format) I-type (immediate format) and J-type (Jump format), all of which are 32 bits wide.

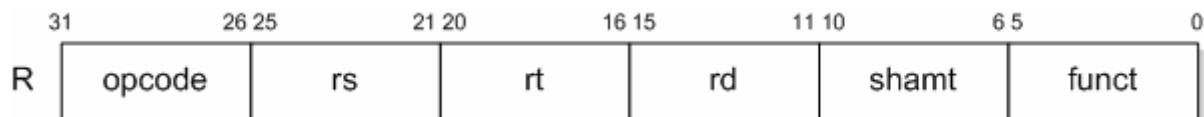


Figure 1: R-type instruction format (adapted from Patterson and Hennessy, 2004)

R-type instructions have two source registers ‘rs’, ‘rt’ and one destination register ‘rd’: the function applied to the source registers is defined in the ‘funct’ field and the ‘shamt’ field contains the shift amount used for shift instructions. R-type instructions are easily identifiable because they have an ‘opcode’ value equal to zero.

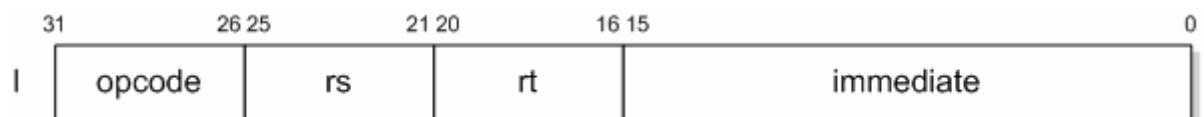


Figure 2: I-type instruction format (adapted from Patterson and Hennessy, 2004)

I-type instructions have one source register ‘rs’, one 16-bit immediate source value and one destination register ‘rt’. I-type instructions are also used for memory load and stores and some conditional branches.



Figure 3: J-type instruction format (adapted from Patterson and Hennessy, 2004)

J-type instruction is primarily used for a jump instruction where the lower 26-bit immediate address value is loaded into the program counters lower 26-bits.

There are also a number of “pseudo instructions” which are not implemented directly in the processor hardware but are represented as a combination of (hardware) instructions when code is compiled.

3.3 Registers

The register bank contains 32×32 -bit registers, where register ‘r0’ is unique in that it is always held equal to zero and write instructions to it are ignored². Register ‘r31’ is a general purpose register but is used specifically for holding the return address across function calls. Details of the registers in the register bank and their assignments are shown in Table 1.

Name	Number	Use	Preserved across Call
\$zero	0	Constant Value 0	N.A.
\$at	1	Assembler Temporary	No
\$v0-\$v1	2-3	Values for Function Results and Expression Evaluation	No
\$a0-\$a3	4-7	Arguments	No
\$t0-\$t7	8-15	Temporaries	No
\$s0-\$s7	16-23	Saved Temporaries	Yes
\$t8-\$t9	24-25	Temporaries	No
\$k0-\$k1	26-27	Reserved for OS Kernel	No
\$gp	28	Global Pointer	Yes
\$sp	29	Stack Pointer	Yes
\$fp	30	Frame Pointer	Yes
\$ra	31	Return Address	Yes

Table 1: Register names and assignments (from Patterson and Hennessy, 2004)

² This provides a useful reference register for instructions (such as conditional branches) which can use the contents of r0 to in comparisons.

The program counter (PC) is not part of the register bank and is not directly accessible to the programmer. More specifically, the programmer only has access to the PC through jump and branch instructions (write access) and jump and link instructions (read access).

Note that the 2 least-significant bits of the PC are always zero as each instruction is 32 bits wide, word aligned and separated by an address of 4.

3.4 Branch Conditions

The processor does not contain condition flags but uses conditional branch instructions to make decisions. There are two branch instructions: BEQ (Branch Equal) and BNE (Branch Not Equal). When these two instructions are combined with SLT (Set Less Than) or SLTI (Set Less Than Immediate), they provide the conditions required to support ‘C’ programs whilst keeping the number of instructions to be implemented to a minimum.

Name	Mnemonic	Pseudo	Compiled to	Logic
Branch Equal	BEQ	-	BEQ	
Branch Not Equal	BNE	-	BNE	
Branch Less Than	BLT	Pseudo	SLT \$t0, \$s0, \$s1 BNE \$t0, \$0, Less	if (\$s0 < \$s1) \$t0 = 1 if (\$t0 != 0) goto Less
Branch Greater Than	BGT	Pseudo	SLT \$t0, \$s1, \$s0 BNE \$t0, \$0, Greater	if (\$s0 > \$s1) \$t0 = 1 if (\$t0 != 0) goto Greater
Branch Less Than or Equal	BLE	Pseudo	SLT \$t0, \$s1, \$s0 BEQ \$t0, \$0, LEqual	if (\$s0 > \$s1) \$t0 = 1 if (\$t0 = 0) goto LEqual
Branch Greater Than or Equal	BGE	Pseudo	SLT \$t0, \$s0, \$s1 BEQ \$t0, \$0, GEqual	if (\$s0 < \$s1) \$t0 = 1 if (\$t0 = 0) goto GEqual

Table 2: Conditional branches (adapted from Patterson and Hennessy, 2004)

3.5 Load and Store

Memory load and store instructions follow the I-type instruction format. In this case, the pointer (in the base register ‘rs’) is added to the immediate offset value to form an address. This address is, in turn, loaded with the contents of the destination register ‘rt’ (in the case of “load” instructions), or used to store the contents of ‘rt’ (in the case of store instructions).



Figure 4: I-type instruction format (adapted from Patterson and Hennessy, 2004)

The RAM is 32 bits wide but is byte addressable. It is a requirement with the PH Processor that 32-bit words should be word aligned, and 16-bit half words should be aligned to half-word boundaries (Figure 5).

Double Word							
Word				Word			
Half Word		Half Word		Half Word		Half Word	
Byte	Byte	Byte	Byte	Byte	Byte	Byte	Byte
0	1	2	3	4	5	6	7

Figure 5: Data alignment (adapted from Patterson and Hennessy, 2004)

Please note that there are instructions in the MIPS I ISA to support unaligned word load and stores such as LWL (Load Word Left) and LWR (Load Word Right). However, these instructions are patented by MIPS Technologies and are not implemented here.

3.6 Dealing with control hazards

The PH Processor has a 5-stage pipeline based on the design summarised in Figure 6. Each pipeline stage takes one clock cycle. Between each stage we require pipeline registers to store the outcome of the previous stage.

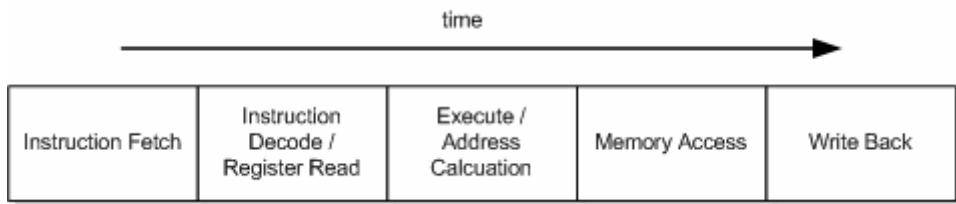


Figure 6: The basis of the 5-stage pipeline used in the PH Processor (adapted from Patterson and Hennessy, 2004)

When making a conditional decision - such as a branch command where the decision cannot be taken until the command is executed - the pipeline stages will already have been loaded with the “next” instructions before the decision is made. If care is not taken, we will execute the “next” instructions, regardless of the result of the conditional decision: this possibility is known as a control hazard.

To avoid control hazards, it is best to identify the outcome of conditional decisions as early as possible in the pipeline. For example, if we identify the results of a branch command in the ID (Instruction Decode) stage, there will only be one (possibly) unwanted instruction in the pipeline.

One method to remove this unwanted instruction is to detect and stall the pipeline after the branch instruction by forcing a NOP command directly after the branch. However, a more efficient solution is to have the compiler reorder its output in such a way that an instruction which must (always) be executed regardless of the branch decision is placed directly after the branch instruction: this is known as the branch delay slot. Note that - if no suitable instruction can be identified - then the compiler can place a NOP command in the delay slot.

In the present version of the PH Processor (v1.0), we include optional hardware features to deal with control hazards but by default we assume that a suitable compiler is employed (see Section 6.2).

3.7 Dealing with data hazards

When a “current” instruction is dependant on the results of a “preceding” instruction we may have what is known as a “data hazard”. Such a situation arises when the preceding instruction has not reached the write-back stage (and updated the register file) when the current instruction requires the register value.

To deal with data hazards, a forwarding unit can be used to check if a register to be read in the EX (Execution) stage is to be written to in the MEM (Memory) or WB (Write Back) stages. If this “read before write” situation arises then the forwarding unit routes the register value from either the MEM or WB stage to the EX stage. If both the WB and MEM stage have their own copy of the register then the MEM stage value is forwarded to the EX stage (as it is the most recent copy).

Note that there is one condition when a value cannot be forwarded: this is when an instruction following a load instruction is dependant on the value to be loaded from memory for the register in use. This is because there is no way to force the value out of the data memory any faster than its read time (which is after the MEM stage). The solution to this is similar to that described for control hazards in Section 3.6: in the case of data hazards, we can use a load delay slot to place a NOP instruction directly after a load instruction in situations where the following instruction is dependant on the output of the current instruction. This solution may not always be ideal as it will increase code size, so a data hazard unit can be used to stall the IF (Instruction Fetch) stage to force hardware NOP command instead.

The present implementation of the PH Processor follows Patterson and Hennessy (2004) and uses a hardware data-hazard unit.

4. Hardware platform used

To test the design outlined in the previous sections, the processor was set up on an FPGA as shown in Figure 7. Implemented in this way, the core contains its own instruction and data memories and the device can be programmed and debugged through the serial UART, which connects to a PC application that can monitor everything down to the control and data paths.

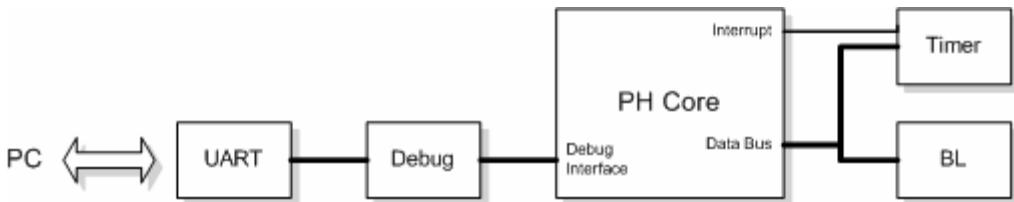


Figure 7: Layout of the PH core and supporting systems

The BL (Buttons and Lights) block uses memory map IO to interface to the onboard LEDs and IO pins. The timer is attached on the data bus where the necessary registers are easily addressed through normal memory load and store instructions.

Our current implementation of the PH Processor was created using VHDL with Xilinx ISE tools targeting a Xilinx 200K gate Spartan 3 FPGA chip on a Digilent Spartan 3 development board. There are 216Kbits of block RAM on the chip and 1MB of SRAM on-board. The board contains a serial port, LEDs, seven-segment display, buttons and switches: it costs around £60 (UK pounds).



Figure 8: Digilent Spartan 3 development board (photograph by Royan Ong)

5. Implementation Details

The core was implemented to run all (non patented) integer instructions in order to create a simple processor core base. Note that co-processor devices, extra instructions and peripherals may be easily added to the design.

5.1 Pipeline

The details of the 5-stage pipeline used closely resembles the setup described by Patterson and Hennessy (2004), with one exception. As discussed by Brej (2002), the branch instructions in the Patterson and Hennessy design require a conditional test on two registers in the ID stage: the values required are obtained from the register file, not from the most recent forwarded value from the consecutive stages. The forwarding unit makes available the most recent value to the EX stage and - since we wish to keep the branch execution in the ID stage to minimise the number of delay slots after a branch instruction - we could add an extra separate forwarding unit. This would work but would complicate the design. Brej (2002) describes a modification which shrinks the ID stage to half a clock cycle: this means that the condition branch instructions can use the forwarded values in the EX stage and update the PC before the next clock cycle, thereby maintaining the one-branch delay slot. This is an efficient solution and the Brej implementation was used in the present PH Processor design.

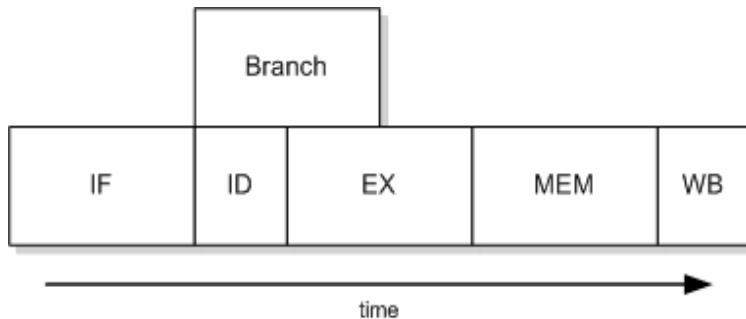


Figure 9: Modified 5-stage pipeline (Brej, 2002)

Reducing the WB stage to half a cycle also aids with synchronous RAM register descriptions where load and stores are not simultaneous.

5.2 Register bank

Two implementations of the register file were created, one synchronous (using block RAM) and one asynchronous (using distributed RAM). In many circumstances the synchronous block RAM implementation is preferable in that it is faster and does not use a large amount of logic. However, while the distributed RAM is created directly from logic and look-up tables (which may be scarce

once a system design is complete) it has the advantage that the RAM can be read and written at the same time. In the current PH Processor implementation, the choice of register-file implementation can be made when compiling the design. Since the synchronous block RAM must be clocked for both read and writes, one half of the clock cycle is used for writing whilst the other half is used for reading.

In order to generate the register file (which contains one write port and two read ports) two dual-ported RAMs with identical copies of the register values are used (Figure 10). This allows simultaneous reads of two registers, as required. Also if the register to be read is ‘r0’ then the output must be zero, which is achieved by blocking any writes to ‘r0’ (ensuring that only the initial RAM value - 0 - is read).

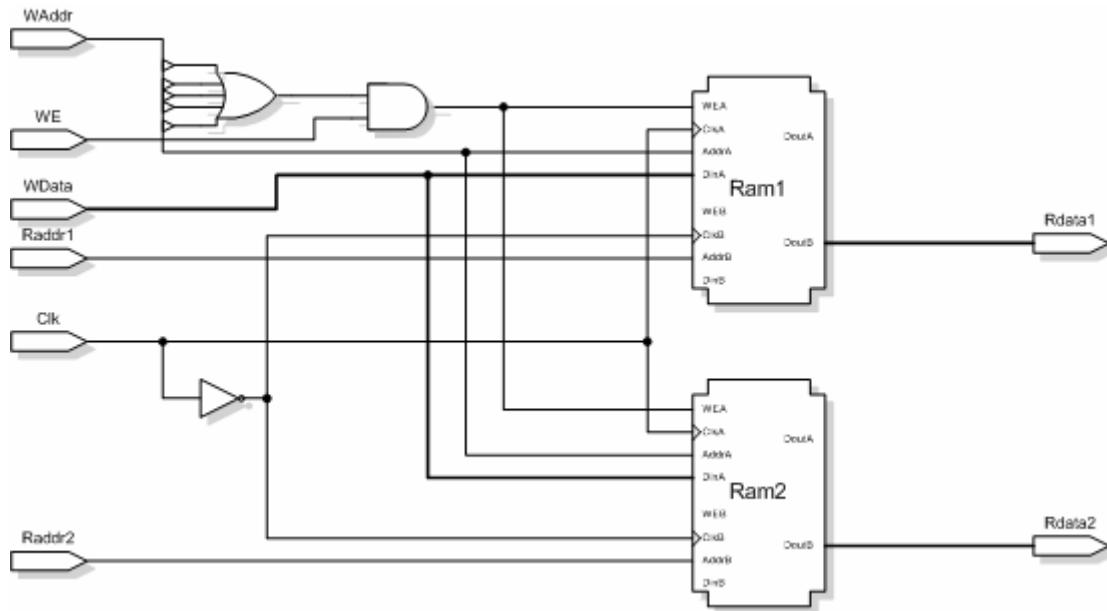


Figure 10: Block RAM register file configuration

5.3 Control

As with the Patterson and Hennessy (2004) design there are two control units in the PH Processor, the main control unit in the ID stage for non R-type instructions and an R-type controller in the EX stage (which not only controls the ALU but also the Shifter and R-type jump commands). Splitting the control unit into two increases the instruction decoding speed which is useful since – as discussed in Section 5.1 - the ID stage has been shrunk to half a clock cycle. For more details of the PH processor data and control paths, see (Figure 11).

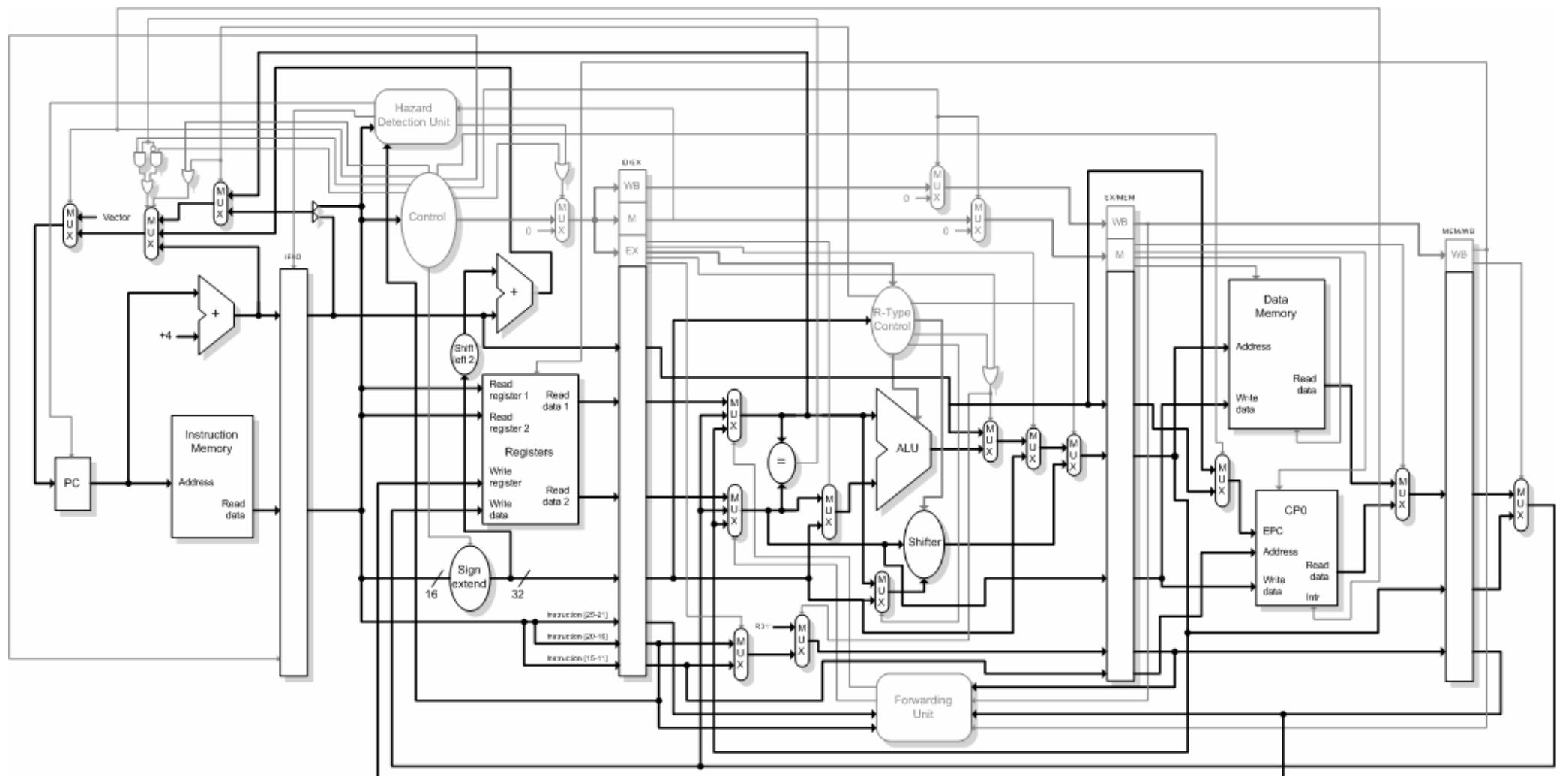


Figure 11: Data (black) and control (grey) paths of the PH Processor

5.4 ALU

The ALU is made up of combinational logic where the two input values are run through each of the logic operations (AND, OR, NOR and XOR) and a selectable adder/subtract unit: the required value is selected through a multiplexer. This solution was chosen purely for simplicity.

The ALU also detects overflows which are used to generate exceptions for certain instructions. The SLT instructions are also implemented in the ALU.

5.5 Shifter

Barrel shifters, as used in processors such as ARM: (Furber, 2000), are very fast in ASICs, but in FPGAs they require 32×32 -bit multiplexers, which take up a large number of gates. A solution to this problem (which maintain a variable left or right shift in one clock cycle) is to implement a funnel shifter (Braj, 2002). The funnel shifter passes the value to be shifted through 5 multiplexers which are wired to give the corresponding shift amounts (1, 2, 4, 8 and 16). The bits in the ‘shamt’ field of R-type instructions select the individual multiplexers to provide the required shift. To keep things simple, the values pass through two sets of 5 multiplexers, one for shift right and one for shift left, with a multiplexer used to select the required shift direction.

5.6 Memory

The data memory is 32 bits wide but needs to be byte addressable, so that instructions LW, LH, LB can load 32-bit, 16-bit and 8-bit values respectively.

To implement this a 32-bit wide memory can be used with the 2 least significant bits selecting a multiplexer to give the required aligned value out. However this solution presents a problem with the store instructions such as SB (which write a selected byte to a word without affecting the rest of values in that word). Since the on-board RAM does not have byte-select pins, the 32-bit RAM was made up of four 8-bit RAMs connected in parallel: this ensures that the SB command can enable the write-enable pin of the individual RAM block to which the byte must be written.

Note that synchronous block RAM was used as the logic required to generate a large amount of distributed RAM would use a large portion of Spartan 3 chip used in this test implementation (which contains only 256K bits of on-board RAM)³.

5.7 Branch

The input to the PC is selectable through 3 multiplexers (Figure 11). The first multiplexer is used to select the interrupt/exception vector address. The second multiplexer selects input from PC+4 (branch address or a jump address). The third multiplexer selects the jump address from the forwarding general-purpose register path or from the four most significant PC+4 bits plus the lower immediate 26-bit value.

The registers used for the branch decision are taken from the forwarding paths and compared: if their contents are the same, the branch multiplexer is set. The branch address is calculated from the old PC+4 address plus the 16-bit sign extended immediate value. If the instruction is a jump or link then the PC+4 value is multiplexed into the ALU result path to be written to the register file and the write register path is made equal to ‘r31’ (the return address register).

5.8 Forwarding unit

If a register number to be used in the EX stage is equal to the register to be written in the MEM or WB stage and the destination register is not ‘r0’, then the most recent value is multiplexed in as an input to the ALU. The forwarding unit performs the operations shown in Table 3 (as detailed in Patterson and Hennessy, 2004).

MEM Forwarding Equations	WB Forwarding Equations
<pre>if (MEM.RegWrite and (MEM.RegisterRd ≠ 0) and (MEM.RegisterRd = EX.RegisterRs)) ForwardA = 10</pre>	<pre>if (WB.RegWrite and (WB.RegisterRd ≠ 0) and (ForwardA ≠ MEM) and (WB.RegisterRd = EX.RegisterRs)) ForwardA = 01</pre>
<pre>if (MEM.RegWrite and (MEM.RegisterRd ≠ 0) and (MEM.RegisterRd = EX.RegisterRt)) ForwardB = 10</pre>	<pre>if (WB.RegWrite and (WB.RegisterRd ≠ 0) and (ForwardB ≠ MEM) and (WB.RegisterRd = EX.RegisterRt)) ForwardB = 01</pre>

Table 3: Forwarding unit equations (adapted from Patterson and Hennessy, 2004)

³ If more RAM is required the Digilent board has 1MByte of asynchronous 10ns SRAM arranged as two 256 x 16bit chips with individual byte select pins which is ideal for MIPS implementations and can be used to expand either the data or instruction RAM.

5.9 Exceptions

The MIPS processor can support four tightly-coupled coprocessors with 32 general registers and 32 control registers (Kane and Heinrich, 1992). One of these coprocessors, CP0 (“Coprocessor Zero”) - which is also known as the system control coprocessor - contains special-purpose registers for use with memory management and exception handling. In particular it contains the EPC (Exception Program Counter), Cause register and SR (Status Register). The SR register is important as it contains flags to enable and mask interrupts: please note that on the PH Processor we only implement the IE (Interrupt Enable) flag as we only have one interrupt source.

When an interrupt occurs the pipeline is flushed and the EPC register is automatically loaded with the address of the currently executing instruction⁴. The coprocessor registers are accessed by instructions MTC0 and MFC0 which are very similar to load and store instructions: the coprocessor is therefore implemented within the MEM stage of the pipeline.

6. Programming and debugging tool

A simple (programming and) debug tool was created using Visual Basic. The tool, and the process of programming the PH Processor, are described in this section.

6.1 Overview

The debug program executes on a PC and is connected to the FPGA board via the UART. The debug control unit was created to interface with the processor core. The transmissions are sent in 64-bit data packets (with room left for user added debug functions) (Figure 12).

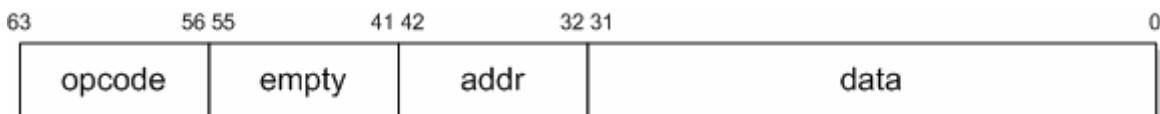


Figure 12: PH debug controller data packets

⁴ This does not happen if the instruction is a branch delay slot (as would be indicated by the BD flag in the cause register). In these circumstances, the EPC is loaded with the address of the previously-executing instruction.

6.2 Compiler

Programs can be compiled and assembled using a MIPS port of GNU GCC and Binutils. The output of the assembled, compiled and linked process is an ELF 32 format object file, so a program was written to extract the binary data of the text and data segments into two binary files called ‘code.bin’ and ‘data.bin’.

6.3 Programming

The two files created from the compilation process outlined in Section 6.2 are loaded into the onboard instruction and data RAMs at address zero via the debug program (using the serial port). The instruction memory window is updated with a (disassembled) view of the program loaded into the instruction RAM. The data memory window is updated with the contents of the data RAM.

Once loaded the processor must be reset by clicking the reset button before trying to run or single-step the code.

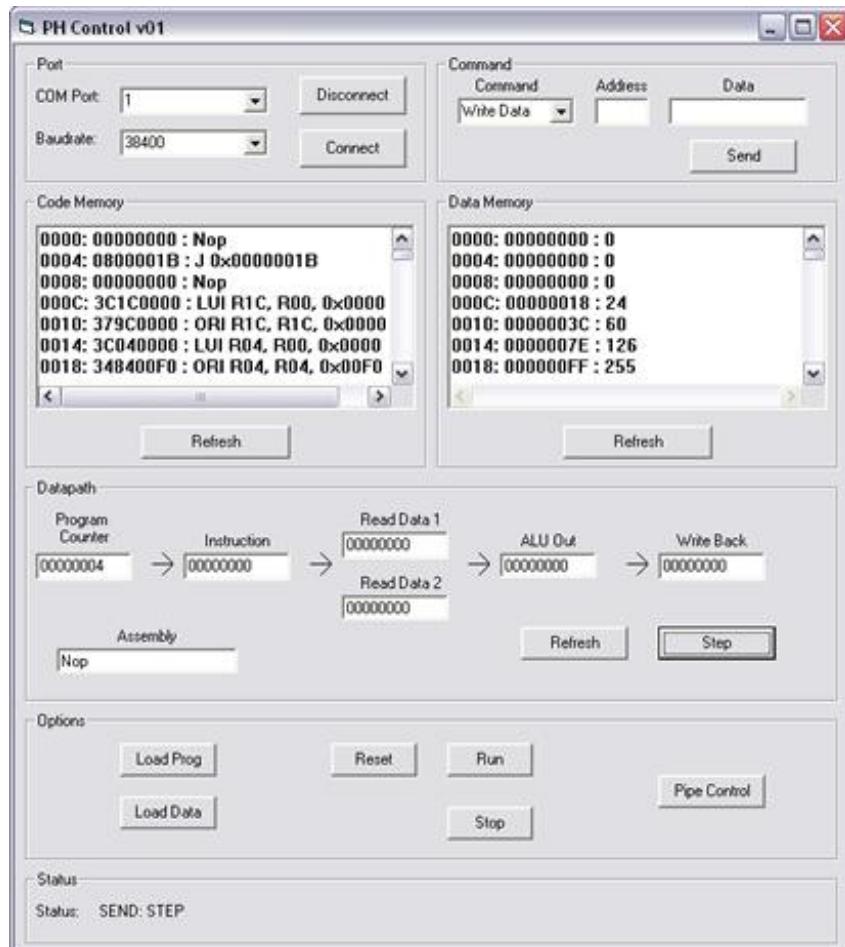


Figure 13: PH main control debug form

6.4 Control and Data path Debugging

By clicking Pipe Control on the form shown in Figure 13, the user opens a new form which allows the loaded program to be single stepped whilst monitoring the instructions data and control paths as they filter through the pipeline stages. The contents of the register file and the data memory are also displayed during this process. This system is useful as it allows the developer to check that any processor modifications operate as required and that new instructions added to the processor are operating as intended.

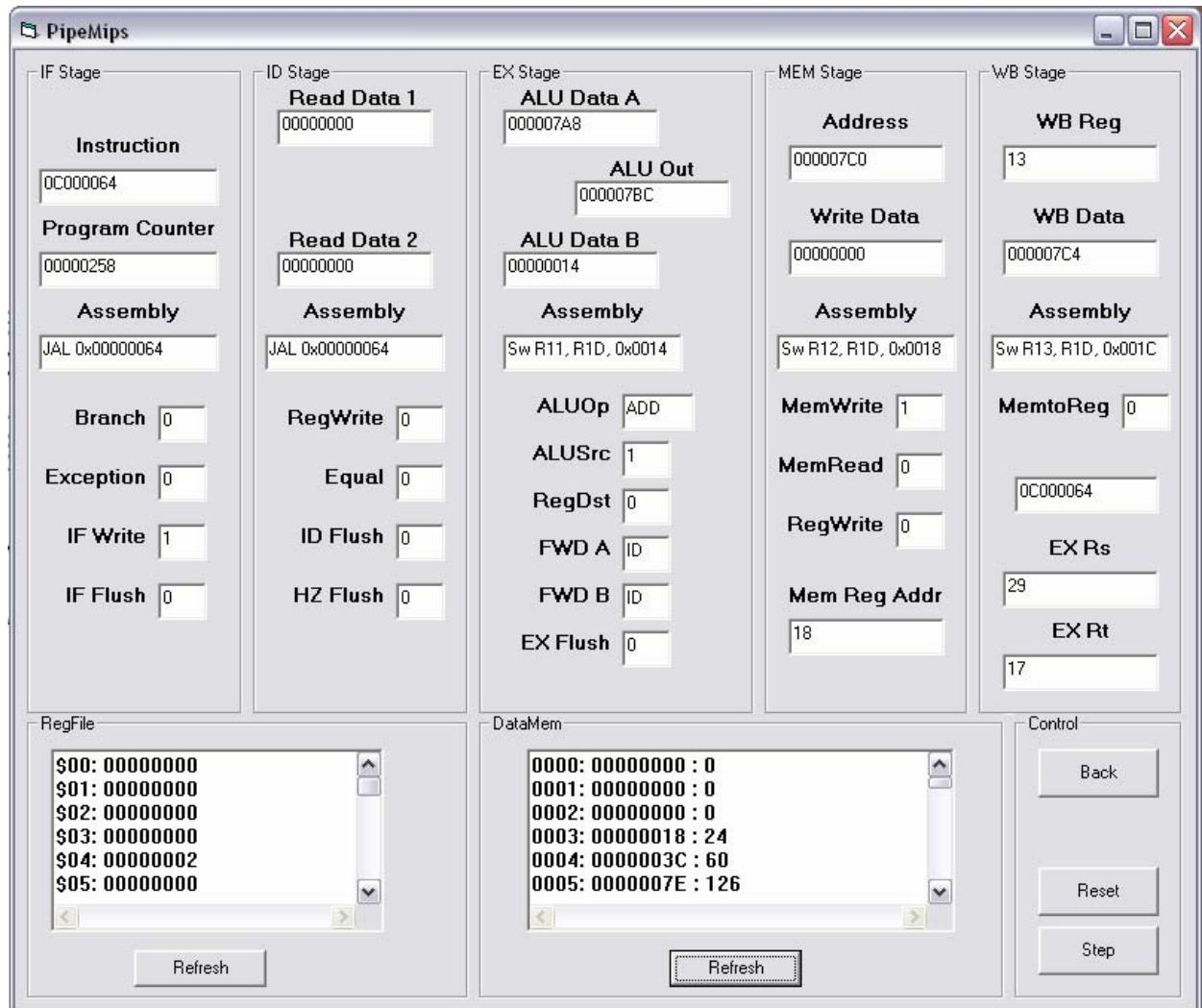


Figure 14: PH single step control and data paths debug form

7. Tests

The PH Processor has been tested using simulations and by running (with the debugger) a range of test programs, including a co-operative scheduler with multiple tasks flashing the on-board LEDs. These tests have been conducted at a clock speed of 25MHz.

The core plus the additional timer, UART, debugger control unit and BL controller utilize about 79% of the 200K Spartan chip. This includes the large distributed RAM register file implementation, 2KB of instruction RAM and 8KB of data RAM.

8. Conclusion

This paper has presented the results from the first stage of a new research programme in which we are seeking to develop a novel processor which is designed to support only time-triggered software. In the present paper we have described a 32-bit (conventional) processor – the PH Processor – which will be used as a platform for this work.

The core of the PH Processor is simple and well documented as it follows the outline described by Patterson and Hennessy (2004). This processor provides a good foundation for further research on processor design.

Further information about this processor can be found on the ESL WWW site:

<http://www.le.ac.uk/eg/embedded/PH.htm>

References

- Brej, C. (2002) “A MIPS R3000 microprocessor on an FPGA”, Yellow Star. Manchester University, UK. http://brej.org/yellow_star/
- Callahan, T.J. (2002) “Automatic Compilation of C for Hybrid Reconfigurable Architectures”, A dissertation submitted in partial satisfaction of the requirements for the degree of Doctor of Philosophy, University Of California, Berkeley
- Diligent Inc. (2004) “Spartan 3 Board”, USA. <http://www.digilentinc.com/info/S3Board.cfm>
- Fernandez-Leon, A. Pouponnot, A. and Habinc, S. (2002) “ESA FPGA Task Force: Lessons Learned” European Space Agency/ESTEC, The Netherlands and Gaisler Research, Gothenburg, Sweden.
- Furber, S. (2002) “ARM System-on-Chip Architecture”, 2nd Edition, Addison Wesley, ISBN: 0 20167 519 6
- Gray, J. (2000) “Designing a Simple FPGA-Optimized RISC CPU and System-on-a-Chip”, Gray Research LLC, <http://www.fpgacpu.org>
- Hammarberg, J. Nadjm-Tehrani, S. (2003) “Development of Safety-Critical Reconfigurable Hardware with Esterel”, Linköping University, Sweden.

Hennessy, J. (1981) "MIPS: a VLSI processor architecture", Technical Report: CSL-TR-81-223, Stanford University.

Hennessy, J. (1982) "Code Generation and Reorganization in the Presence of Pipeline Constraints", Technical Report No. CSL-TR-81-224, Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. Mexico.

Infineon (2000) "C167CR Derivatives 16-Bit Single-Chip Microcontroller: User manual", Infineon Technologies.

Jasinski, R. Pedroni, V.A. (2004) "Evaluating Logic Resources Utilization in an FPGA-Based TMR CPU", Federal Center of Technological Education of Parna, Brazil.

Kane, G. and Heinrich, J. (1992) "MIPS RISC Architecture – Introducing the R4000 Technology", Prentice Hall, New Jersey.

Mei, B. Schaumont, P. Vernalde, S. (2000) "A Hardware-Software Partitioning and Scheduling Algorithm for Dynamically Reconfigurable Embedded Systems", IMEC vzw, 11th ProRISC workshop on Circuits, Systems and Signal Processing Veldhoven, Netherlands.

Mei, B. Vernalde, S. Verkest, D. Lauwereins. (2004) "Design methodology for a tightly coupled VLIW/Reconfigurable Matrix Architecture: A Case Study", IMEC vzw, Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'04)

MIPS Technologies, (2005) "Markets Overview" MIPS Technologies, Inc.
http://www.mips.com/content/Markets/Overview/content_html

Mwelwa, C., Pont, M.J. and Ward, D. (2003) "Towards a CASE tool to support the development of reliable embedded systems using design patterns", paper presented at the workshop "Quality of Service in Component-Based Software Engineering", June 20th, 2003, Toulouse, France.

Mwelwa, C., Pont, M.J. and Ward, D. (2004) "Using patterns to support the development and maintenance of software for reliable embedded systems: A case study", Proceedings of the IEE / ACM Postgraduate Seminar on "System-On-Chip Design, Test and Technology", Loughborough, UK, 15 September 2004. Published by IEE. ISBN: 0 86341 460 5 (ISSN: 0537-9989)

Patterson, D.A.. and Hennessy, J. L. (2004), "Computer Organization and Design: The Hardware/Software Interface", (Third Edition). Morgan-Kaufmann, San Francisco

Philips (2004) "LPC2119/ 2129/ 2194/ 2292/ 2294 microcontrollers: User manual", Philips Semiconductor.

Pont, M.J. (2001), "Patterns for time-triggered embedded systems: Building reliable applications with the 8051 family of micro controllers", Addison-Wesley / ACM Press.

Pont, M.J. (2003) "Supporting the development of time-triggered co-operatively scheduled (TTCS) embedded software using design patterns", *Informatica*, 27: 81-88.

Pont, M.J. and Banner, M.P. (2004) "Designing embedded systems using patterns: A case study", *Journal of Systems and Software*, 71(3): 201-213.

Pont, M.J. and Mwelwa, C. (2003) "Developing reliable embedded systems using 8051 and ARM processors: Towards a new pattern language" Paper presented at VikingPLoP 2003 (Bergen, Norway, September 2003).

Pont, M.J., Norman, A.J., Mwelwa, C. and Edwards, T. (2003) "Prototyping time-triggered embedded systems using PC hardware". Paper presented at EuroPLoP 2003 (Germany, June 2003).

Siemens (1997) "C515C 8-bit CMOS microcontroller: User manual", Siemens.

Singleterry Jr, R.C. Sobieszcanski-Sobieski, J. Brown, S. (2002) "Field-Programmable gate array computer in structural analysis: An Initial Exploration", NASA Langley Research Center. Virginia.

Sinha, S.K. Kamarchik, P.M. Goldstein, S.C. (2000) "Tuneable Fault Tolerance for Runtime Reconfigurable Architectures", Carnegie Mellon University, Pittsburgh.

Wolf, T. Franklin, M. (2000) "Commbench – A Telecommunications Benchmark For Network Processors", In IEEE intl. Symp. On Performance Analysis of Systems and Software

Appendix: Instruction Set of PH Processor

Instructions taken from MIPS 1 ISA (Kane and Heinrich, 1992).

Non R-type instructions

Mnemonic	Name	Opcode
LW	Load Word	100011
SW	Store Word	101011
LB	Load Byte	100000
LBU	Load Byte Unsigned	100100
LH	Load Halfword	100001
LHU	Load Halfword Unsigned	100101
SB	Store Byte	101000
SH	Store Halfword	101001
BEQ	Branch On Equal	000100
BNE	Branch On Not Equal	000101
BLEZ	Branch Less Than and Equal Zero	000110
BGTZ	Branch Greater Than Zero	000111
ADDI	Add Immediate	001000
ADDIU	Add Immediate Unsigned	001001
ANDI	And Immediate	001100
J	Jump	000010
JAL	Jump And Link	000011
LUI	Load Upper Immediate	001111
ORI	Or Immediate	001101
XORI	Exclusive Or Immediate	001110
SLTI	Set Less Than Immediate	001010
SLTIU	Set Less Than Immediate Unsigned	001011

Mnemonic	Name	Opcode	RT
BLTZ	Branch On Less Than Zero	000001	00000
BLTZAL	Branch On Less Than Zero and Link	000001	10000
BGEZ	Branch On Greater Than and Equal Zero	000001	00001
BGEZAL	Branch On Greater Than and Equal Zero and Link	000001	10001

R TYPE

Mnemonic	Name	Funct
JR	Jump Register	001000
JALR	Jump And Link Register	001001
ADD	Add	100000
ADDU	Add Unsigned	100001
SUB	Subtract	100010
SUBU	Subtract Unsigned	100011
AND	And	100100
OR	Or	100101
SLT	Set Less Than	101010
SLTU	Set Less Than Unsigned	101011
NOR	Nor	100111
XOR	Exclusive Or	100110
SLL	Shift Left Logical	000000
SLLV	Shift Left Logical Variable	000100
SRA	Shift Right Arithmetic	000011
SRAV	Shift Right Arithmetic Variable	000111
SRL	Shift Right Logical	000010
SRLV	Shift Right Logical Variable	000110

Special Coprocessor instructions

Mnemonic	Name	Opcode	MT/MF
MTC0	Move To System Control Coprocessor	010000	00100
MFC0	Move From System Control Coprocessor	010000	00000

Two novel shared-clock scheduling algorithms for use with CAN-based distributed systems

Devaraj Ayavoo¹, Michael J. Pont¹, Michael Short¹ and Stephen Parker²

¹*Embedded Systems Laboratory,
University of Leicester, University Road, Leicester LE1 7RH.*

²*Pi Technology,
Milton Hall, Ely Road, Milton, Cambridge CB4 6WZ.*

Abstract

The Controller Area Network (CAN) protocol is widely employed in the development of distributed embedded systems. Previous studies have illustrated how a “Shared-Clock” (S-C) algorithm can be used in conjunction with CAN-based microcontrollers to implement time-triggered network architectures. This study explores some limitations of the existing S-C algorithms (“TTC-SC1” and “TTC-SC2”), and introduces two new algorithms (TTC-SC3 and TTC-SC4). The results presented in the paper suggest that TTC-SC3 and TTC-SC4 are useful additions to the range of shared-clock algorithms.

Acknowledgements

This work is supported by an ORS award (to DA) from the UK Government (Department for Education and Skills), by Pi Technology and by the Leverhulme Trust. Work on this paper was completed while MJP was on Study Leave from the University of Leicester.

1. Introduction

Over recent years, we have considered various ways in which time-triggered software architectures can be employed in low-cost embedded systems where reliability is a key design consideration (e.g. Pont, 2001; Pont, 2003; Pont and Banner, 2004). Our previous work in this area has focused on the development of both single- and multi-processor designs. In the case of multi-processor designs, we have sought to demonstrate that a “Shared-Clock” (S-C) architecture provides a simple, flexible platform for many systems (Pont, 2001). In such designs, the Controller Area Network (CAN) protocol - introduced by Robert Bosch GmbH in the 1980s (Bosch, 1991) - provides high-reliability communications at low cost (Farsi and Barbosa, 2000; Fredriksson, 1994; Sevillano et al., 1998; Thomesse, 1998). Since the CAN protocol has become widely used in many sectors, such as automotive and automation (Farsi and Barbosa, 2000; Fredriksson, 1994; Misbahuddin and Al-Holou, 2003; Pazul, 1999; Sevillano et al., 1998; Thomesse, 1998; Zuberi and Shin, 1995), most modern microprocessor families now have members with on-chip support for this protocol (e.g. Infineon, 2004; Philips, 1996; Philips, 2004; Siemens, 1997).

The original S-C protocols were introduced in 2001 (Pont, 2001). In this paper, we consider some of the features and limitations of two such protocols, which will be referred to here as “TTC-SC1” and “TTC-SC2”*. We go on to present two new S-C protocols – TTC-SC3 and TTC-SC4 – which have features better matched to the needs of some applications.

The paper is organised as follows. Section 2 and Section 3 of the paper gives an overview of the TTC-SC1 and TTC-SC2 algorithms, respectively. Section 4 discusses some of the limitations of TTC-SC1 and TTC-SC2. TTC-SC3 and TTC-SC4 are introduced in Section 5 and Section 6, respectively. An initial evaluation of the TTC-SC3 and TTC-SC4 algorithms is presented in Section 7. Section 8 goes on to discuss some of the weaknesses of the TTC-SC3 and TTC-SC4 algorithms. Our conclusions are presented in Section 9.

* Please note that the algorithm referred to here as “TTC-SC1” was referred to as “SCC Scheduler” in the original publication (Pont, 2001). “TTC-SC2” was originally viewed as a variant of TTC-SC1.

2. The TTC-SC1 algorithm

Although CAN is often viewed as an event-triggered protocol (Leen and Heffernan, 2002), it has been shown that time-triggered behaviour can be achieved using TTC-SC1 (Pont, 2001). An overview of the TTC-SC1 algorithm is presented in this section.

a) Synchronising the nodes

The TTC-SC1 algorithm is a time division multiple access (TDMA) protocol based on CAN. The key idea behind TTC-SC1 is to synchronise the clocks on the individual nodes by sharing a single clock source between the various processor boards (see Figure 1).

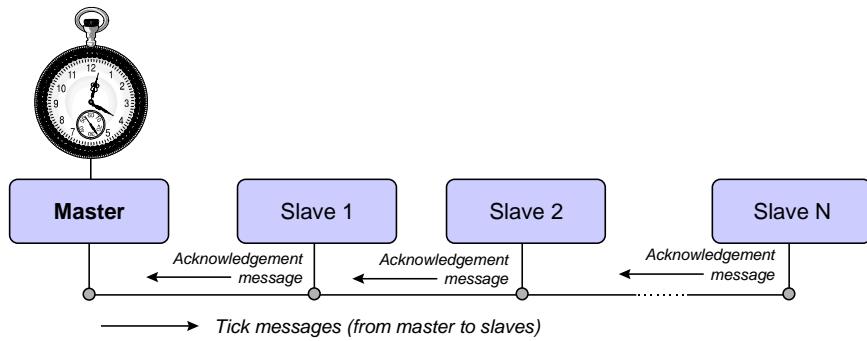


Figure 1: Communication between Master and Slaves nodes in S-C architectures

In this case, we have a single accurate clock on the Master node that generates periodic timer interrupts to drive the scheduler (for example a time-triggered co-operative scheduler in Pont, 2001) of the Master node. In addition, the Master node also generates a Tick message that is sent to the Slave nodes connected on the network using a CAN message. All the Slave nodes respond to the Tick message by generating an interrupt (from the CAN hardware). This interrupt will in turn be used to drive the scheduler of the Slave nodes.

b) Detecting communication and node failures

Besides synchronising the individual nodes on the network, TTC-SC1 is also responsible for detecting network and node failures. TTC-SC1 does this by having the Slave nodes return an “Acknowledgement” (Ack) message back to the Master node (Figure 1). This way, the Master node will know the status of all its Slaves after one TDMA round

With each Tick message, the Master node identifies which Slave should return an Ack message by embedding that particular Slave ID in the data stream. Only the Slave with this particular ID will send an Acknowledgement back to the Master. The Master will then check the status of this Slave, and send the next Tick message out with a new Slave ID.

Figure 2 illustrates an example of the TDMA round for a network with one Master and three Slaves, where Tick messages originate from the Master while Ack-X message is transmitted from Slave-X.

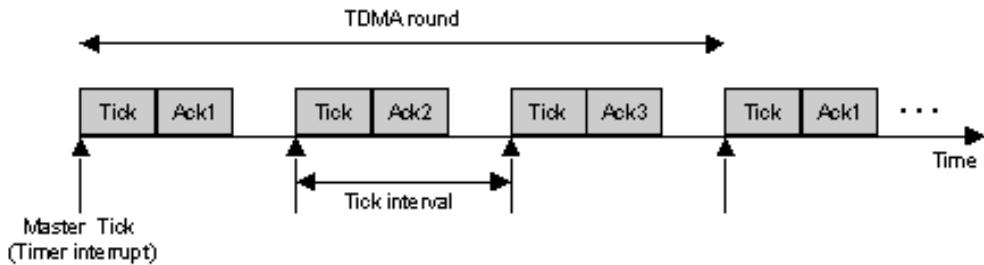


Figure 2: The round-robin TDMA configuration using TTC-SC1

c) Exchanging data between the nodes

CAN messages may be up to 8 bytes long. Only a limited overhead (typically up to 1 byte in each message) is required to support the TTC-SC1 protocol. This leaves around 7 bytes / message for data transfers between the Master and Slave nodes.

Please note that in this protocol, Slave-to-Slave communication is not permitted: all communication is directed via the Master node (through Tick and Ack messages).

d) Implementation

In the TTC-SC1 algorithm, only two CAN messages are exchanged within a Tick interval.

Typically, on the Master node, CAN Message Object (CMO) 0 will be configured to send the Tick messages. A second CMO will be configured to receive the Ack messages from all the Slaves.

Similarly, on the Slave nodes, CMO 0 will usually be configured to receive Tick messages and CMO 1 will be configured to transmit the Ack messages.

On the Master node, no CAN interrupts should be employed. On the Slave nodes, the CAN interface will be configured to generate a CAN interrupt upon receipt of a valid Tick message.

3. The TTC-SC2 algorithm

In some networks, the round-robin approach used to communicate with the Slave nodes in TTC-SC1 may not be efficient. For example, it may be that the Master node is required to check the status of a particular Slave node more frequently than the other Slaves. To do this, a modified version of the TTC-SC1 algorithm can be used: this is referred to here as “TTC-SC2”.

In the TTC-SC2 algorithm, the configuration of the TDMA round is assumed to be flexible. For example, for the similar system illustrated in Figure 2, it may be necessary that the status of Slave 1

is checked more frequently. Using TTC-SC2, the TDMA round illustrated in Figure 3 may be more suitable.

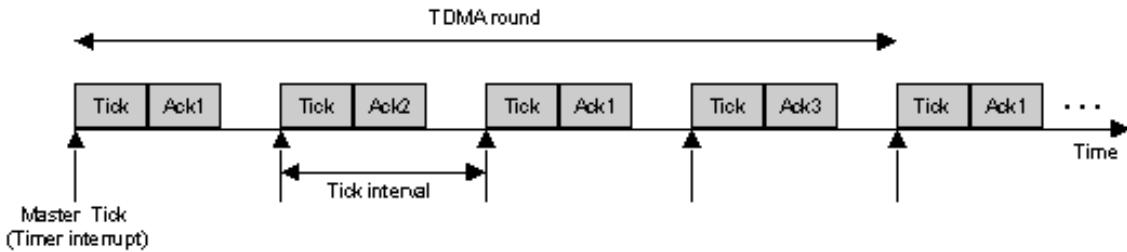


Figure 3: A different TDMA round for a four-node system using TTC-SC2

4. Problems with TTC-SC1 and TTC-SC2

We consider some of the drawbacks of TTC-SC1 and TTC-SC2 in this section.

a) Overview

The TTC-SC1 and TTC-SC2 algorithms are very simple and allow the creation of low-cost, time triggered, CAN-based networks with highly predictable patterns of behaviour. The algorithms are flexible and can also be used with a range of other network protocols, including RS485, without difficulty (see Pont, 2001).

However – inevitably – neither algorithm is a perfect match for all applications. In particular, when used with CAN, both TTC-SC1 and TTC-SC2 have the following limitations:

- i) Direct transfer of messages between Slave nodes is not supported, with the consequence that Slave-to-Slave transmission times are comparatively long.
- ii) To detect the failure of a given Slave node will take up to $(N+1) \times T$ seconds (where N is the number of Slave nodes, and T is the network Tick interval).
- iii) They suffer from task jitter, due to CAN bit stuffing.

We consider each of these issues in more detail in the remainder of this section.

b) Slave-to-Slave message latency

In TTC-SC1 and TTC-SC2, the design of the (TDMA) protocol means that all communication between Slave nodes is directed via the Master node. This makes bus traffic easy to predict, but increases the delays involved in Slave-to-Slave communications.

For example, if all the nodes on the network – including the Master – are sending eight-byte data messages, this makes message piggy-backing (see Tindell and Burns, 1994) impossible. Therefore, for each Tick, the Master needs to decide which data message should be relayed out to the Slaves. The technique used to relay the messages could be either priority-based or round-robin. Relaying the messages out to the Slaves will cause additional delay to the Slave-to-Slave message latency.

Additional delays such as these can sometimes have a detrimental impact on overall system performance. In control systems, for example, large delays between a sampling instant and a corresponding actuator response can seriously degrade system stability and performance (Sandfridson, 2000).

c) Failure detection time

In TTC-SC1, the Master node has to wait for a complete TDMA round before the status of all the Slaves on the network can be verified (and as the number of Slave nodes increases, the duration of the TDMA round becomes longer).

The worst-case failure detection time for the TTC-SC1 algorithm is given by Equation 1:

$$\text{Failure detection time} = (\text{Number of slaves} + 1) \times \text{Tick} - \text{CANTickmsg} \quad (1)$$

where *CANTickMsg* refers to the time taken for the Master node to transmit a Tick message

Take the system illustrated in Figure 4 as an example. Here, it would take up to four Tick intervals for the Master to detect a failure on Slave 1.

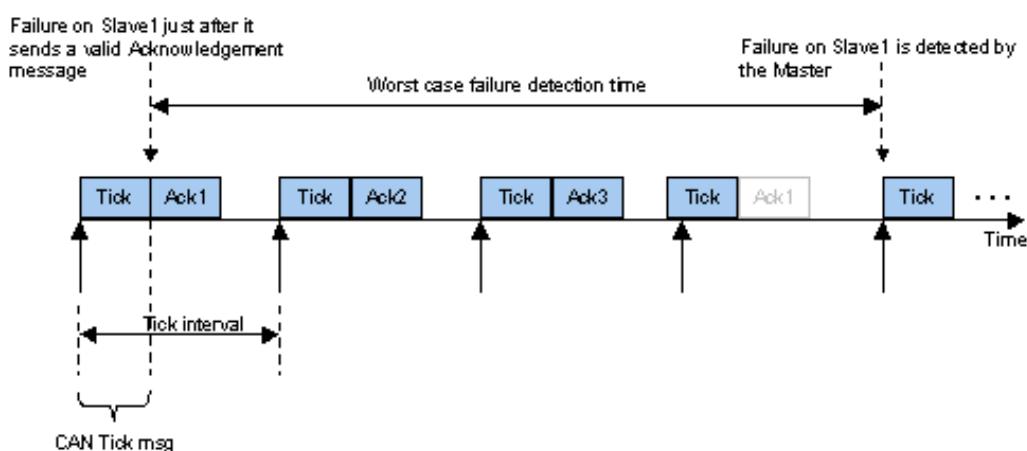


Figure 4: Failure detection time for TTC-SC1

This delay could be slightly reduced by using the TTC-SC2 algorithm, as illustrated in Figure 5. However, a failure of a “low priority” Slave (one that is sent Tick messages infrequently) - such as Slave2 - will still take a long time to be detected by the Master.

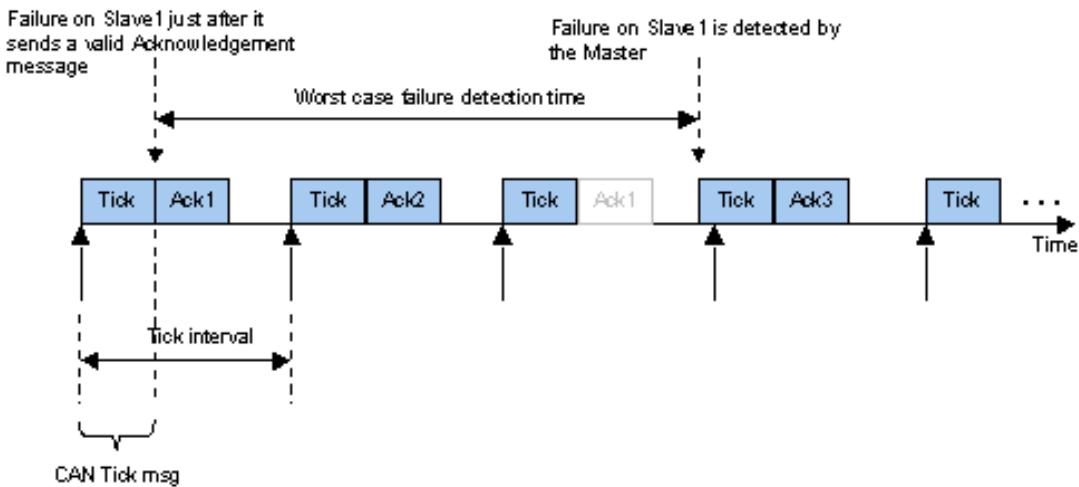


Figure 5: Failure detection time for TTC-SC2

d) Jitter due to bit-stuffing

The CAN protocol uses "Non Return to Zero" (NRZ) coding for bit representation. Under such a scheme, drift in the receiver's clock can occur when a long sequence of identical bits has been transmitted. Such a drift might, in turn, result in message corruption.

To avoid the possibility of such a scenario, the CAN communication protocol (at the physical level) employs a bit-stuffing mechanism which operates as follows: after five consecutive identical bits have been transmitted in a given frame, the sending node adds an additional bit, of the opposite polarity. All receiving nodes remove the ‘inserted’ bits to recover the original data (Farsi and Barbosa, 2000). Whilst providing an effective mechanism for clock synchronization in the CAN hardware, the bit-stuffing mechanism causes the frame length to become a complex function of the data contents.

It is useful to understand the level of message variation that this process may induce. When using (for example) 8-byte data and standard CAN identifiers, the minimum message length will be 111 bits (without bit stuffing) and the maximum message length will be 135 bits (with the worst-case level of bit stuffing): see Nolte et al., 2003 for details. At the maximum CAN baud rate (1 Mbit/sec), this translates to a possible variation in message lengths of 24 μ s.

These variations in message transmission times can have important implications in any real-time systems in which it is important to be able to predict event timing at the microsecond level. For example, in systems using TTC-SC1 and TTC-SC2, variations in the duration of “Tick” messages can have a significant impact on the levels of task jitter in the Slave nodes (see Nahas and Pont, 2004; Nahas et al., 2005).

This process is illustrated in Figure 6.

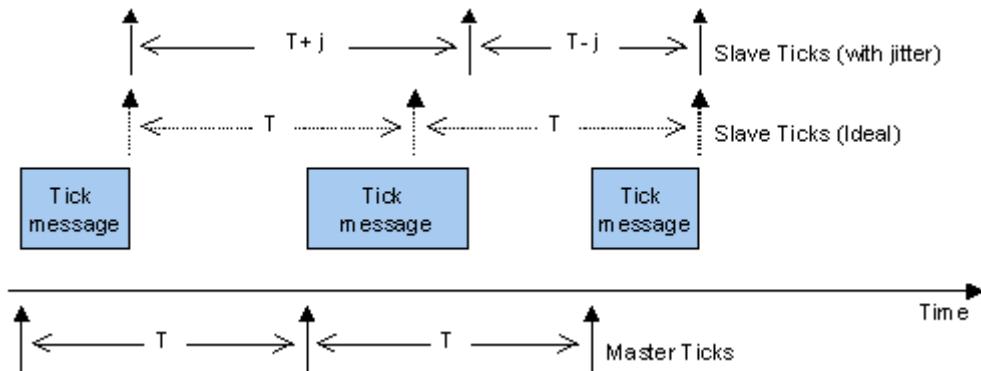


Figure 6: Impact of frame length on the timing of Slave Ticks in TTC-SC1 and TTC-SC2

5. TTC-SC3 algorithm

To resolve some of the shortcomings of the TTC-SC1 and TTC-SC2 algorithms, we developed TTC-SC3. An overview of this new protocol is presented in this section.

a) More than one Slave can reply in a Tick interval

In the TTC-SC3 algorithm, more than one Slave is allowed to reply within one Tick interval. Each time a Tick message is sent from the Master, an ID is also sent within the message (similar to TTC-SC1 and TTC-SC2). However, with TTC-SC3, it is possible to have more than one Slave reply to each ID. In this case, we let the CAN controller handle any message collisions. The Master node then checks that the appropriate Slaves have replied to a designated Tick ID before transmitting the next Tick message.

For example, let us assume a four-node system is to be implemented on a single CAN bus. Figure 7 shows the typical message exchange on the CAN network.

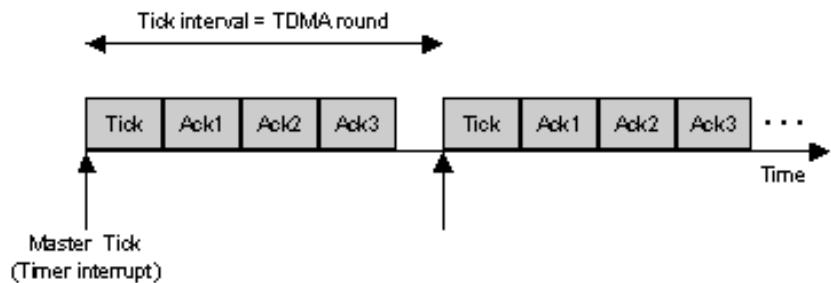


Figure 7: Tick and Acknowledgements for the TTC-SC3 algorithm

As an example of a more complicated configuration, suppose that we have a system with N Slaves, it is possible that all N Slaves reply within one Tick interval, or M Slaves reply within the first Tick interval and N-M Slaves reply in the second Tick interval; where $M < N$. In the latter instance, the TDMA round is extended across two tick intervals. The algorithm can be reconfigured such that the TDMA round is extended across more Tick intervals. The example in Figure 8 illustrates two possible examples of how a seven-node system can be configured.

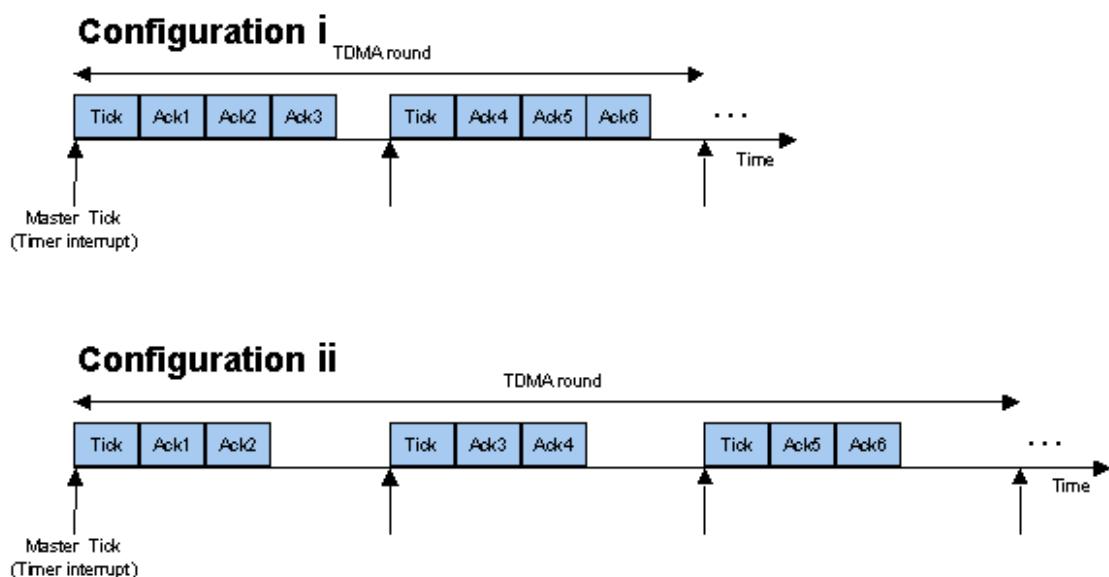


Figure 8: Two possible TDMA configurations using the TTC-SC3 algorithm for a seven-node system

b) All messages are broadcast

In the TTC-SC3 algorithm, all messages sent from the Slave nodes are broadcasted to all nodes (including the other Slaves).

c) Implementation

The broadcasting of Slave messages is made possible (on a CAN network) by assigning to each Slave node a unique CMO for its Ack message.

Please note that - as with TTC-SC1 and TTC-SC2 - these Ack messages should not trigger CAN interrupts.

6. The TTC-SC4 algorithm

TTC-SC4 is another S-C algorithm which builds on TTC-SC3. We describe TTC-SC4 in this section.

a) Tick only messages

When using TTC-SC4, the Master node is configured to send out “empty” Tick messages. These messages synchronise the network but – after the initialisation process – will not generally contain data. As such, the Master node simply generates the “heartbeat” of the network, but does no data processing. This approach allows the Tick message to have a fixed data content, which results in a constant CAN Tick message length. Thus, jitter caused by the Tick messages can be reduced.

Please note that, compared to TTC-SC1, TTC-SC2 and TTC-SC3 (where the Master can be involved in the system data processing), the total number of nodes on the system will – usually – increase by one.

Please also note that this is not the only way to reduce the jitter due to bit stuffing in CAN-based networks (e.g. see Nahas et al., 2005, Nahas and Pont, in press).

b) More than one Slave can reply in a Tick interval

As for TTC-SC3.

c) All messages are broadcast

As for TTC-SC3.

7. Evaluating the TTC-SC3 and TTC-SC4 algorithms

We describe the results of a small number of experiments carried out to illustrate the use of the TTC-SC3 and TTC-SC4 algorithms in this section.

a) Reduced failure detection time

TTC-SC3 and TTC-SC4 allows the Master node to quickly obtain Ack messages from the Slaves.

For example, Figure 9 illustrates an example where Slave 1 suffers a failure as soon as it has transmitted its Ack message. We assume we have all Slaves reply to each Tick message. As a result, the longest possible time that the Master node takes before a failure on the Slave node can be

detected is calculated using Equation 2. This duration is slightly less than two Tick intervals (which is significantly less than TTC-SC1 / TTC-SC2 in non-trivial networks).

$$\text{failure detection time} = 2 \times \text{Tick} - \text{CAN tick msg} \quad (2)$$

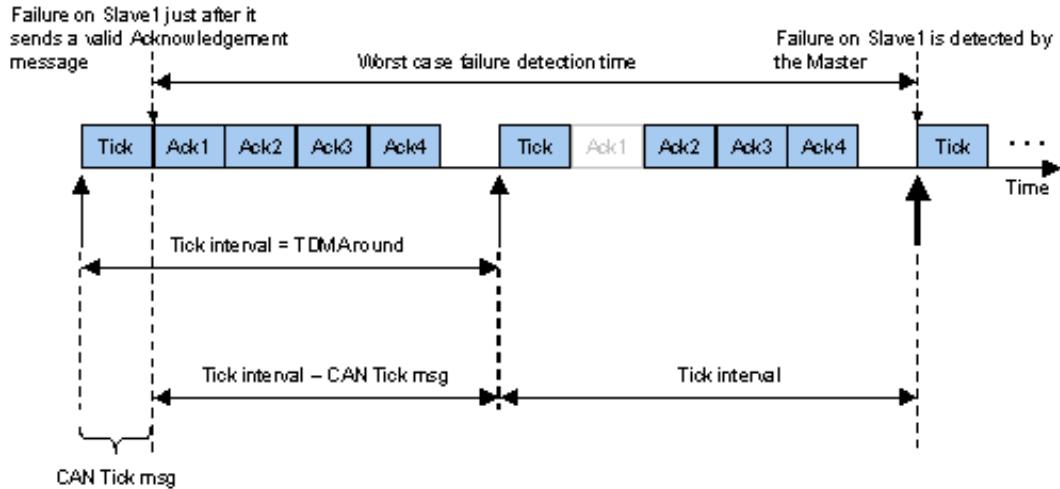


Figure 9: Calculating the worst-case Slave failure detection time of TTC-SC3/TTC-SC4

Of course, the precise failure detection time will depend very much on the way the TDMA round was scheduled. If the TDMA round is extended across more than one Tick interval, then Equation 3 is used to calculate the failure detection time. This is illustrated more clearly in Figure 10 where Slave 1 suffers a failure as soon as it has transmitted its Ack message.

$$\text{failure detection time} = \text{TDMAround} - \text{CAN tick msg} + \text{Tick} \quad (3)$$

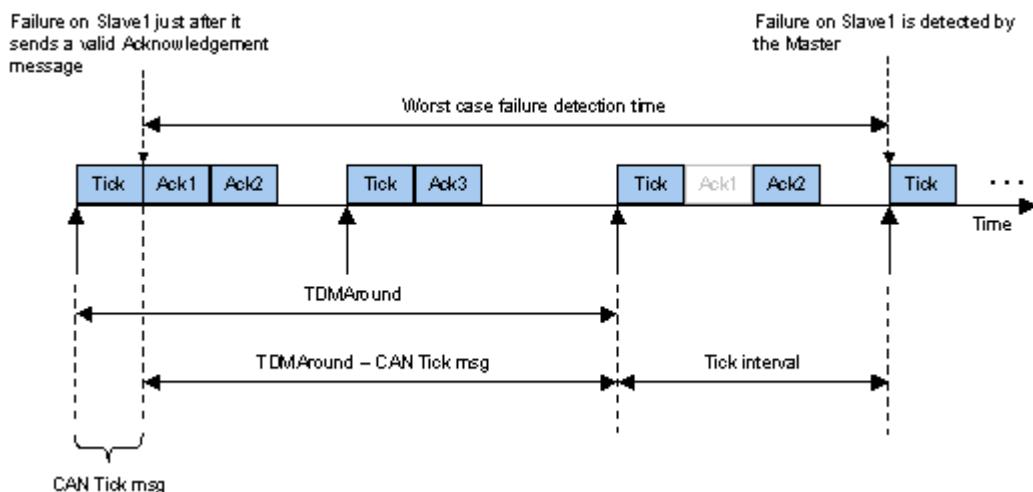


Figure 10: Calculating the worst-case Slave failure detection time for TTC-SC3 when the TDMA round extends across more than one tick interval

b) Reduced Slave-to-Slave message latency

When compared with TTC-SC1 and TTC-SC2, the latency of message transmission between Slaves is reduced in both TTC-SC3 and TTC-SC4.

To illustrate this, a comparison of the implementation between TTC-SC1, TTC-SC3 and TTC-SC4 was carried out (see Figure 11)[†].

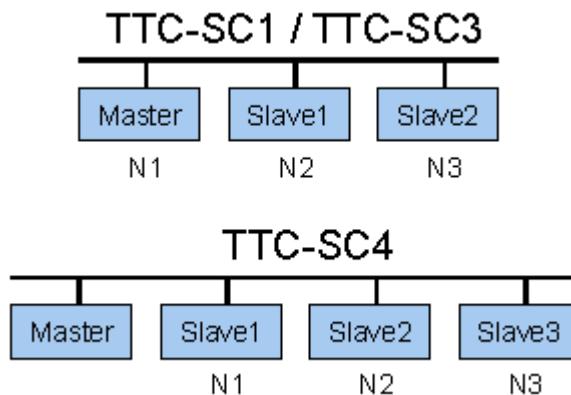


Figure 11: Comparing TTC-SC1, TTC-SC3 and TTC-SC4

Referring to Figure 11, each of the main nodes (N1, N2, N3) executes several periodic tasks which exchange data around the network.

We made our measurements using a (dummy) control system that involves all three nodes.

Specifically, N1 had a control task that issued a periodic request for data from N2 and N3. N2 and N3 each sent data back to N1 as soon as they received the request. N1 then produced a control value. We measured the interval between the data request (on N1) and the completion of the control value calculation on this node.

Table 1 shows the control delay for the different versions of this system. The results indicate that the control delay when using TTC-SC3 and TTC-SC4 was shorter than that obtained using TTC-SC1. In addition, the variation in the control delay for the TTC-SC3 and TTC-SC4 implementations was insignificant when compared to the corresponding TTC-SC1 results.

[†] Note that, in both the TTC-SC3 and TTC-SC4 implementations, all Slaves replied to each Tick message.

Table 1: Comparison of measured control delays for TTC-SC1, TTC-SC3 and TTC-SC4

	TTC-SC1	TTC-SC3	TTC-SC4
Minimum (μs)	4013	3003	4008
Maximum (μs)	6012	3003	4022
Average (μs)	5012	3003	4012
Max-Min(μs)	1999	0	14
Std. Deviation	0.000821	0.000095	0.000126

c) Jitter due to bit-stuffing

With the TTC-SC3 algorithm, the level of jitter due to bit stuffing remains the same as that obtained using TTC-SC1 and TTC-SC2.

To illustrate the reduction in jitter obtained with TTC-SC4, two versions of a three-node system (each with 1 Master and 2 Slaves) were implemented using TTC-SC3 and TTC-SC4. A CAN baudrate of 500kbits/sec was used in each system. For TTC-SC3, the content of the Tick data messages were periodically rotated among three different values (the Tick messages were “empty” when using TTC-SC4).

Measurements of the interval between sending the Tick message on the Master and receiving the message on the Slaves were carried out (for both the Slave nodes). The results obtained (shown in Table 2) indicate that the TTC-SC3 algorithm had higher jitter compared to TTC-SC4. The results also indicate that the maximum jitter for the TTC-SC4 algorithm on the CAN message transmission time was +/- 1 bit time (at 500 kbits/sec, one bit time is 2μs). This is in line with the results obtained previously (Nahas and Pont, 2004) for minimal jitter levels in CAN messages.

Table 2: Message transmission times for TTC-SC3 and TTC-SC4

	TTC-SC3		TTC-SC4	
	Slave1	Slave2	Slave1	Slave2
Minimum (μs)	186	186	122	122
Maximum (μs)	200	200	126	126
Average (μs)	192	192	124	124
Max-Min (μs)	14	14	4	4
Std. Deviation	0.000005	0.000005	0.000001	0.000001

8. Discussion

Although we have shown that TTC-SC3 and TTC-SC4 algorithms have several benefits (when compared to TTC-SC1 and TTC-SC2), there are also some drawbacks. We consider these here.

a) Number of network nodes

Using the TTC-SC4 algorithm, the total number of nodes required in each network will be increased by one. This is because a separate Master node is required to function as the network synchroniser. This will obviously add to the system cost.

b) Tick interval

In most cases, TTC-SC3 and TTC-SC4 require all the Slaves to reply within one Tick interval. As such, the following relationship must hold:

$$\text{Tick interval} > \text{Time take for all Ack messages to be received by the Master}$$

That is the Tick interval of the system is related to the number of Slaves, and the size of each of the Slave's Ack messages. This may be a significant drawback in networks requiring a low Tick interval.

c) Portability

The TTC-SC3 and TTC-SC4 algorithms cannot be easily ported to other network protocols, such as RS485. This is due to the fact that the algorithms require more than one Slave to reply to each Tick message. Most CAN controllers can deal with this because they can handle message conflicts and support multiple receive buffers. By contrast, RS485 has one receive buffer and no direct support (in hardware) for handling message conflicts.

Please note:

- Some CAN controllers (such as MCP2510) do not have 15 message receive buffers. Overall, this reduces the portability of the TTC-SC3 and TTC-SC4 algorithms when compared with TTC-SC1 and TTC-SC2.
- The number of nodes connected to the CAN bus will depend on the number of message objects supported by the CAN hardware (in most cases, this will be up to 15: see, for example, Siemens, 1996). If more than 15 nodes need to be connected, a second CAN controller can be used. Many microcontrollers now have two or more on-chip CAN controllers and can support such requirements. However, such an arrangement further reduces portability (and further adds to costs).

d) Babbling Slaves

TTC-SC3 and TTC-SC4 will – typically - rely on all the Slaves to send an Acknowledgement back to the Master within one Tick interval. If one of the Slaves have a “babbling idiot” problem (Kopetz, 1998) or there is constant message retransmission from one of the Slaves, then lower priority CAN messages from other Slaves will not have access to the network. This will cause the Master to “think” that the Slaves with the lower priority CAN messages are faulty when in-fact it is only a single node that is causing the problem.[‡]

To reduce the impact of this problem, the CAN controller can be configured to disable its automatic message retransmission. However, this is not a complete solution, and the feature is only available on certain CAN implementations (such as that used in the Infineon XC167).

9. Conclusions

This study has investigated the use of shared-clock (S-C) algorithms with CAN-based systems. Specifically, we have looked at two new S-C algorithms (TTC-SC3 and TTC-SC4) which are – when compared with TTC-SC1 and TTC-SC2 - intended to reduce Slave-to-Slave message transmission times, reduce failure detection times and (in the case of TTC-SC4) reduce task jitter.

While no single algorithm is ever likely to provide a perfect solution to all networking problems, the results and discussion presented here suggest that TTC-SC3 and TTC-SC4 are very useful additions to the range of S-C algorithms.

References

- Bosch, R.G., 1991. CAN Specification. Version 2.0, Robert Bosch GmbH, Postfach 50, D-7000 Stuttgart 1, Germany.
- Farsi, M. and Barbosa, M., 2000. CANopen Implementations: Applications to Industrial Networks. Research Studies Press Ltd.
- Fredriksson, L.B., 1994. Controller Area Networks and the Protocol CAN for Machine Control Systems. Mechatronics, 4(2): 159-192.
- Infineon, 2004. Connecting C166 and C500 Microcontroller to CAN, Infineon Technologies.
- Kopetz, H., 1998. A comparison of CAN and TTP, 15th IFAC Workshop on Distributed Computer Control Systems, Como, Italy.
- Leen, G. and Heffernan, D., 2002. TTCAN; A New Time-Triggered Controller Area Network. Microprocessors and Microsystems, 26(2): 77-94.

[‡] Although TTC-SC1 and TTC-SC2 also suffer from problems caused by “babbling idiot” failures, the impact is more severe (and harder to guard against) in TTC-SC3 and TTC-SC4, due to the smaller error margins.

- Misbahuddin, S. and Al-Holou, N., 2003. Efficient Data Communication Techniques for Controller Area Network (CAN) Protocol, ACS/IEEE International Conference on Computer Systems and Applications, Tunis, Tunisia.
- Nahas, M. and Pont, M.J., 2004. Reducing Task Jitter in Shared-Clock Embedded Systems using CAN. In: A. Koelmans, A. Bystrov and M.J. Pont (Editors), Proceedings of the UK Embedded Forum, Birmingham, UK.
- Nahas, M. and Pont, M.J., in press. Using XOR Operations to Reduce Variations in the Transmission Time of CAN Messages. In: A. Koelmans, A. Bystrov and M.J. Pont (Editors), Proceedings of the 2nd UK Embedded Forum, Brimingham, UK.
- Nahas, M., Short, M.J. and Pont, M.J., 2005. The Impact of Bit Stuffing on the Real-Time Performance of a Distributed Control System, 10th international CAN Conference, Rome, Italy.
- Nolte, T., Hansson, H., Norström, C. and Punnekkat, S., 2003. Using Bit-Stuffing Distributions in CAN Analysis, IEEE/IEE Real-Time EMbedded Systems Workshop, (Satellite of the IEEE Real-Time Systems Symposium) London.
- Pazul, K., 1999. Controller Area Network (CAN) Basics, Microchip Technology Inc.
- Philips, 1996. PCA82C250/251 CAN Tranceiver, Philips Semiconductors.
- Philips, 2004. SJA1000 Stand-Alone CAN Controller.
- Pont, M.J., 2001. Patterns For Time Triggered Embedded Systems. Addison Wesley.
- Pont, M.J., 2003. Supporting the Development of Time-Triggered Co-operatively Scheduled (TTCS) Embedded Software Using Design Patterns. *Informatica*, 27(1): 81-88.
- Pont, M.J. and Banner, M.P., 2004. Designing Embedded Systems using Patterns; A Case Study. *Journal of Systems and Software*, 71(3): 201-213.
- Sandfridson, M., 2000. Timing Problem in Distributed Real-Time Computer Control Problems. ISSN 1400-1179, Mechatronics Lab, Department of Machine Design, Royal Institute of Technology, KTH, Stockholm, Sweden.
- Sevillano, J.L., Pascual, A., Jimenez, G. and Civit-Balcells, 1998. Analysis of Channel Utilization for Controller Area Networks. *Computer Communications*, 21(16): 1446-1451.
- Siemens, 1996. C167 Derivatives - User's manual, Version 2.0.
- Siemens, 1997. Proceedings of the European Pattern Languages of Programming Conference.
- Thomesse, J.P., 1998. A Review of the Fieldbuses. *Annual Reviews in Control*, 22: 35-45.
- Tindell, K. and Burns, A., 1994. Guaranteed Message Latencies for Distributed Safety-Critical Hard Real-Time Control Networks. YCS 229, Real-Time Syetems Research Group, University of York.
- Zuberi, K.M. and Shin, K.G., 1995. Non-Preemptive Scheduling of Messages on Controller Area Network for Real-Time Control Applications, Proceedings of the First IEEE Real-Time Technology and Applications Symposium, Chicago, USA, pp. 240-249.

Comparing the performance and resource requirements of “PID” and “LQR” algorithms when used in a practical embedded control system: A pilot study

Ricardo Bautista, Michael J. Pont and Tim Edwards

*Embedded systems laboratory,
University of Leicester, University Road, Leicester LE1 7RH, UK.*

Abstract

We have previously discussed the ways in which an inverted pendulum may be used as an effective testbed by researchers who wish to evaluate different design options for reliable embedded systems. The present paper employs an extended and modified version of this testbed in order to explore the impact of different control algorithms on the system performance and resource requirements. The particular focus of the paper is on “Proportional Integral Differential” (PID) and “Linear Quadratic Regulator” (LQR) control algorithms.

Acknowledgements

This work is supported by the National Council for Technology Education (COSNET), México. Work on this paper was carried out while MJP was on Study Leave from the University of Leicester. The authors would like to thank Alan Wale and Julian Jones (University of Leicester) for their invaluable assistance in the construction of the pendulum test bed described in this paper.

1. Introduction

We have previously discussed the ways in which an inverted pendulum may be used as an effective testbed by researchers who wish to explore different design options for reliable embedded systems (Edwards et al., 2004). One reason for selecting such a testbed is that this system is inherently unstable and provides a demanding control task, with a simple – clearly visible – indication of performance. In addition, as this is a well-studied control problem (e.g. Dorf and Bishop, 2001; Franklin et al., 2002; Ogata, 1995), a great deal of useful information is available to support the development of such a testbed (e.g. Friedland, 1998; Lundberg, 2003; Gaixin, 2002).

We have described a prototype pendulum testbed in a previous paper (Edwards et al., 2004). The present paper employs an extended and modified version of this testbed in order to begin exploring the impact of different control algorithms on the system performance and resource requirements. The particular focus of the paper is on “Proportional Integral Differential” (PID) and “Linear Quadratic Regulator” (LQR) control algorithms.

There have been many previous studies which have compared the performance of different control algorithms (e.g. Ogata, 2002; Heng-Ming, 1994; Varsek et al, 1993; Natale et al., 2004). However, there have been very few cases in which “real hardware” has been used in such comparisons: indeed, in most cases, the comparisons have been conducted using simulation environments, including MATLAB and similar tools (e.g. see Campa, 2000). Similarly, most studies fail to consider the implementation costs (in terms of, for example, CPU and memory requirements) of different control algorithms: such costs are a key factor when different options for resource-constrained embedded control systems are considered.

In this paper, we consider the practical implementation of PID and LQR control algorithms on a small, low-cost hardware platform. Specifically, we use an LPC2129 processor (Philips LPC2000 family). During the course of the paper, we consider and compare the performance and resource requirements of these two different control algorithms when used to control an inverted pendulum.

We begin by describing the testbed used in this study.

2. The test-bed

As noted in the introduction, the focus of this paper is on an inverted-pendulum test-bed, driven by a DC motor.

The inverted pendulum has two equilibrium points which are stable in absence of any motor voltage. The unstable equilibrium point represents the upright pendulum and is the focus of the analysis presented in the next section.

Figure 1 shows the basic free body representation of the inverted pendulum.

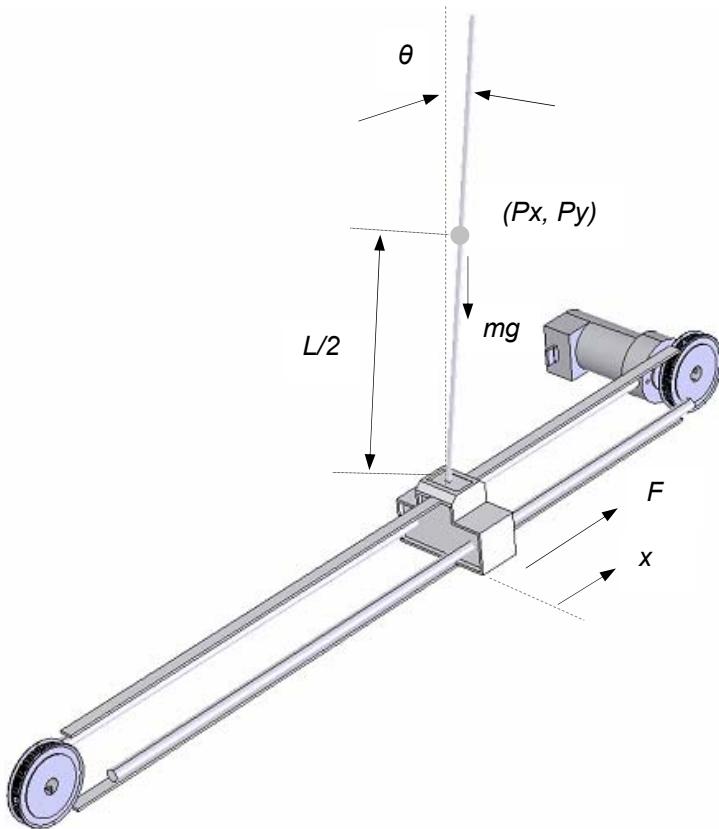


Figure 1: Diagram of the inverted pendulum and its variables

Note that we wish not only to hold the pendulum upright, but we also wish to maintain the cart at a pre-defined “set point” (SP) position on the track (Figure 1). As a result, we need to measure both the rod angle and the cart position. In both cases we use incremental encoder sensors to achieve this.

Figure 2 shows the motor assembly used to drive the pendulum, along with the gearbox and integrated encoder.

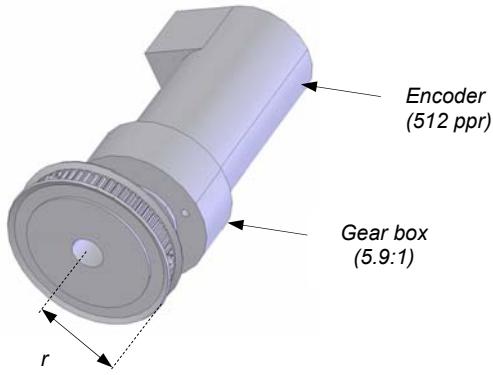


Figure 2: The pendulum is driven by a brushed DC motor with integrated gearbox and encoder.

Equation 1 defines the relation between the pulses given for the encoder and the position of the pendulum cart:

$$\beta_1 = \frac{2\pi \cdot r}{\text{Res_Enc} \cdot \text{Gear_Box}} \quad \underline{\text{Equation 1}}$$

where r is the radius of the pulley (in this case 36 mm), Res_Enc is the resolution of the encoder (in this case 512 “pulses per revolution”, ppr), and Gear_Box is the reduction ratio of the gears in the motor (in this case 5.9).

For the rotational sensor in the base of the rod we use another encoder. The resolution is given in Equation 2:

$$\beta_2 = \frac{2\pi}{\text{Res_Enc}} \quad \underline{\text{Equation 2}}$$

We also – of course – need to drive the motor. We use a (10-bit) PWM output from the microcontroller plus associated driver hardware (see Figure 3).

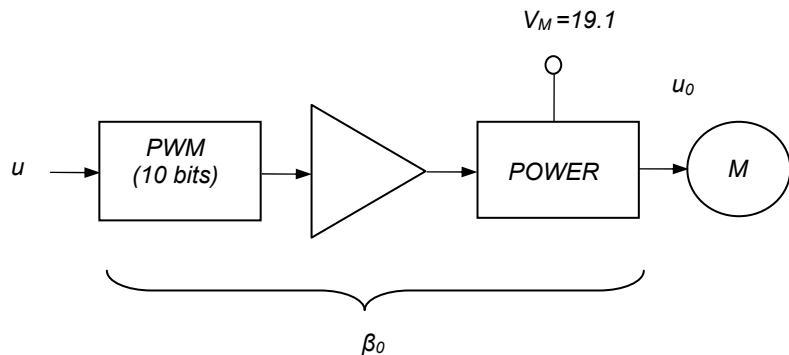


Figure 3: The interface between the PWM output from the microcontroller and the motor itself.

For this system $\beta_0 = V_M/2^{10}$.

Based on appropriate empirical tests, the real parameters are listed in Table 1.

Parameter	Description
$b0=19.1/1024$ Volts	Dimension factor of the voltage
$b1=6.8327e-5$ mts	Dimension factor of the linear position (related to encoder, pulley and gear box)
$b2=1.57e-3$ rad	Dimension factor of the angular position (related to the rod sensor)

Table 1: Real motor and sensor parameters (see also Table 2).

3. Modeling the test-bed

Using Euler-Lagrange representation (Spong, 1989) we can obtain a (nonlinear) mathematical model of the testbed: such an approach considers the kinetic and potential energy of the mechanical chain and obtains a model of the torque at each link. Using such an approach we aim to control the system by adding external force (F) to the system to make it stable at the upright equilibrium point:

$$(M_c + m)\ddot{x} + ml \cos \theta \ddot{\theta} - ml\dot{\theta}^2 \sin \theta + \mu_x \dot{x} = F \quad \text{Equation 3}$$

$$ml \cos \theta \ddot{x} + (J + ml^2)\ddot{\theta} - mgl \sin \theta + \mu_\theta \dot{\theta} = 0$$

Our aim is to maintain the rod close to the unstable equilibrium point: this represents angle 0, at which point:

$$\sin \theta \approx \theta, \quad \cos \theta \approx 1, \quad \dot{\theta}^2 \sin \theta \approx 0, \quad \mu_x \dot{x} \approx 0, \quad \mu_\theta \dot{\theta} = 0$$

We thereby obtain:

$$(M_c + m)\ddot{x} + ml\ddot{\theta} = F \quad \text{Equation 4}$$

$$ml\ddot{x} + (J + ml^2)\ddot{\theta} - mgl\theta = 0$$

As previously noted, the system is moved by a DC motor. Our model is based on force as an input variable, so that we have to find the dynamic relation between the input voltage u_0 and force F :

$$F = -\frac{J_M}{r^2}\ddot{x} - \frac{\alpha_2}{r}\dot{x} + \frac{\alpha_1}{r}u_0 = -M_0\ddot{x} - c_2\dot{x} + c_1u_0 \quad \text{Equation 5}$$

where:

- J_M is the inertia of the rotor of the DC motor
- r is the radius of the pulley
- α_1 and α_2 are factors that depend of the electric characteristics of the DC motor

Substituting Equation 5 in Equation 4, we have:

$$(M+m)x + ml\ddot{\theta} + c_2\dot{x} = c_1u_0 \\ ml\ddot{x} + (J+ml^2)\ddot{\theta} - mgl\theta = 0 \quad \underline{\text{Equation 6}}$$

Equation 6 shows the differential equation of the pendulum. In this, the states x and θ are variables that are expressed in meters and radians respectively.

For space state representation we choose the next vector state:

$$\chi = \begin{bmatrix} x \\ \theta \\ \dot{x} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} \quad \underline{\text{Equation 7}}$$

Simplifying, by using state equations, we have:

$$\tau_0 \frac{d}{dt} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & -f_1 & -f_2 & 0 \\ 0 & f_3 & f_4 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 1 \\ -g_4 \end{bmatrix} u \quad \underline{\text{Equation 8}}$$

where:

$$\tau_0^2 = \frac{H\beta_1}{(J+ml^2)c_1\beta_0}, \quad f_1 = \frac{m^2l^2g\beta_2}{(J+ml^2)c_1\beta_0}, \quad f_2 = \frac{c_2\beta_1}{c_1\beta_0\tau_0} \\ f_3 = \frac{(M+m)mgl\beta_1}{(J+ml^2)c_1\beta_0}, \quad g_4 = \frac{ml\beta_1}{(J+ml^2)\beta_2}, \quad f_4 = f_2 \cdot g_4$$

$$H = (M+m)J + M \cdot ml^2$$

The input vector is then:

$$y = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} \quad \text{Equation 9}$$

The parameters characterised are shown in Table 2.

Parameter	Description
<i>mc=0.27 Kilograms.</i>	Mass of the cart (parameter obtain experimentally that contain the effect due to the inertia of the motor)
<i>m=0.05 Kilograms</i>	Mass of the rod
<i>l=0.305 meters.</i>	Length of the pendulum between 2 (Modelled in the mass centre)
<i>j=1.2e-3 Kg mts</i>	Inertia of the pendulum
<i>c1=4.48 Nw/volts</i>	Coefficient 1 related to the dynamic of the motor
<i>c2=38.14 Kgm/s</i>	Coefficient 2 related to the dynamic of the motor

Table 2: Real parameters of the testbed (see also Table 1).

4. Design of an LQR controller

Having a model of the pendulum system, we are now able to design a suitable controller. As noted in the introduction, in the first approach we will employ a linear quadratic regulator (LQR). This control algorithm is commonly used for multivariable linear systems (e.g. see Chen, 1999) this is appropriate in our case since we aim not only to maintain the pendulum in an upright position but also to control the position of the cart (usually at the centre of the track).

LQR is a control algorithm used to control multivariable linear time invariant (LTI) systems (Ogata, 1995). This method may be used not only to control the plant but also to minimise the cost function, which is related to the performance of the system in the closed loop situation. In Equation 10 we have Q and r that are positive-definite Hermitian matrices. The second term of this equation represents the expenditure of the energy of the control signal. The matrices Q and r determine the relative importance of the error and the expenditure of this energy, however the formulation assumes that vector $u(t)$ is unconstrained, consequently we

need to make a trade-off between the energy used and the performance of the system in a practical implementation.

We introduce vector x that consists of five states (position of the cart x_1 , velocity of the cart x_3 , position of the rod x_2 , velocity of the rod x_4 and the compensate delay state w_0) multiplied by the gain vector (k) to provide the input voltage. We therefore use the control law $u=-kx$ in order to minimize a quadratic cost function of the form:

$$J = \frac{1}{2} \sum_{k=0}^{\infty} \{x(k)^T Q x(k) + r u(k)^2\} \quad \text{Equation 10}$$

For this application, minimising the energy used to control the states of the system means applying the minimum voltage to the motor in order to reduce the electrical current. Overall, we aim to ensure that the controller should use minimal energy to maintain the pendulum in the upright position and - at the same time – which the cart must reach the required (set point) position.

The relationship between the elements in the diagonal of the Q matrix represents the relative importance of the four terms in Equation 7 to the control law. In this case, the same value was chosen in each “slot” because all of the terms were considered to be of equal importance. As far as r is concerned, altering the value can change the speed of response of the controlled system (larger values of r make the response slower).

The values shown in Equation 11 were chosen using the simulator. The aim was to reach the goal “as quickly as possible” without exceeding the maximum actuator voltage ($u(t) < V_{max} = 19.1$ volts).

$$Q = \begin{bmatrix} 50 & 0 & 0 & 0 \\ 0 & 50 & 0 & 0 \\ 0 & 0 & 50 & 0 \\ 0 & 0 & 0 & 50 \end{bmatrix} \quad \text{Equation 11}$$

$$r = 1000$$

Based on the discrete model and using a LQR controller, the gain vector for the linearised system is:

$$k = (-0.2158 \quad -6.9694 \quad -9.8645 \quad -89.3738)$$

Equation 12

$$k_0 = 0.0702$$

A simulation of the system (Figure 4) shows that the cart takes around 1.6 seconds to reach “position 4000”, which is equivalent to a position 27.3 cm on the linear axis of the testbed. In addition, the rod is stable in the equilibrium point (upright) when the cart reaches this position.

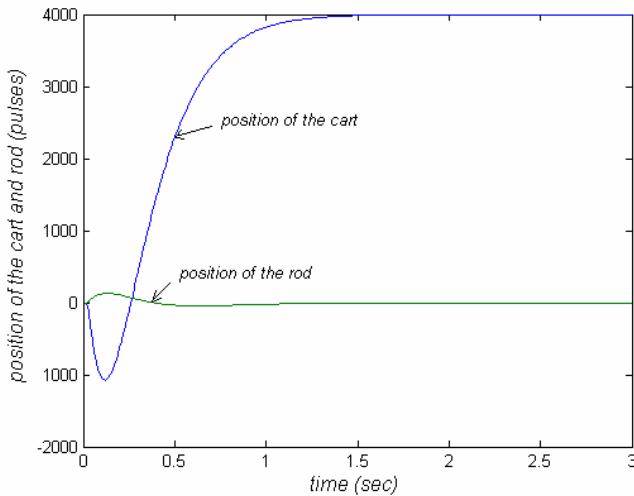


Figure 4: Simulation of the testbed using an LQR controller

5. Implementation of an LQR controller

In our implementation of the LQR algorithm we measured two states from the system (the cart position and the rod angle). We estimated two states (velocity of the cart, velocity of the rod) using “observers”. The gain for the compensation delay state was determined off line (as outlined below).

To determine the velocity of the rod, we measured its position periodically (at known time intervals – in this case, every 4 ms) and used a simple “pseudo derivative” (first order derivative) to approximate the velocity. We then applied a low-pass filter to reduce the impact of the high frequency harmonics generated in this approximation: in this case, we employed a moving-average filter (Hamming 1989):

$$\tilde{v}_f = \alpha \tilde{v}_f + (1 - \alpha) \tilde{v} \quad \underline{\text{Equation 13}}$$

where:

\tilde{v}_f Speed after the filter process

α Filter coefficient

The coefficient of the filter (α) was set to 0.95 in this case.

An identical procedure was followed when calculating the cart position.

In order to determine the gain for the compensate delay state; we used the function “dlqr” from MATLAB (based on the discrete model of the pendulum).

Figure 4 shows the step response of the resulting implementation.

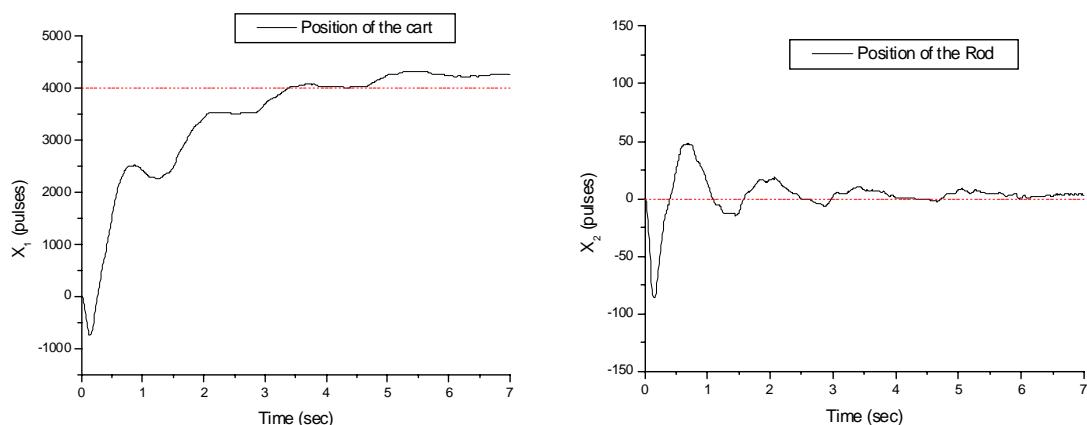


Figure 5: System under LQR controller using an step input (Set Point = “4000 pulses”)

6. Design and implementation of a PID controller

The multivariable PID algorithm used for this application employed a PID action for the control of the cart position and a PD action to control the rod position*. As with the LQR implementation, we estimated states in the microcontroller to generate the control law. In this case, the estimated states were: velocity of the rod, velocity of the cart, and integral of the error between cart and set point.

A numerical simulation of the PID controller is shown in Figure 6. As we can see, the angular position goal is reached in less time than with LQR; however, the cart takes more time to reach the required set point. Note that the integral component in the error of the cart was incorporated to eliminate the steady state due to friction, but dynamical and static frictions were not considered in this numerical simulation.

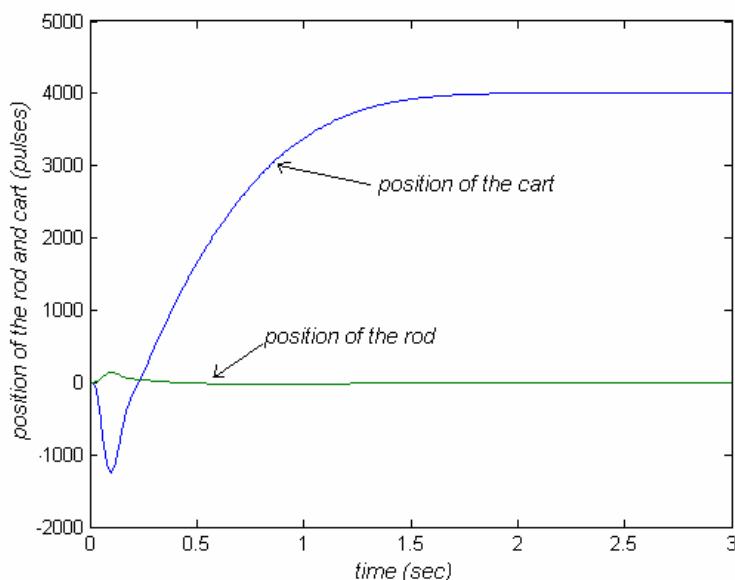


Figure 6: Simulation of the testbed by using PID controller

The experiments using these components illustrate the real effects caused by frictional torque (due to the bearings that hold the rod) and linear frictional force (due to the cart structure). In addition the parametric uncertainty means that the numerical simulation does not represent the real system. Overall, only the real implementation can allow us to determine how far the mathematical model differs from the real physical system.

* Integral action in the rod can cause inherent instability (Chen, 1999).

The program of the PID algorithm is based on data acquisition and digital estimators previously described in Section 5. The only significant difference is in the control law itself.

The following gains were determined by trial and error:

PID for the cart

```
kpc=-600; // proportional constant related to the cart error position
kdc=-1813; // derivative constant related to the cart error position (velocity of the cart)
kic=100; // integral action to the error of the cart position
```

PD for the rod

```
kpr=3848.8; // proportional constant related to the rod error position
kdr=784; // derivative constant related to the rod error position (velocity of the rod)
```

Using a step input we can compare the differences between the simulated and measured responses for the PID controller (see Figure 7 and compare with Figure 6). Note that the same gains were used in the simulation and actual implementation.

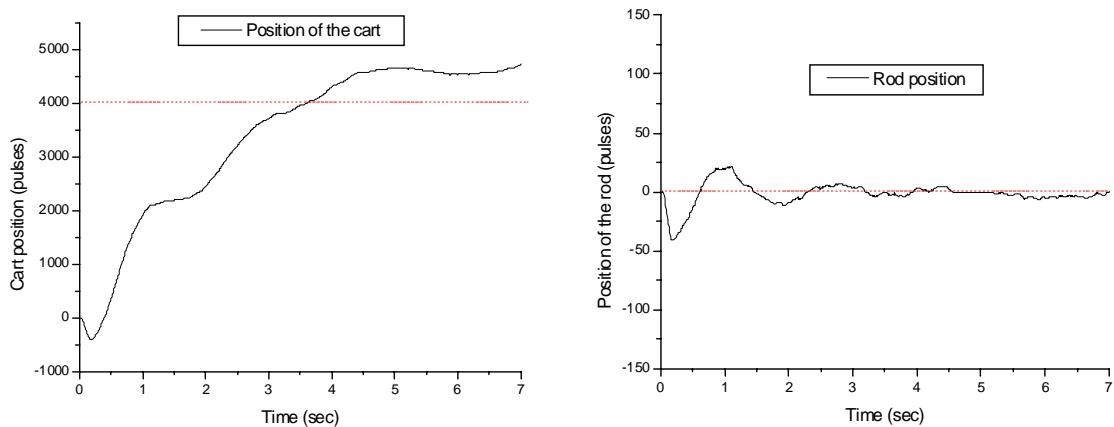


Figure 7: System under PID controller using a step input (SP=4000)

7. Comparing the simulations and system implementations

In this simple comparison, we consider how easy it is to predict the step response behavior for each system based on a (MATLAB) simulation. The data presented in the tables are extracted from Figure 5 and Figure 7.

	<i>Simulation</i>		<i>Testbed</i>	
Feature	For the cart:	For the rod:	For the cart:	For the rod:
Undershoot	-68.32 mm	0 rad	-47.8 mm	0.092 rad
Overshot	0 mm	0 mm	0 mm	0.057 rad
Steady state error	0 mm	0 mm	20.49 mm	0.01157 rad
Rising time	1.5 sec	0.6 sec	99% at 4.5 sec	99% at 6.5 sec

Table 3: Features of the system using LQR with a SP=4000

	<i>Simulation</i>		<i>Testbed</i>	
Feature	For the cart:	For the rod:	For the cart:	For the rod:
Undershoot	81.9 mm	0 mm	27.3 mm	0.046 rad
Overshot	0 mm	0 mm	34.1 mm	0.037 rad
Steady state error	0 mm	0 mm	0 mm t=20sec	0.0092 rad
Rising time	2 sec	1.25 sec	12.5% up at 6 sec	5 sec

Table 4: Features of the system using PID with a SP=4000

Please note that – based on this comparison - LQR seems to be both more accurate and faster than PID, its only weakness being an inability to reach the set point (without an offset). However, in situations with high frictional forces (higher than in the present testbed), the ability of PID control to meet the set point may be particularly advantageous.

8. Comparing the basic performance of the two controllers

We describe how the performance of the two controllers was compared in this section.

a) Overview

One effective way of comparing the performance of different control algorithms is to consider the response of the controlled system to a dynamic set point (e.g. see Reyes 1996). In the comparisons described in this section, we used a set point (SP) specified as follows:

$$SP = A \sin(\omega t) \quad \text{Equation 14}$$

The initial frequency used was 1 Hz and this was reduced to 0.5 Hz, 0.25 Hz, 0.0125 Hz, 0.0625 Hz, and finally 0.03125 Hz. Identical tests were performed for the LQR and PID controllers and the results are presented in this section.

Overall, if our key goal is to maintain the rod close to the equilibrium point then PID control is a more appropriate choice in this system. However, if we wish to maintain rod and cart position, then LQR control may be a more appropriate choice.

b) 1 Hz

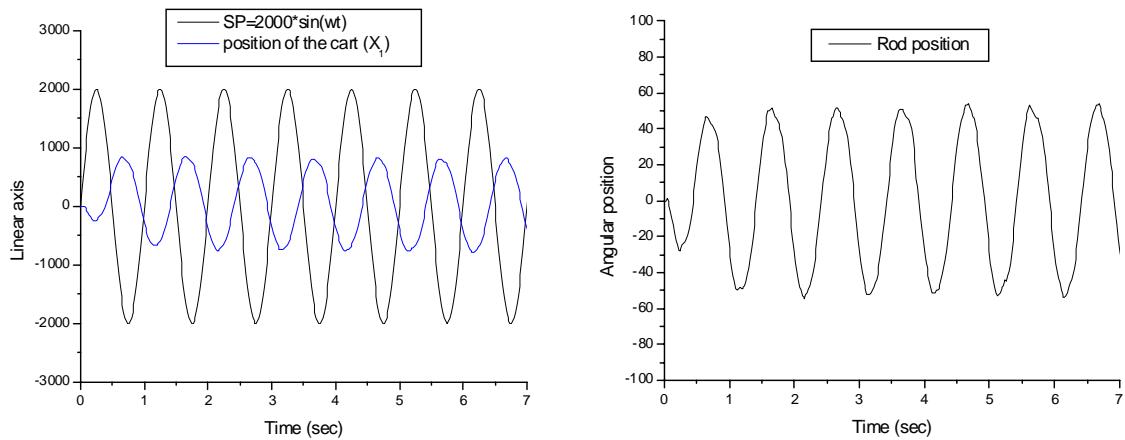


Figure 8: LQR Tracking control using a dynamical $SP=2000 \sin(\omega t)$, $F = 1\text{Hz}$

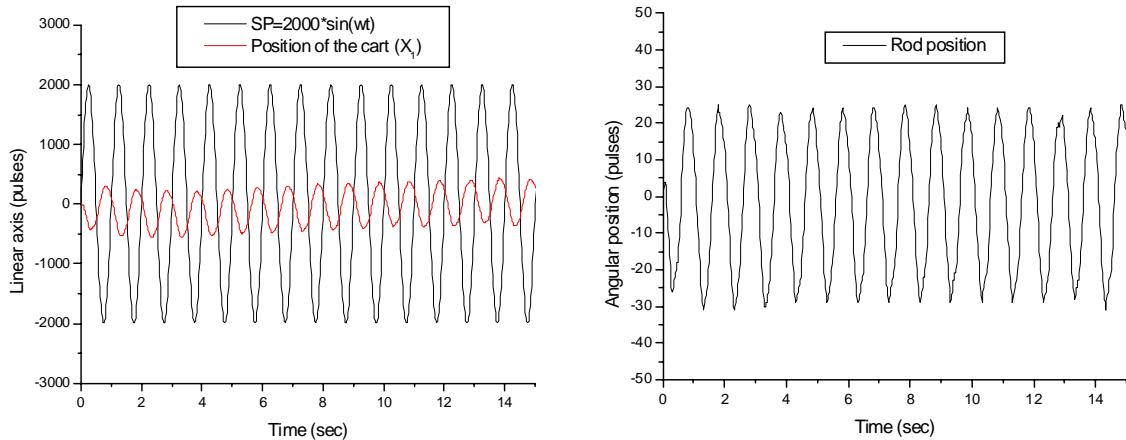


Figure 9: PID Tracking control using a dynamical $SP=2000 \sin(\omega t)$, $F = 1\text{Hz}$

As we can see from Figure 8 and Figure 9, the system under LQR control has less attenuation than the system under PID control using a 1 Hz input signal. Another interesting feature is that the system under PID control tries to eliminate the effect of a sub-harmonic frequency present in the experiment shown in Figure 9.

c) 0.5 Hz

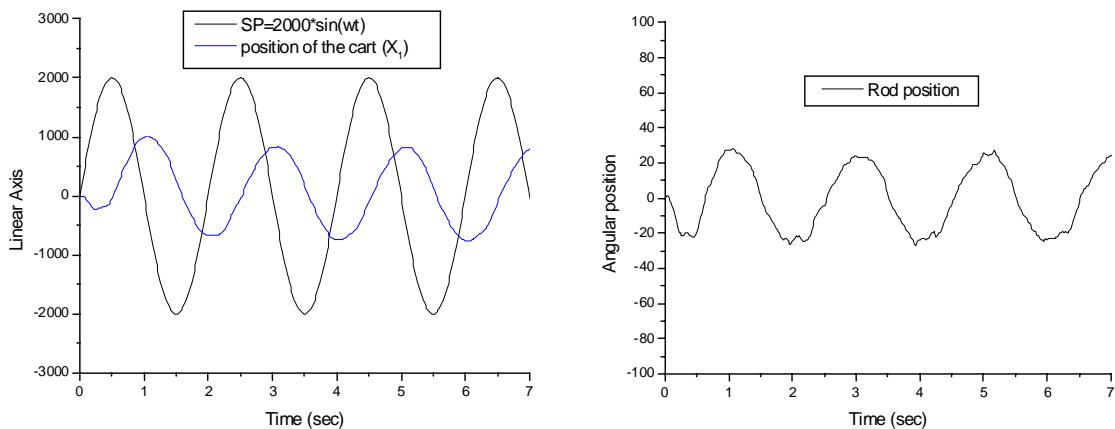


Figure 10: LQR Tracking control using a dynamical $SP=2000 \sin(\omega t)$, $F = 0.5\text{Hz}$

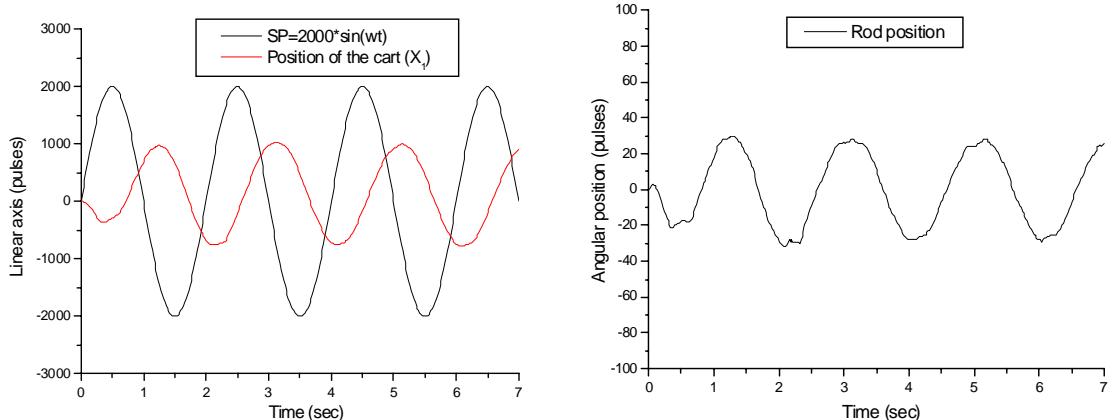


Figure 11: PID Tracking control using a dynamical $SP=2000 \sin(\omega t)$, $F = 0.5\text{Hz}$

With a 0.5 Hz signal (Figure 10 and Figure 11), PID and LQR systems have the same attenuation and phase delay. However, the system under PID action rejects the sub-harmonic effects that LQR can not avoid.

d) 0.25 Hz

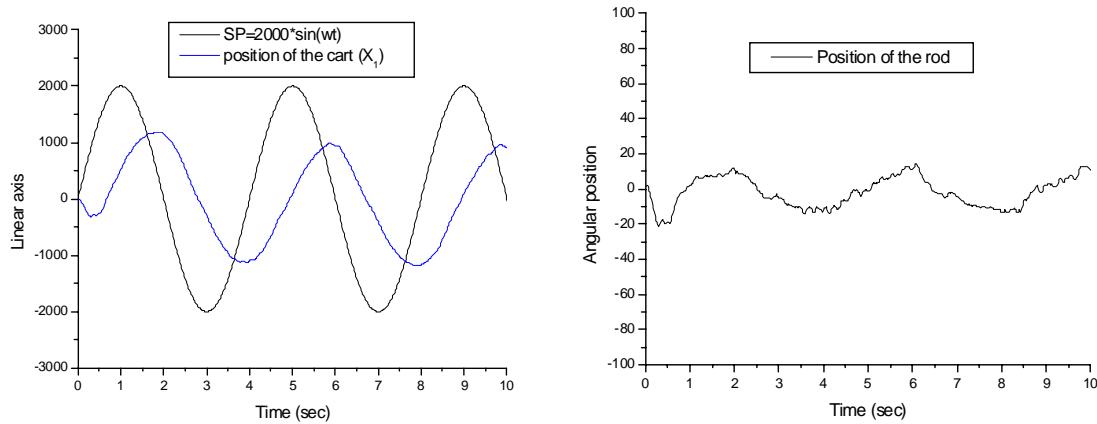


Figure 12: LQR Tracking control using a dynamical SP=2000 sin (wt), F = 0.25Hz

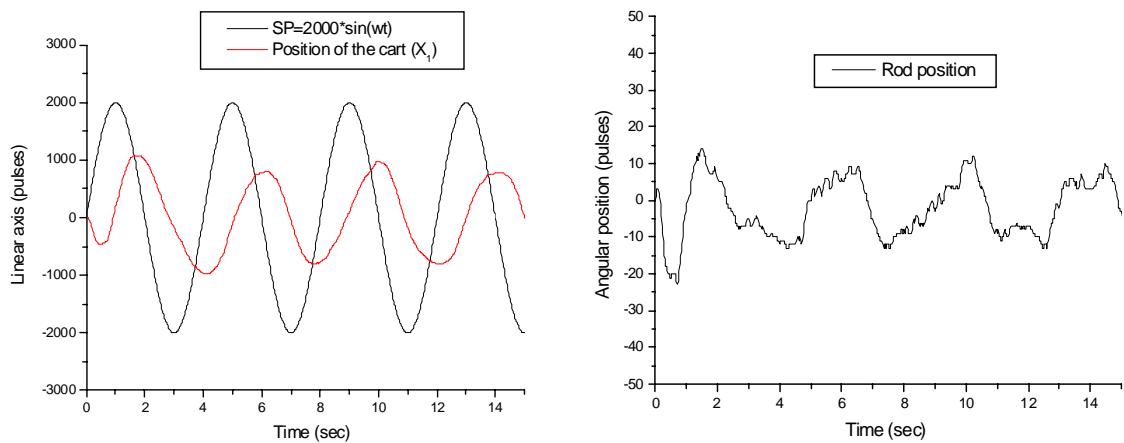


Figure 13: PID Tracking control using a dynamical SP=2000 sin (wt), F = 0.25Hz

The LQR controller used for the experiments shown in Figure 12, Figure 14 and Figure 16 gradually becomes more stable and follows the trajectory smoothly: that is, the errors reduce over time.

By contrast, the system under PID control becomes erratic and the error between the equilibrium point and the rod position does not improve over time (Figure 13, Figure 15, Figure 17 and Figure 19).

e) 0.125 Hz

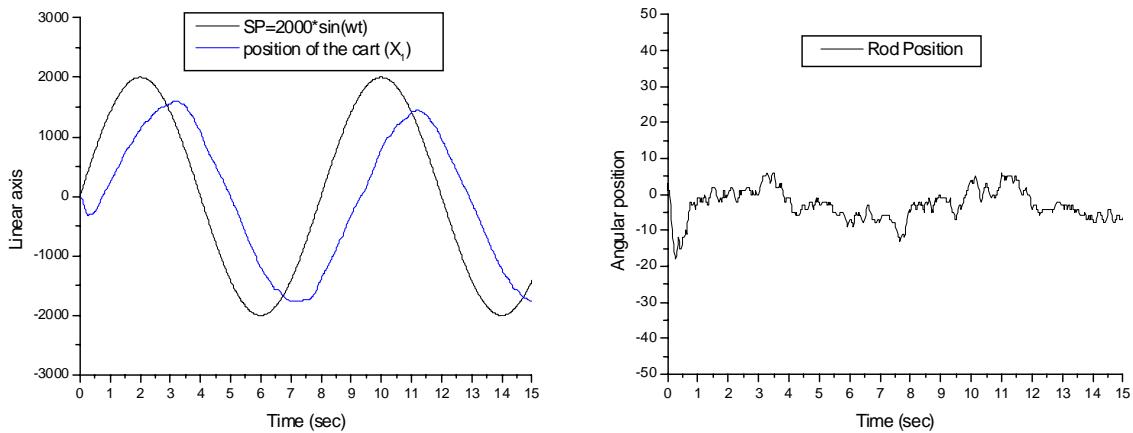


Figure 14: LQR Tracking control using a dynamical $SP=2000 \sin(\omega t)$, $F = 0.125\text{Hz}$

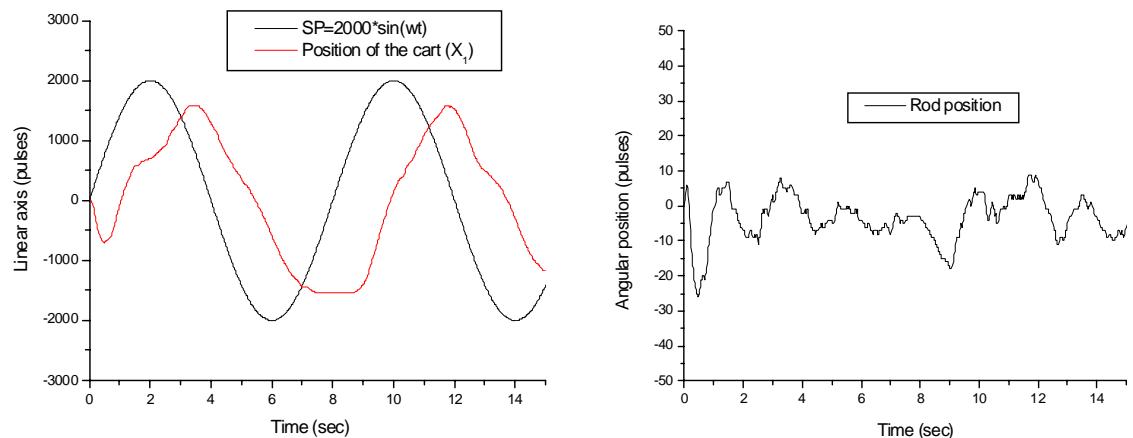


Figure 15: PID Tracking control using a dynamical $SP=2000 \sin(\omega t)$, $F = 0.125\text{Hz}$

f) 0.0625 Hz

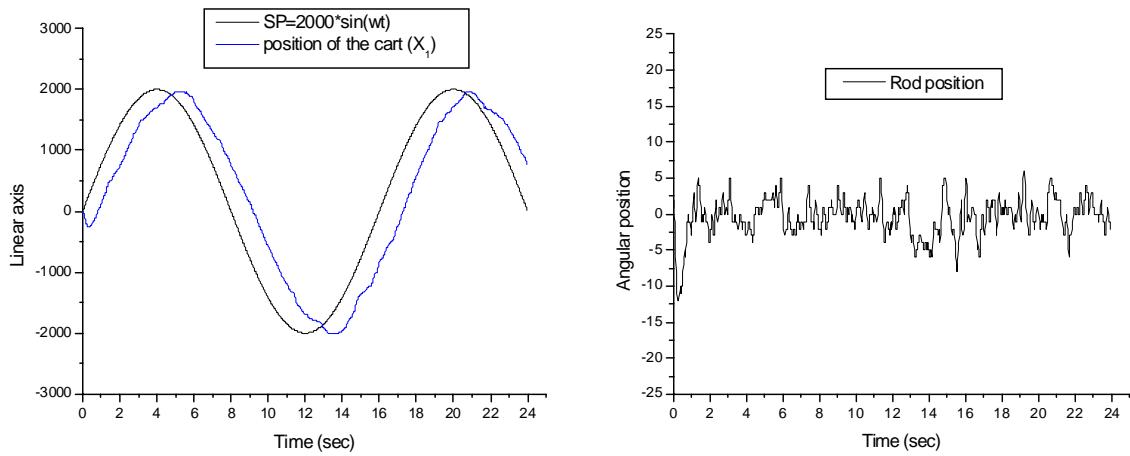


Figure 16: LQR Tracking control using a dynamical $SP=2000 \sin(\omega t)$, $F = 0.0625\text{Hz}$

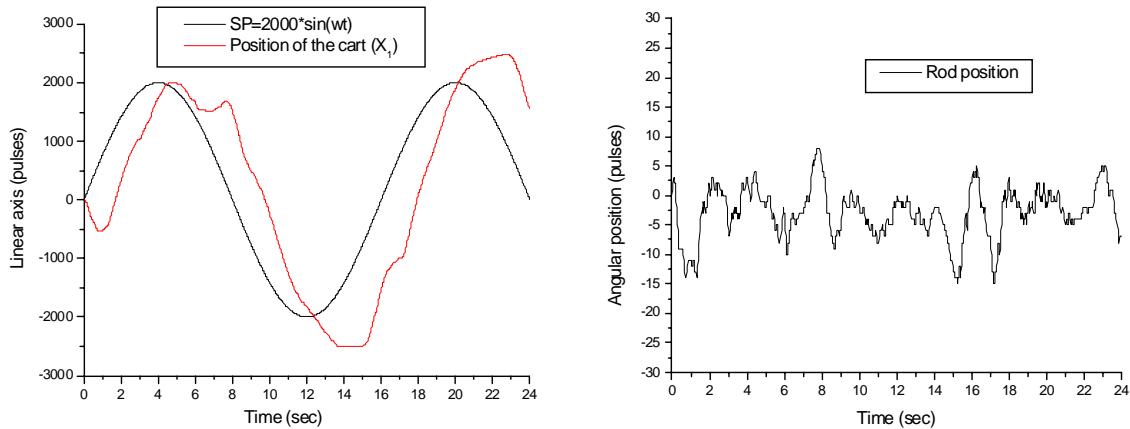


Figure 17: PID Tracking control using a dynamical $SP=2000 \sin(\omega t)$, $F = 0.0625\text{Hz}$

g) 0.03125

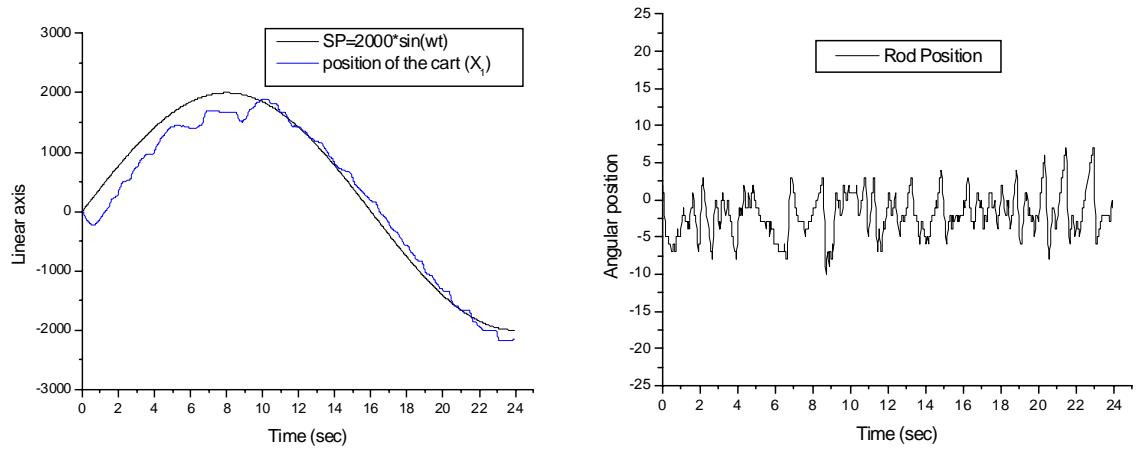


Figure 18: LQR Tracking control using a dynamical $SP=2000 \sin(wt)$, $F = 0.03125\text{Hz}$

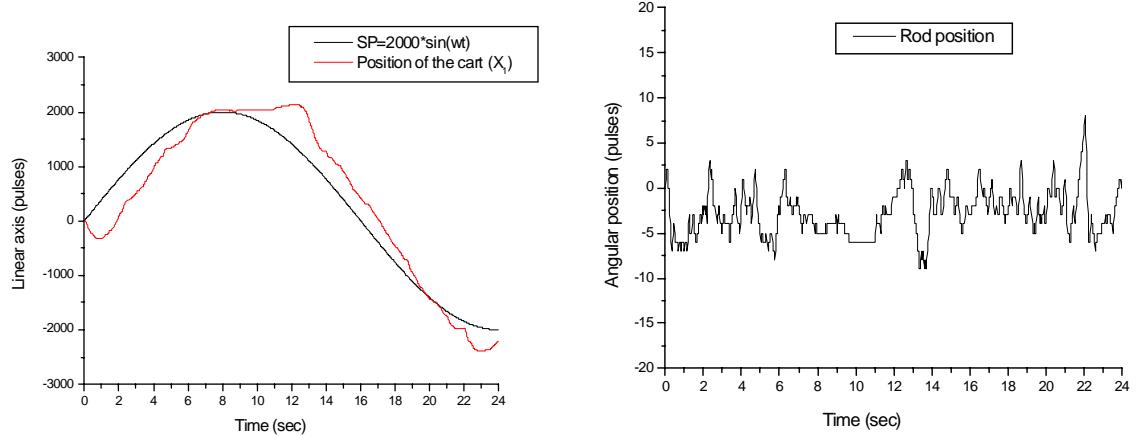


Figure 19: PID Tracking control using a dynamical $SP=2000 \sin(wt)$, $F = 0.03125 \text{ Hz}$

9. Comparing the robustness of the two controllers

To test the robustness of the controllers, we changed the mass and the length of the rod (without altering the software in any way).

Specifically, the changes to the system were:

1. Adding a mass of 165 grams at the top of the rod
2. Cutting down the half of the rod

a) Original systems

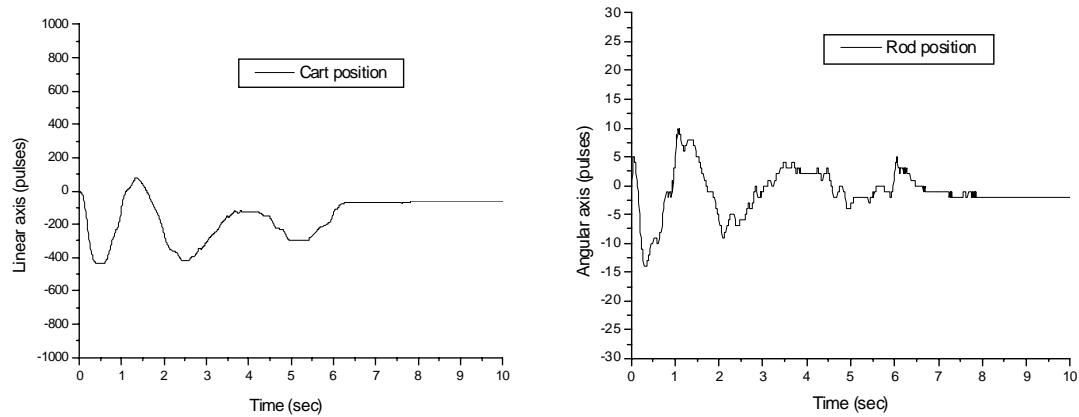


Figure 20: LQR controller, parameters fully known

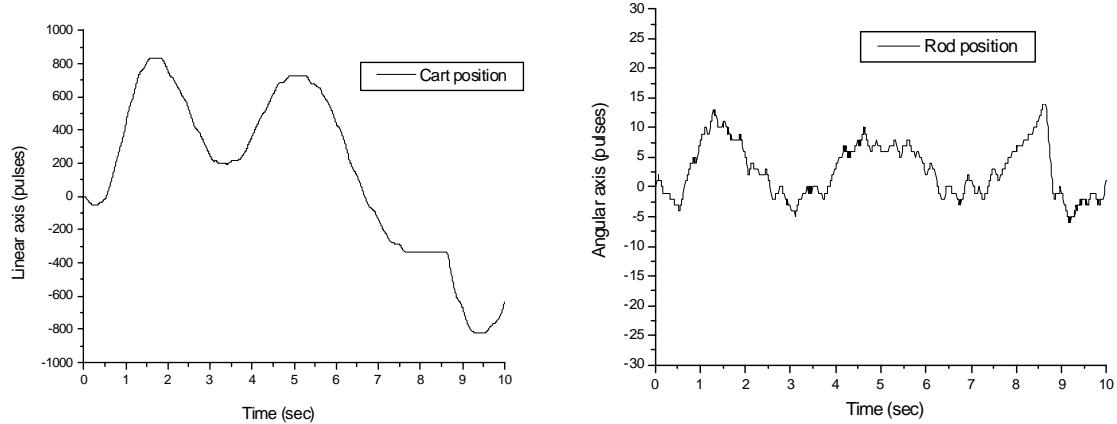


Figure 21: PID controller, parameters fully known

In Figure 20 we can see the LQR system tuned to the original parameters: the system is stable and reaches the equilibrium point in around 8 seconds (for both states).

Figure 21 shows the behaviour of the PID controller. The gains used are the same as those used in the experiments shown in Figure 7. The system is locally stable.

b) Reducing the rod length

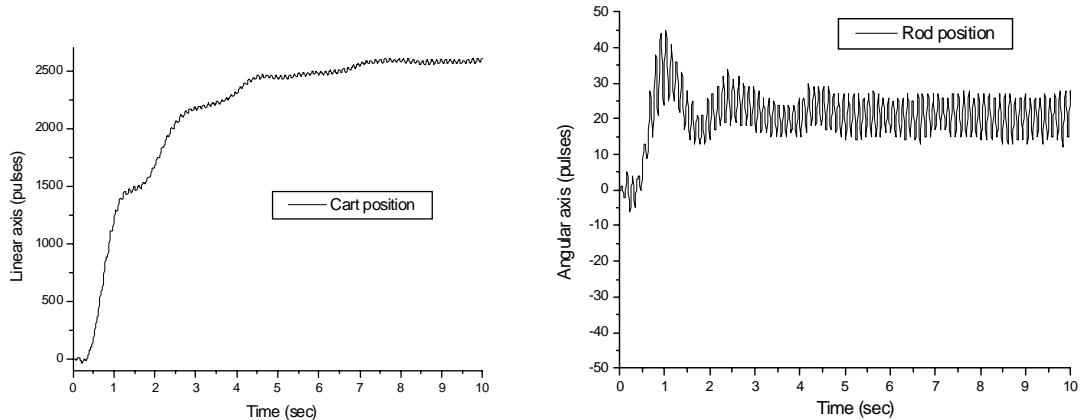


Figure 22: LQR controller, half length of the rod

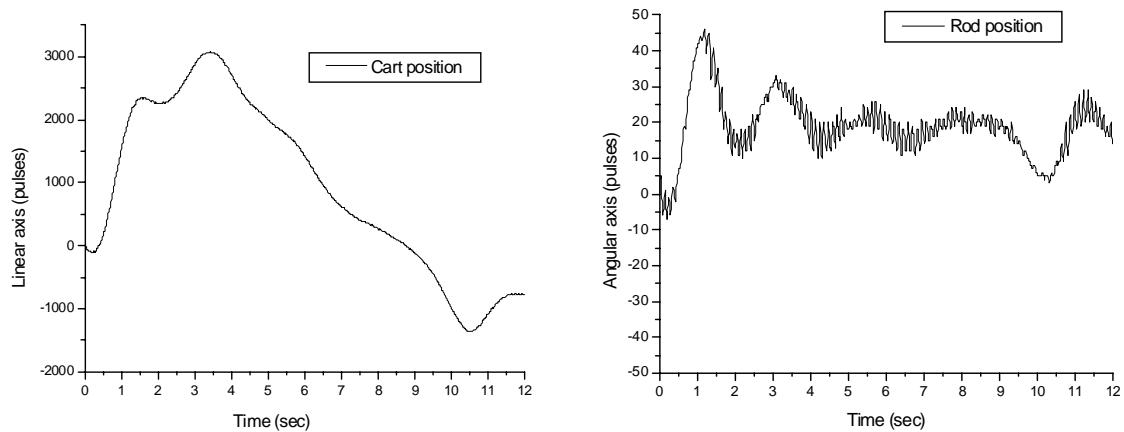


Figure 23: PID controller, half length of the rod

When the pendulum rod length is reduced by half, we can see that – for the LQR controller (Figure 22) - the offset of the cart position increases considerably, and there is “chattering” evident in the rod position control.

For the PID controller, the impact appears to be less (Figure 23).

c) Changing the rod mass

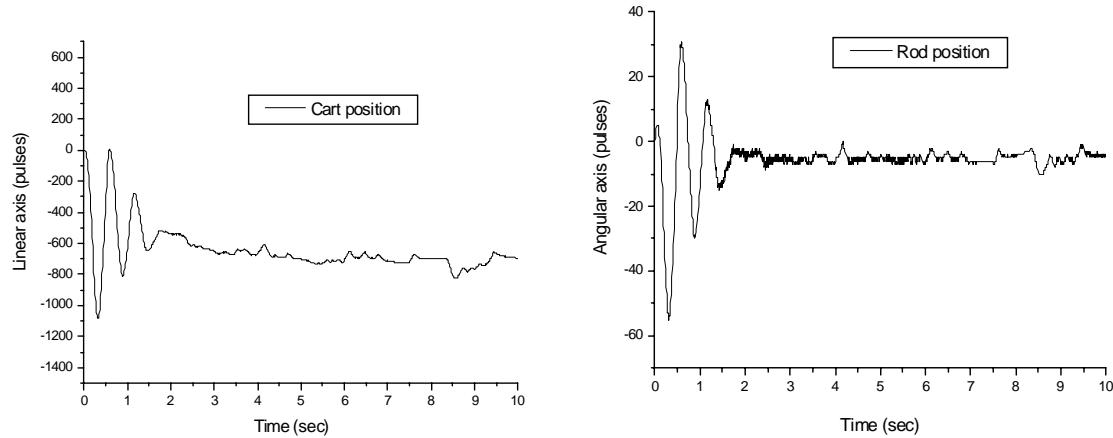


Figure 24: LQR controller, 165 gr. mass in the top of the rod

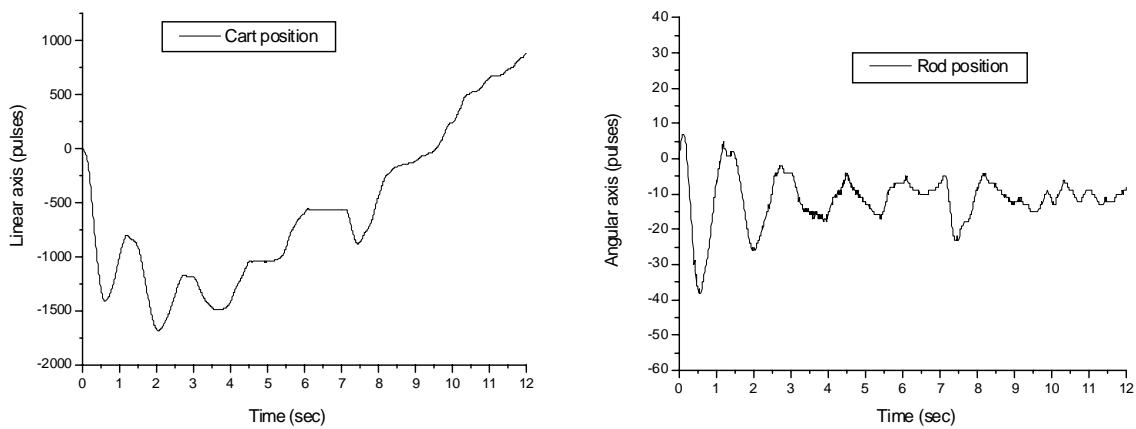


Figure 25: PID controller, 165 gr. mass in the top of the rod

For the LQR controller (Figure 24), changing the mass of the rod has less impact than the change in the length: however, there is still some chattering in the rod position.

By contrast, the performance of the PID controller is more significantly degraded by the mass change (Figure 25).

10. Comparing the resource requirements for the two controllers

Although the way to obtain the gains for LQR and PID are completely different, the embedded microcontroller algorithms are essentially the same. Overall, there are just two important differences:

- i) The PID controller has an integral state which the LQR controller does not have.
- ii) The LQR controller has an internal delay compensator state that the PID algorithm does not have.

We will describe the impact of these differences on the system resource requirements in this section.

a) LQR controller

For the LQR controller, the following processes must be implemented:

- i) We acquire position of the cart and rod using the microcontroller hardware (timers and interrupt handling).
- ii) Using the above position information we obtain an approximation of the speed using pseudo derivates and digital filters.
- iii) The control algorithm requires 5 states, position of cart and rod and their speeds, plus an additional state that stores information about the motor voltage.
- iv) The program multiplies each state by a gain and adds the results. This amounts to 5 multiplications and 4 additions, after which the output is compared with a maximum value.
- v) The program determines the direction of movement and uses this information to control the direction pin at the power stage.
- vi) An offset is added to avoid the “dead zone” of the actuator (DC motor)

The program resources for the LQR algorithm are:

Program Size: data=1308 const=152 code=4328.

b) PID controller

For the PID controller, the following processes must be implemented:

- i) We acquire position of the cart and rod using the microcontroller hardware (timers and interrupt handling).
- ii) Using the above position information we obtain an approximation of the speed using pseudo derivates and digital filters.
- iii) The control algorithm requires 5 variables (position of cart and rod, speed of cart and rod, plus the integral of the cart position - including the stored integral).
- iv) The program multiplies each state by a gain and adds the results. This amounts to 5 multiplications and 4 additions, after which the output is compared with a maximum value.
- v) The program determines the direction of movement and uses this information to control the direction pin at the power stage.
- vi) An offset is added to avoid the “dead zone” of the actuator (DC motor)

The program resources for the PID algorithm are:

Program Size: data=1308 const=148 code=4372

Overall, the PID has a slightly higher memory requirement than the LQR algorithm.

11. Conclusions

In the pilot study described in this paper, we have considered the control of an inverted pendulum system using both LQR- and PID-based embedded systems.

Overall, the two systems have been shown (in the tests discussed here) to have similar control performance. In addition, the “standard” PID control algorithm was seen to have similar resource requirements to a less-common LQR implementation.

Further studies are currently being carried out in order to explore the topics raised in this paper in greater depth.

References

- Chen C-T. (1999) "Linear Systems Theory and Design", Oxford Series in Electrical and Computer Engineering.
- Dorf R. C., and Bishop R. H. (2001) "Modern Control Systems" (9th Edition), Prentice Hall.
- Edwards T., Pont M.J., Scotson P. and Crumpler S. (2004) "A test-bed for evaluating and comparing designs for embedded control systems". In: Koelmans A., Bystrov A. and Pont M.J. (Eds.) Proceedings of the UK Embedded Forum 2004 (Birmingham, UK, October 2004) pp.106-126. Published by University of Newcastle upon Tyne [ISBN: 0-7017-0180-3].
- Franklin G. F., Powell J. D., and Emami-Naeini A. (2002) "Feedback Control Of Dynamic Systems" (4th Edition), Prentice Hall.
- Friedland B. (1998) "On controlling systems with state-variable constraints", Dept. of Electr. & Comput. Eng., New Jersey Inst. of Technol., Newark, NJ, USA. In: American Control Conference, 1998. Proceedings of the 1998 Publication Date: 24-26 June 1998 Volume: 4
- Gixin D., Nanchen H., Wu G., Peiren Z, Zhiyuan Q and Demin S. (2002) "The rotational inverted-pendulum based on DSP controller", Dept. of Autom., Univ. of Sci. & Technol. of China, Hefei, China. In: Intelligent Control and Automation, 2002. Proceedings of the 4th World Congress on Publication Date: 10-14 June 2002 Volume 4.
- Campa, G., Davini, M. and Innocenti, M. (2000) "MvTools: Multivariable Systems Toolbox", In: Proceedings of IEEE International Symposium on Computer-Aided Control System Design, 2000 (CACSD 2000), 25-27 Sept. 2000, pp. 163 – 167.
- Hamming R.W. (1989) "Digital Filters", Prentice Hall.
- Heng-Ming T., and Shenoi S. (1994) "Robust fuzzy controllers", Centre for Intelligent Syst. Tulsa Univ., OK, USA; This paper appears in: Systems, Man, and Cybernetics, 1994. 'Humans, Information and Technology', 1994 IEEE International Conference on Publication Date: 2-5 Oct. 1994, Volume: 1, On page(s): 85 - 90 vol.1
- Lundberg K.H. and Roberge J.K. (2003) "Classical dual-inverted-pendulum control", Dept. of Electr. Eng. & Comput. Sci., Massachusetts Inst. of Technol., Cambridge, MA, USA. In: Decision and Control, 2003. Proceedings. 42nd IEEE Conference on Publication Date: 9-12 Dec. 2003 Volume: 5
- Natale O.R., Sename O. and Canudas-de-Wit C. (2004) "Inverted pendulum stabilization through the Ethernet network, performance analysis", Dept. of Eng., Univ. del Sannio, Benevento, Italy.
- Ogata K. (1995) "Discrete-Time Control Systems", Prentice Hall.
- Ogata K. (2002) "Modern control engineering", Prentice Hall.
- Reyes F. and Kelly R. (1997) "Experimental Evaluation of Identification Schemes on a Direct Drive Robot", División de Física Aplicada, CICESE, Apdo. Postal 2615, Adm. 1, Carretera Tijuana-Ensenada Km. 107, Ensenada, B.C. 22800, Mexico, Robotica, Volume 15, Issue 05. September 1997. pp.563-571
- Seikiguchi M., Sugasaka T. and Nagata S. (1991) "Control of a multivariable system by a neural network [inverted pendulum]". In: Robotics and Automation, 1991. Proceedings,

1991 IEEE International Conference on Publication Date: 9-11 April 1991, pp.2644 - 2649 vol. 3

Spong M.W. and Vidyasagar M. (1989) "Robot Dynamic and Control", John Wiley and Sons.

Varsek A., Urbancic T. and Filipic B. (1993) "Genetic algorithms in controller design and tuning". This paper appears in: IEEE Transactions on Systems, Man and Cybernetics, Volume: 23, Issue: 5, Sept.-Oct. 1993 pp1330 – 1339.

Yasuhiko D. (1990) "Servo Motor and Motion control using digital signal processors", Prentice Hall and DSP series Texas instruments 1990.

An initial comparison of synchronous and asynchronous network architectures for use in embedded control systems with duplicate processor nodes

Tim Edwards¹, Michael J. Pont¹, Michael Short¹, Pete Scotson² and Steve Crumpler²

¹*Embedded systems laboratory,
University of Leicester, University Road, Leicester LE1 7RH, UK.*

²*TRW Conekt,
Stratford Road, SOLIHULL, B90 4GW, UK*

Abstract

Embedded processors are becoming a common component in safety related and safety critical control systems. To provide the required levels of reliability, it is generally accepted that some form of redundancy is required in such designs. In this paper, we are particularly concerned with the use of duplicate processor nodes in distributed embedded control systems. In the scenario considered in this paper, we have a single sensor, a single actuator and two (duplicate) processor nodes. We consider and compare two architectures. In the first, the processor nodes operate independently and asynchronously. In the second architecture, the two nodes operate synchronously. We demonstrate that - under some circumstances - a simple asynchronous architecture can achieve performance levels equal to (or exceeding) those of the synchronous architecture.

Acknowledgements

The work described in this paper was supported by the UK Government (EPSRC), TRW Conekt and the Leverhulme Trust.

1. Introduction

The use of embedded systems in real-time control applications is now widespread. Such implementations are popular because they are affordable and may be easily modified and maintained (Heiner and Thruner, 1998; Storey, 1996). In a number of existing applications (e.g. fly-by-wire) the safety-critical nature of the systems requires very high levels of safety and reliability. Specified levels of performance must be guaranteed under all conditions, even if a fault occurs in part of the system (whether that part is – for example - a sensor, actuator, processor or communications bus). This fault-tolerant behaviour is often reliant on redundancy in every aspect of the system (Isermann *et al.*, 2002; Ladkin, 1995; Stanton, 1996). For example, in the aerospace industry it is common to find triple modular redundancy or dual duplex (self-checking pair) arrangements (Hammett, 2002; Pratt, 2000; Rushby, 2000).

As levels of fault tolerance increase, both system complexity and cost are also likely to rise significantly (Isermann *et al.*, 2002). In some emerging applications, such as the highly competitive automotive industry, there are restrictions on the level of redundancy that can be implemented due to cost and - in some cases - physical constraints (Hedenetz & Belschner, 1998).

Our concern in this paper is with cost-effective ways of achieving redundancy in the actuation stage of embedded control systems. In such systems, “Fail silence” is a key requirement since a faulty actuator can hinder the operation of a duplicate. As a consequence, redundancy in actuators can often involve complex electro-mechanical designs, and this in turn can cause significant control challenges (for example see Härkegård & Glad 2005; Napolitano *et al.* 2000). As an alternative, some motor-driven actuators include redundant motor windings or use dual non-self locking motors or gears (Dilger *et al.*, 1997).

In the study discussed in this paper, we consider a slightly different scenario in which we have duplicate (embedded) processors “fighting” for control of a single actuator. Our focus is on two specific system architectures. The first architecture involves running two duplicate processors independently (and, therefore, asynchronously). The second architecture involves running the two processors synchronously and deliberately out of phase. An initial evaluation and comparison of these architectures is carried out using a simple version of a throttle-by-wire traction controller operating as part of a hardware-in-the-loop (HIL) testbed.

The remainder of the paper is organised as follows. In Section 2 the HIL testbed is described and experimental data are presented to illustrate its. Section 3 describes the two architectures and the techniques used to assess and compare them. In Section 4 we discuss the results obtained and present our conclusions

2. The testbed

As noted in the introduction, the experiments described in this paper are based around a hardware-in-the-loop (HIL) testbed. We describe the testbed in this section.

2.1 Why HIL?

There are numerous benefits to an HIL approach when compared to testing with a physical system. For example, a simulated environment allows for repeatable sets of test conditions, which can improve the accuracy of comparisons between different controller designs. In addition, real-world testing is not a viable option at this stage in the development process of many systems for safety reasons.

2.2 Throttle-by-wire

A throttle-by-wire control application forms the focus of the present study. Throttle-by-wire refers to automotive throttle control using a combination of sensors, one or more embedded controllers and an electromechanical actuator motor (e.g. see Rossi *et al.*, 2000)

The simulated environment used for this testbed is based on a motorway model developed in our laboratory (Short and Pont, 2005). The simulation is implemented on a DOS-based desktop PC using a time-triggered scheduler (Pont *et al.*, 2004). The simulation was modified to use only the throttle-control hardware.

The throttle controller nodes themselves are kept simple in the present study. Using just three variables from the simulator (vehicle speed, wheel speed and the drivers throttle setting) a basic traction control algorithm is implemented: this limits the throttle commands to prevent wheel slippage from over-acceleration.

The original version of the motorway simulator was designed to interface with a more complex distributed automotive system for throttle and brake control (Short *et al.*, 2004). In such a system the throttle control node would expect sensor information over some form of communications bus rather than interfacing directly with the sensors. To replicate this behaviour an extra C167

processor is used as an interface between the simulation and the throttle controller. The inter-processor communications are carried out using CAN (Bosch, 1991), an established and widely used protocol in the automotive sector (Kopetz, 1998; Thomasse, 1998).

The PC broadcasts new data to the interface node every 10ms over and RS-232, this data is immediately transmitted to the controller via. CAN. Figure 1 illustrates the full control loop for this test bed. The actuator gateway regulates access to the throttle actuator. There are a number of ways this node can be configured; in this paper we consider only the case where all new commands sent to the actuator gateway are immediately relayed to the actuator itself.

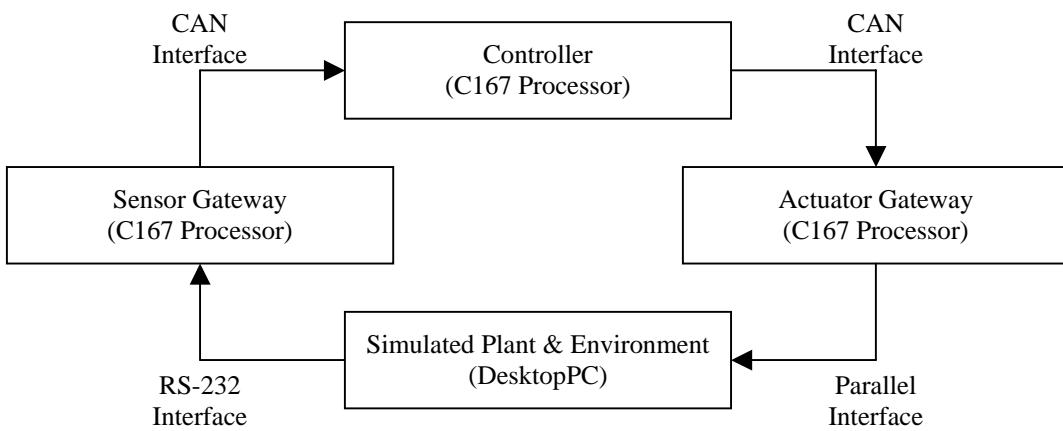


Figure 1: Illustration of the actual control loop implementation

2.3 System performance

With a testbed system in place, a benchmark of performance was recorded using a single controller node. A range of sample periods were implemented, from 1 to 1000 ms. In each case, the test involved accelerating the car up to 20, 40 and then 60 MPH. Each of the target speeds was set for 20 seconds.

Figure 2 shows the vehicles speed over a single 60 test case. The set speeds are indicated with dotted lines. This example with a 10ms sample period was used as our benchmark for a tuned system. Figure 3 shows how the performance degrades as the sample period is increased (in this case up to 50ms)

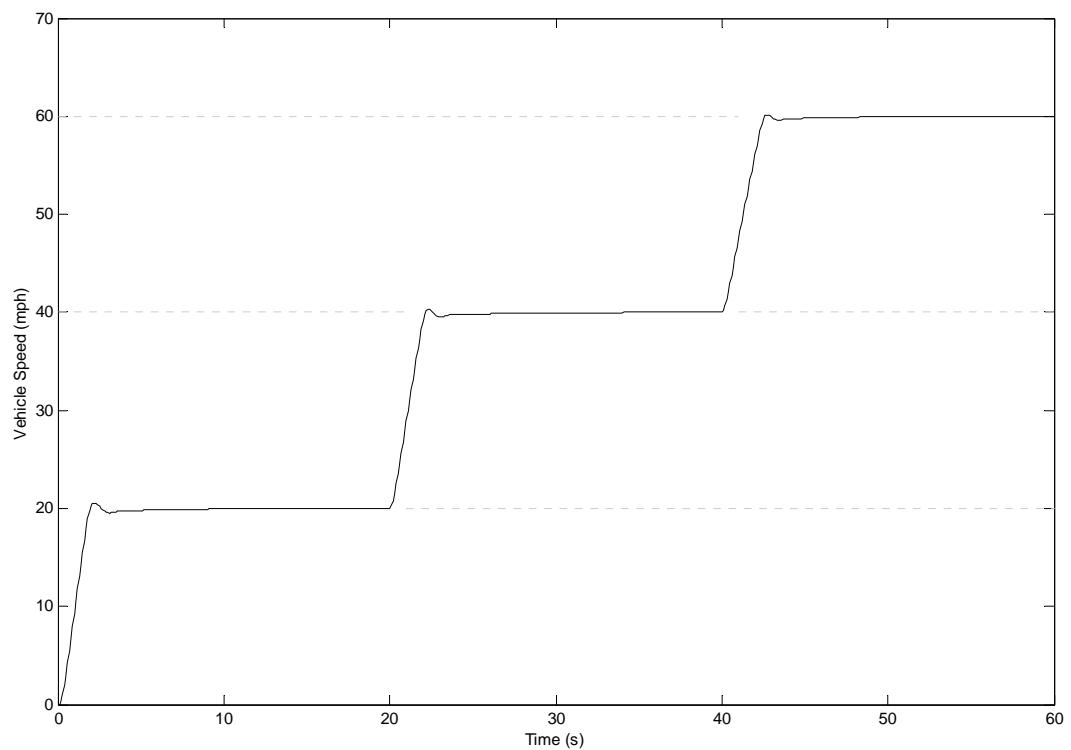


Figure 2: A benchmark for speed control (sampling period = 10ms)

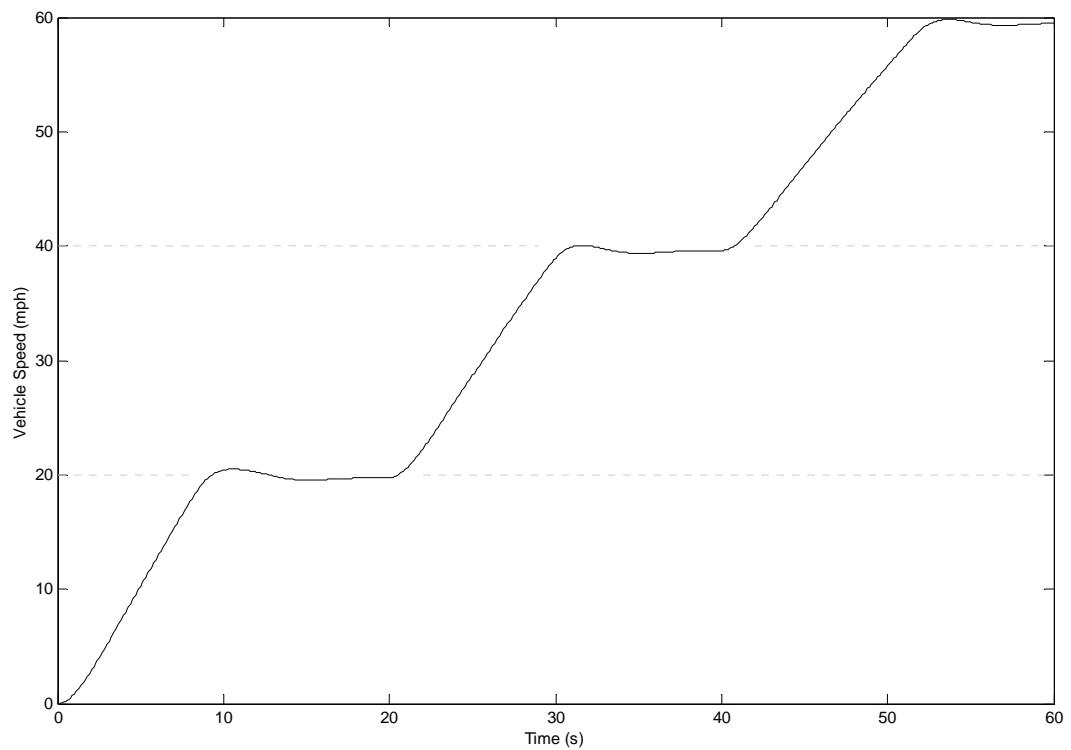


Figure 3: Degraded speed control with a 50ms sampling period

This single controller implementation was tested with a wide range of sample periods (from 1 – 1000ms) to get a clear idea of the systems tolerance to delays. The results were assessed using a number of standard performance indices. The results for IAE (Integral of absolute error) are shown in Figure 4, and for ISE (Integral of squared error) in Figure 5. Full details of these measures are widely available in control text books (e.g. Dorf and Bishop, 2001).

Both figures show that the performance degradation becomes more rapid after a threshold period (approximately 120ms) has been crossed. Very soon after this point speed control is lost entirely which is why the curves begin to level out.

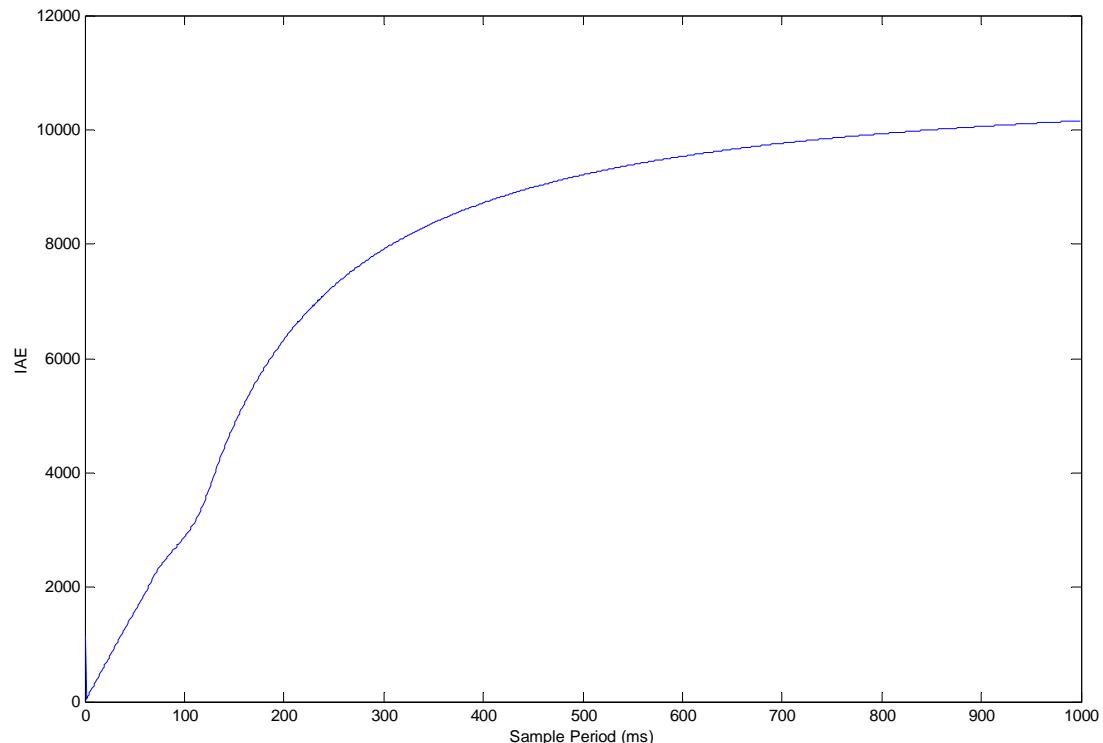


Figure 4: IAE for a single controller implementation with sample periods from 1 to 1000ms

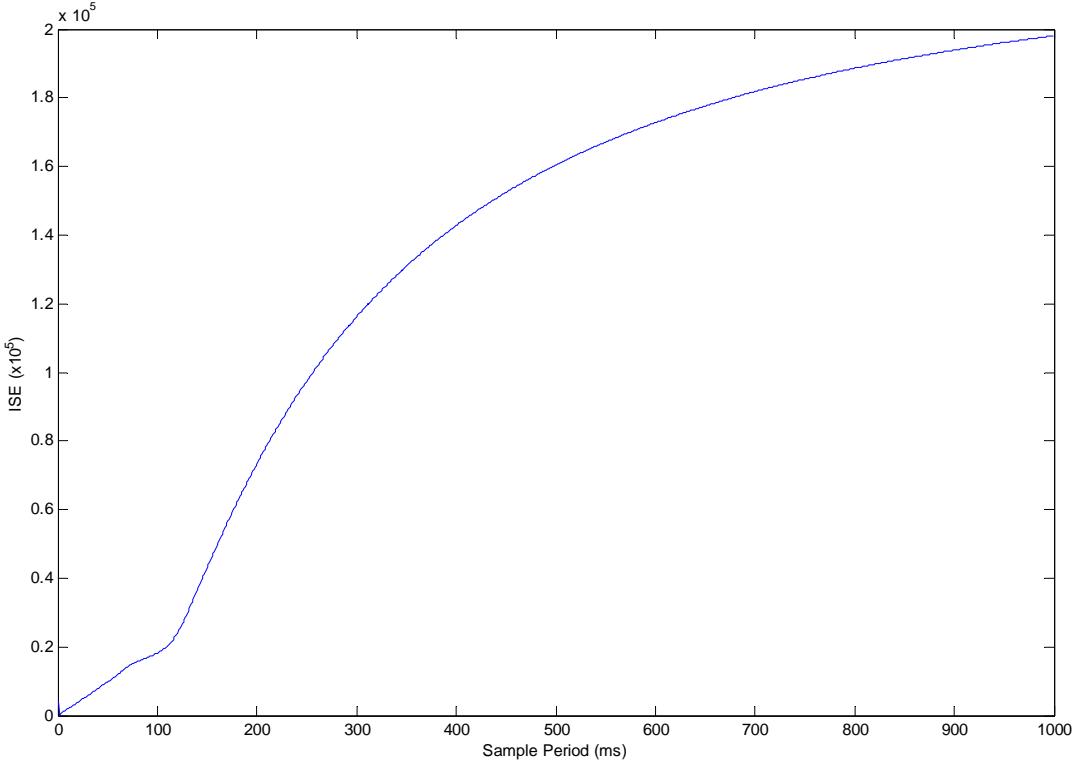


Figure 5: ISE for a single controller implementation with sample periods from 1 to 1000ms.

2.4 Discussion

In real-world systems, especially those of a safety critical nature, it is always desirable to operate well within a safe region of sampling periods. However, it is also desirable to minimize CPU overheads by avoiding over sampling. Moreover, in distributed control architectures, bandwidth constraints can also limit practical sampling rates (Lian, 2002).

In this case our system cannot tolerate much more than 120ms sampling period to maintain even partially degraded control of the vehicle speed. Therefore we will compare the synchronous and asynchronous architectures around this region.

3. Processor redundancy

In this section we consider the addition of a redundant processor node to the system discussed in Section 2.

3.1 Two architectures

In the previous section an HIL testbed based around a single-processor controller was discussed. In this section, we consider the same system but with an additional, duplicate, controller (Figure 6): as Hammett (2002) has discussed, this form of processor redundancy is a cost-effective first step that can be taken to achieve fault-tolerance.

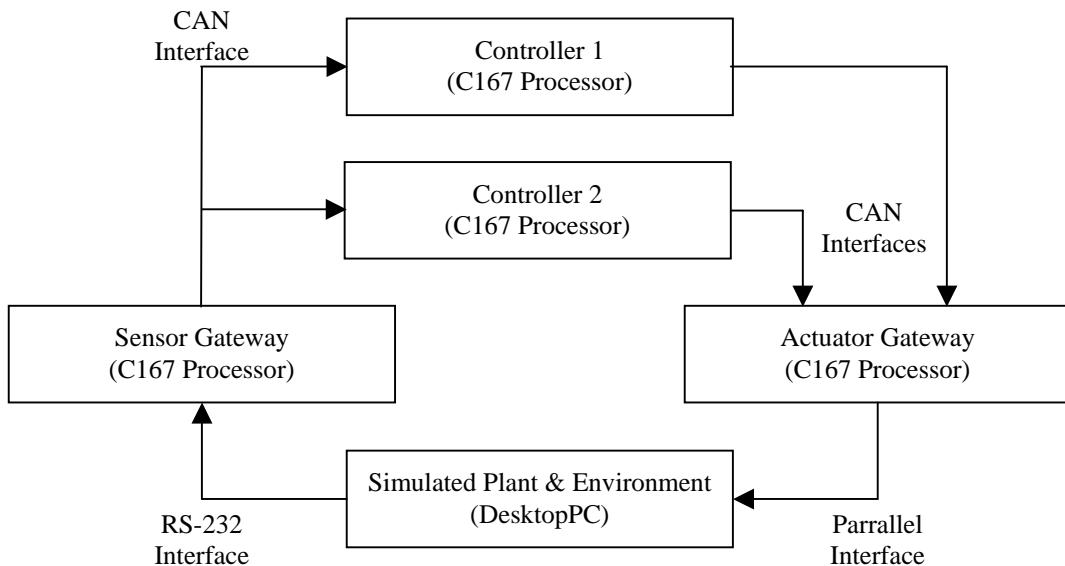


Figure 6: Illustration of the control loop with two controller nodes

In this paper we will address two possible architectures for operating the two controllers. The first involves both processors operating independently and asynchronously. The second architecture involves synchronising the activity on the two processors. In both cases the actuator gateway will relay the latest commands from each node when they arrive.

3.2 Asynchronous operation

A simple approach to incorporating processor redundancy into a real-time embedded control system is to add a second controller node exactly the same as the first. In the case of our test system this means an additional C167 board with its own timer-driven scheduler.

If one were to take this approach, without any alterations to the software, the two boards would begin operation at power up. This starting time would be approximately the same, assuming the boards shared a common power supply (as would usually be the case in automotive applications). This means that initially the two controllers would be sample data and output the new control commands at roughly the same time. However, the clocks would drift apart with time due to differences in the crystal and differences in their environment (e.g. temperature).

Figure 7 illustrates the affect of this on the sample instants. The blocks marked ‘1’ and ‘2’ indicate the execution of the control task by each of the two controller nodes. The clock differences mean the sample periods are not identical for both nodes and the drift effect can be seen.

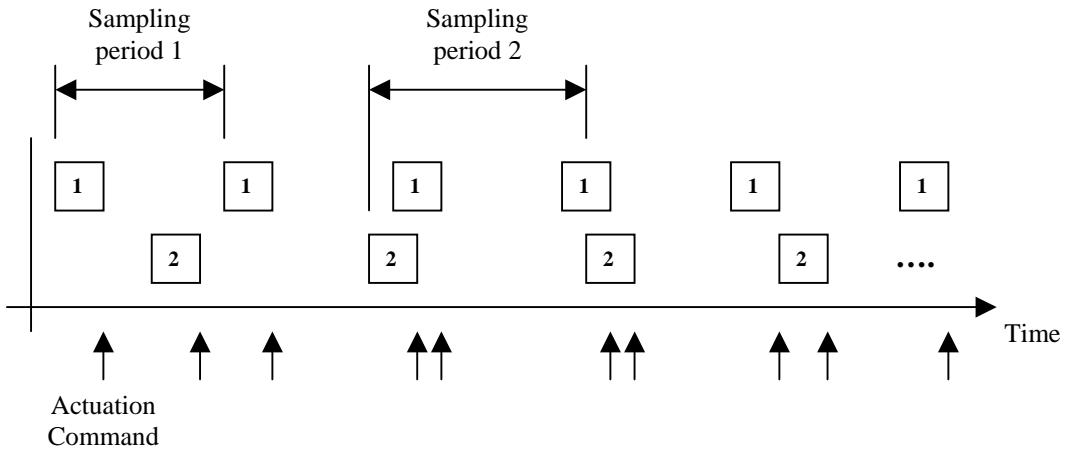


Figure 7: Illustration of the two controller nodes behaviour when operating asynchronously

For the purposes of the tests described here we chose to exaggerate the difference in the clocks of our two controller nodes. In each test the sample period of controller two is offset by 10ms from that of controller one.

3.3 Synchronous operation

Synchronising the two nodes requires some additional design work. The method adopted here is to use the data messages from the sensor interface nodes (already transmitting regular messages every 10 ms) to drive the schedulers of the two controllers. This effectively works like a shared clock scheduler (Pont, 2001) but without any acknowledgement messages being sent in this implementation.

With both controllers running their schedulers in time, they can be set to operate perfectly out-of-phase with each other (Figure 8). This means that, when both are operating properly, the effective sample frequency is doubled.

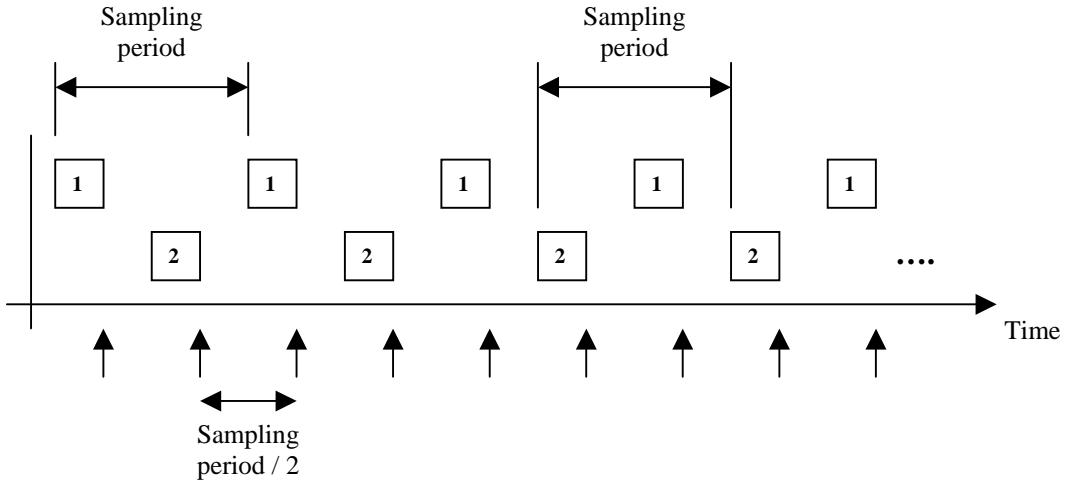


Figure 8: Illustration of the two controller nodes behaviour when operating synchronously

3.4 Test overview

Two different system designs featuring duplicate throttle controllers have been described. To evaluate and compare the performance of these techniques, both systems were tested with sample ranging from 80 to 220 ms. Each test was repeated 10 times to improve the accuracy of results.

Please note that, in these tests, the asynchronous system was set up with deliberate offsets in the task timing and the synchronous controllers were set to operate precisely out of phase with each other.

The control performance is based on the vehicle speed, and in each case two popular performance indices were calculated. The integral of absolute error (IAE) and the integral of the error squared (ISE): see, for example, Dorf (2001).

Figure 9 shows the results using an IAE performance measure. The asynchronous system is represented with a solid line, and the synchronous system with a dashed line. In the same way Figure 10 shows the results using the ISE measure.

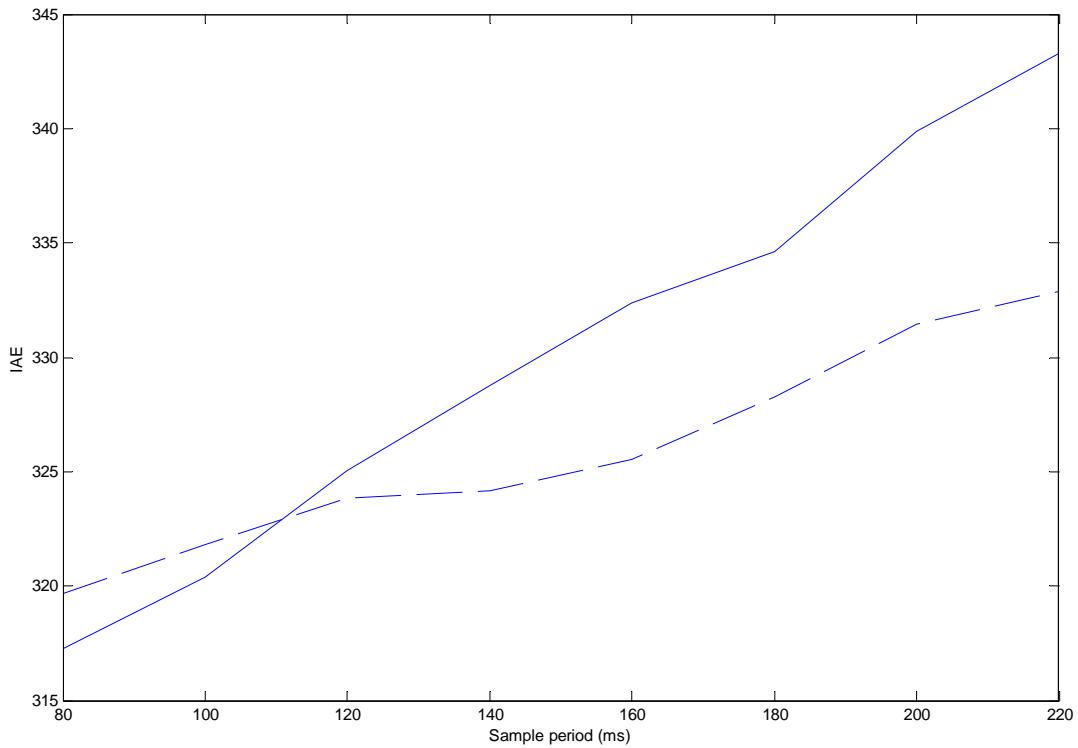


Figure 9: IAE measures (for sampling periods from 80 to 200ms) of speed control using asynchronous (solid line) and synchronous (dashed line) controllers

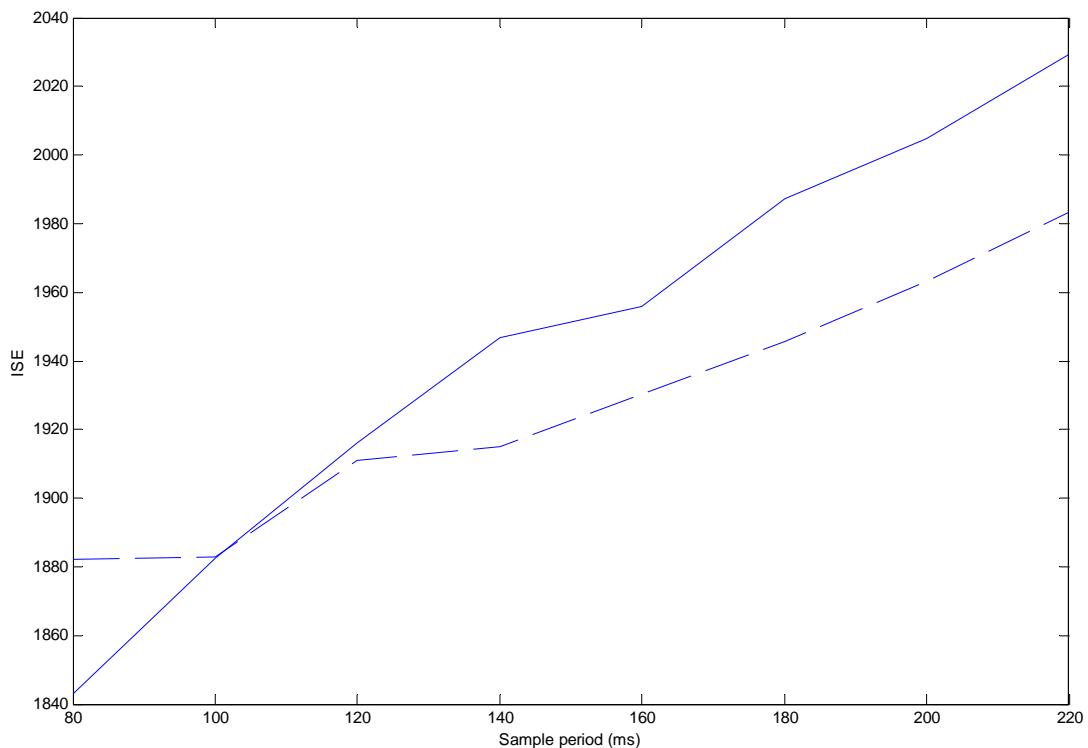


Figure 10: ISE measures (for sampling periods from 80 to 200ms) of speed control using asynchronous (solid line) and synchronous (dashed line) controllers

Figures 9 and 10 both show that for small sample periods the synchronous system can provide performance that is as good or even better than the synchronous architecture. However as the sample rate is increased the performance of the synchronous system is clearly better.

4. Discussion and conclusions

In this paper we introduced an HIL testbed which represents a simple throttle-by-wire system. The system performance was assessed using a single processor controller implemented with a wide range of sampling periods. Using a simple test scenario we showed that the speed control was more robust than throttle control as the sample interval was increased.

A duplicate throttle controller was then introduced into the system and two different system architectures were considered (asynchronous and synchronous). The logged data from 60 comparative tests showed that within a “safe” region of sampling periods, the asynchronous approach could produce the same or better performance than the synchronised system. As the sample period was increased towards the systems upper limit, the performance of the asynchronous system was, in most cases, not as good as that of the synchronised one.

In effect, the asynchronous approach results in a variable sample rate. If there is little margin for error in sample time then the synchronised system is clearly better. The slight increase in complexity required to synchronise the nodes may also bring additional benefits. For example, in a synchronous system the behaviour is deterministic which may mean that fault identification and actuator command “sanity checking” can be implemented more easily and more effectively.

Future tests will consider more complex test cases and system behaviour in the presence of injected faults.

References

- Bosch (1991), “CAN Specification Version 2”, Published by Robert Bosch GmbH
- Dilger et al., (1997), “Towards an architecture for safety related fault tolerant systems in vehicles”, ERSEL – European Conference on Safety and Reliability.
- Dorf R. C., and Bishop R. H., (2001), “Modern Control Systems – 9th Edition”, Prentice Hall.
- Franklin G. F., Powell J. D., and Workman M. L., (1995), “Digital control of dynamic systems”, Menlo Park, California, Harlow, Addison Wesley.

Franklin G. F., Powell J. D., and Emami-Naeini A., (2002), "Feedback Control Of Dynamic Systems – 4th Edition", Prentice Hall.

Hammett R., (2002), "Design by Extrapolation: An Evaluation of Fault Tolerant Avionics", IEEE AES Systems Magazine, April 2002.

Harkegard O., and Glad S., (2005), "Resolving actuator redundancy – optimal control vs control allocation", Automatica Vol 41 Issue 1, Jan 2005, pp. 137-144

Hedenetz B., and Belschner R., (1998) "Brake-by-wire without Mechanical Backup by Using a TTP-Communication Network", SAE Paper Number: 981109

Heiner G. and Thurner T., (1998), "Time-Triggered Architecture for Safety-Related Distributed Real-Time Systems in Transportation Systems", 28th Annual Symposium on Fault Tolerant Computing, IEEE Computer Science Press, Munich, Germany.

Isermann R., Schwarz R., and Stölz S., (2002), "Fault-tolerant drive-by-wire systems", IEEE Control Systems Magazine, October 2002.

Kopetz H., (1998), "A Comparison of CAN and TTP", Vienna University of Technology.

Ladkin P.B., (1995), "Analysis of a Technical Description of the Airbus A320 Braking System", CRIN-CNRS & INRIA

Lewis, D., (2001), "Fundamentals of Embedded Software", Prentice Hall

Lian F., Moyne J., and Tilbury D. (2002), "Network Design Consideration for Distributed Control Systems", IEEE Transactions on Control Systems Technology, Vol. 10, No. 2, pp 297-306.

Napolitano M. R., An Y., and Seanor B. A., (2000), "A fault tolerant flight control system for sensor and actuator failures using neural networks", Aircraft Design 2, pp. 103 – 128.

Nilsson J., (1998), "Real-time control systems with delays", Ph.D Thesis, ISRN LUTFDR2/TFRT-1049-SE, Dept. of Automatic Control, Lund Institute of Technology, Sweden.

Ogata K., (1995), "Discrete-Time Control Systems", Prentice Hall, London

Park H. S., Kim Y. H., Dong-Sung K., and Kwon W. H., (2002), "A Scheduling Method for Network-Based Control Systems", IEEE Transactions on Control Systems Technology, Vol. 10, No. 3, pp 318-329.

Pratt R.W., (2000), "Flight Control Systems – Practical Issues in Design and Implementation", Published by IEE.

Pont, M.J., Norman, A.J., Mwelwa, C. and Edwards, T. (2004) "Prototyping time-triggered embedded systems using PC hardware". In: Henney, K. and Schutz, D. (Eds) Proceedings of

the eighth European conference on Pattern Languages of Programs (EuroPLoP 2003), Germany, June 2003: pp.691-716. Published by Universitätsverlag Konstanz. ISBN 3-87940-788-6.

Pont M. J., (2001), "Patterns for Time-Triggered Embedded Systems", Addison-Wesley.

Rossi C., Tilli A. and Tonielli A., (2000), "Robust Control of a Throttle Body for Drive by Wire Operation of Automotive Engines". Published in IEEE Transactions on Control Systems Technology, Vol. 8, No. 6, November 2000.

Rushby J., (2003), "A Comparison of Bus Architectures for Safety-Critical Embedded Systems", NASA Contractor Report CR-2003-212161

Siemens AG, (1996), "C167 Derivatives – User's manual Version 2.0", Published by Siemens AG.

Short M. and Pont M.J. (2005) "Hardware in the Loop Simulation of Embedded Automotive Control Systems", in Proceedings of the 8th IEEE International Conference on Intelligent Transportation Systems (IEEE ITSC 2005) held in Vienna, Austria, 13-16 September 2005, pp. 226-231.

Short M., Pont M.J. and Huang Q. (2004), "Safety and Reliability of Distributed Embedded Systems: Development of a Hardware-in-the-Loop Test Facility for Automotive ACC Implementations". Embedded Systems Laboratory Technical Report ESL04/03.

Stanton N. A., and Marsden P., (1996), "From Fly-by-wire to Drive-by-wire: Safety Implications of Automation in Vehicles, Safety Science Vol. 24 No. 1, Elsevier Science Ltd.

Storey N., (1996), "Safety-Critical Systems", Addison Wesley.

Thomasse J. P., (1998), "A Review of the Fieldbuses", Annual Reviews in Control, Volume 22, Pages 35-45