Communication Estimation for Hardware/Software Codesign

Knudsen, Peter Voigt; Madsen, Jan

Published in:
Hardware/Software Codesign, 1998. (CODES/CASHE '98) Proceedings of the Sixth International Workshop on

Link to article, DOI:
10.1109/HSC.1998.666238

Publication date:
1998

Document Version
Publisher's PDF, also known as Version of record

Communication Estimation for Hardware/Software Codesign

Peter Voigt Knudsen and Jan Madsen
Department of Information Technology, Technical University of Denmark
pvk@it.dtu.dk,jan@it.dtu.dk

Abstract

This paper presents a general high level estimation model of communication throughput for the implementation of a given communication protocol. The model, which is part of a larger model that includes component price, software driver object code size and hardware driver area, is intended to be general enough to be able to capture the characteristics of a wide range of communication protocols and yet to be sufficiently detailed as to allow the designer or design tool to efficiently explore tradeoffs between throughput, bus widths, burst/non-burst transfers and data packing strategies. Thus it provides a basis for decision making with respect to communication protocols/components and communication driver design in the initial design space exploration phase of a co-synthesis process where a large number of possibilities must be examined and where fast estimators are therefore necessary. The full model allows for additional (money)cost, software code size and hardware area tradeoffs to be examined.

1. Introduction

This paper presents the underlying estimation model for a communication estimation tool which extends the current communication estimation capabilities of the LYCOS [2] co-synthesis system. The model is the basis of a high level communication library that for each supported processing unit and for each supported protocol captures performance/area/price and other characteristics of the necessary drivers and of the communication channel. Our aim is to utilize this library in a communication estimation tool that will work together with the other estimation/partitioning tools in LYCOS as *part* of the design space exploration/co-synthesis cycle. Most current approaches to co-synthesis consider communication synthesis to be a final step in the co-synthesis trajectory [1][3][4]. For instance, [1] presents communication synthesis as an allocation problem to be solved *after* system-level partitioning whereas we *integrate* communication synthesis with design space exploration and system level partitioning. For example, we wish to be able to trade off a fast and expensive communication protocol for a slow but cheaper protocol and a faster co-processor, if that is feasible. This should not be done after system level partitioning as the level of communication overhead between system components influences what the best partition is. For this we need fast estimators of the kind presented in this paper. [6] models communication at various levels of abstraction which enables multi-level system simulation to verify correct behavior given the selected communication components/protocols, but the question of *how* to select the best combination of communication components/protocols still needs to be addressed. Our communication model in combination with the estimation tool helps the designer/design tool answer this question.

2. The communication model

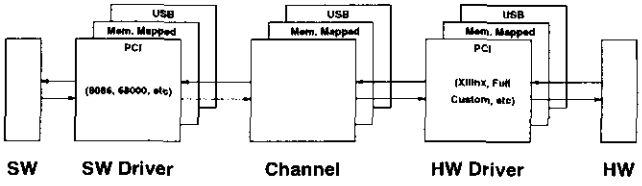

Figure 1. *Communication model overview.*

Figure 1 shows our model of point to point communication. The figure shows communication in a processor/coprocessor target architecture, but the model is not limited to this architecture – it can be used to model and estimate communication overhead in any architecture where a connection between two processing elements has been established. The time overhead of establishing such a connection (arbitration, etc.) is currently not modeled/estimated. Note that, in contrast to prior work, we consider the possible performance degradation imposed by the hardware/software drivers, and not only the characteristics of the channel.

For simplicity, we consider communication in one direction only in this paper. In general, some of the model parameters will depend on the transmission direction. For instance, a PCI bus master read is slower than a write, so the parameters that model channel transmission delay exist

1092-6100/98 $10.00 © 1998 IEEE

in both a "read" version and a "write" version in the full model.

2.1. Driver transmission delay model

Figure 2 defines the parameters which are used to estimate driver transmission delay. The driver receives n_t input

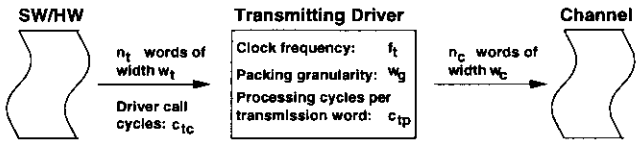

Figure 2. *Driver transmission parameters.*

words for transmission and produces n_c channel words. In order to do so, it may have to pack or split driver input words in order to fit the channel bit width w_c and it may have to perform other kinds of data processing. The packing granularity w_g influences the transmission processing delay and is defined in section 2.6. Given the clock frequency of the transmitting processor, f_t, the number of cycles, c_{tc}, it requires to call the driver for transmission (transfer arguments to, transfer execution flow to, etc.) and the number of transmission processing (packing/splitting/etc.) cycles per driver input word, c_{tp}, we can write the total driver transmission delay as

$$t_{td} = (c_{tc} + c_{tp}n_t)/f_t \tag{1}$$

2.2. Channel transmission delay model

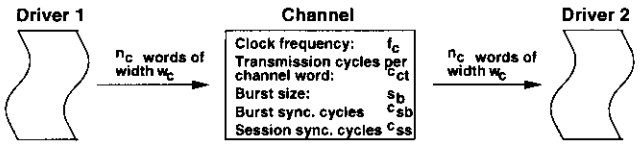

Figure 3. *Channel transmission parameters.*

Assume that the number of transmitted channel words n_c and the number of required synchronization cycles c_{cs} are known (formulas for these will be derived in sections 2.5 and 2.6). Given the clock frequency of the channel f_c and the number of transmission cycles per channel word c_{ct}, the total channel transmission delay is then calculated as

$$t_{cd} = (c_{cs} + c_{ct}n_c)/f_c \tag{2}$$

where we have assumed that a connection has already been established between transmitter and receiver and thus ignored bus arbitration and channel setup delays.

2.3. Driver reception delay model

We assume that the receiving driver in addition to the parameter n_c also receives the parameters w_t and w_g so

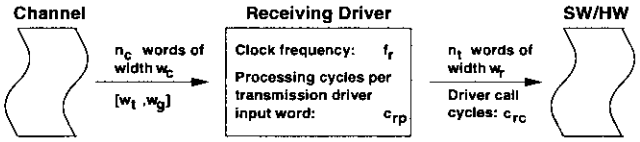

Figure 4. *Driver reception parameters.*

that it knows how data was packed by the transmitting driver[1]. We will also assume that $w_r \geq w_t$ and that each unpacked/unsplit word of size w_t is put on a single output word of bit width w_r. Given the clock frequency of the receiving processor f_r, the number of driver call cycles for reception c_{rc} and the number of reception processing (unpacking/unsplitting/etc.) cycles per *transmission driver input word*, c_{rp}, the formula for driver reception delay simply becomes

$$t_{rd} = (c_{rc} + c_{rp}n_t)/f_r \tag{3}$$

2.4. Total transmission delay

We assume that the driver production of channel words, channel transmission and driver reception of channel words occur in parallel in a pipelined fashion, which means that it is the slowest part that determines the total transmission delay t_t. We set the maximum delay to

$$t_m = \max(t_{td}, t_{cd}, t_{rd}) \tag{4}$$

and calculate the total transmission delay as

$$t_t = t_m + 2\frac{t_m}{n_t} \tag{5}$$

where the last term is an approximation of the pipeline startup/completion delay[2].

2.5. Burst mode modelling – n_c equation

The preceding sections have assumed that n_c and c_{cs} were known. This section and section 2.6 give a detailed derivation of these figures.

In order to be able to handle burst mode transfers, we model n_c to consist of $(n_b - 1)$ bursts of size s_b and a remainder burst of size s_r, $0 \leq s_r < s_b$:

$$n_c = (n_b - 1)s_b + s_r \tag{6}$$

The burst elements all have bit width w_c. We let the variable b_m denote one of three supported burst transfer types, **fixed**

[1] Of course this will not be necessary in the case where the drivers only support fixed values of w_t and w_g – these values can then be hard coded in the drivers.

[2] As the number of channel words may differ from the number of transmission/reception words, the pipeline startup/completion delay is not modeled accurately by the given term. An exact derivation is outside the scope of this paper – however, it is important to include an estimate of the delay as it may have significance for small transfers.

(each burst has a fixed size), **max** (there is a maximum on the burst size, but smaller bursts are allowed) and **inf** (there is no limit on the burst size). We can now calculate n_b and s_r as follows:[3]

$$n_b = \begin{cases} 1 & \text{if } b_m = \textbf{inf} \\ \lceil n_{cd}/s_b \rceil & \text{if } b_m = \textbf{fixed, max} \end{cases} \quad (7)$$

$$s_r = \begin{cases} s_b & \text{if } b_m = \textbf{fixed} \\ n_{cd} - (n_b - 1)s_b & \text{if } b_m = \textbf{max,inf} \end{cases} \quad (8)$$

where n_{cd} is the number of actual channel data values corresponding to the n_t driver input words of bit width w_t which have been packed/split to fit the channel width w_c. An equation for n_{cd} is derived in section 2.6.

Given the number of synchronization cycles per burst c_{sb} (possibly a fraction) and the number of synchronization cycles per transfer session c_{ss}, we can now write the number of channel synchronization cycles c_{cs} as

$$c_{cs} = \lceil n_b c_{sb} \rceil + c_{ss} \quad (9)$$

With these definitions, the equations for n_c and c_{cs} model the following four variants of burst transfers:

1. *Non-burst mode:* This is modeled by setting $b_m =$ **fixed** (or **max**) and the burst size s_b to 1 which results in $n_b = n_{cd}$, $s_r = 1$ and $n_c = n_{cd}$. The number of synchronization cycles becomes $c_{cs} = \lceil n_{cd} c_{sb} \rceil + c_{ss}$ where c_{sb} should now be interpreted as the number of synchronization cycles per channel data word.

2. *Burst mode with fixed burst size s_b.* This is modeled by setting $b_m =$ **fixed** which forces the last burst to be of size s_b regardless of how many values in that burst are actual data values.

3. *Burst mode with maximum burst size s_b.* This is modeled by setting $b_m =$ **max**. The last burst (if any) has size $s_r < s_b$, but it will still require the same number of burst synchronization cycles as the preceding bursts.

4. *Burst mode with unlimited burst size.* This is modeled by setting $b_m =$ **inf** and $s_b = 0$. Then n_b becomes 1 indicating a single burst, $s_r = n_{cd}$, indicating that n_{cd} data values are to be transferred in the single burst, and $n_c = n_{cd}$. The number of synchronization cycles becomes $c_{cs} = \lceil c_{sb} \rceil + c_{ss}$ so we only spend time on a single set of burst synchronization cycles.

Example 1: (PCI burst mode modelling). Consider a PCI bus [5] master read transaction of $n_{cd} = 1000$ words of width $w_c = 32$ on a 32 bit wide, 33 Mhz PCI-bus. The PCI-bus supports burst transfers with maximum, fixed as well as unlimited size.

[3] In the following, $\lceil x \rceil$ denotes the smallest integer larger than or equal to x (truncating upwards). $\lfloor x \rfloor$ denotes the largest integer smaller than or equal to x (truncating downwards).

We assume a maximum size ($b_m =$ **max**) burst transfer of size $s_b = 32$. This ensures a low bus latency that allows other, higher priority, units on the bus to interrupt the transfer. We assume that the bus arbitration latency is 2 clock cycles and that the bus is initially IDLE so that the bus acquisition latency is 0 clock cycles. We set slave device select (DevSel) delay to 1 clock cycle. As the address bus and data bus are multiplexed, the PCI burst transfer consists of an address transfer followed by the (up to) 32 data transfers. For a read transaction, a turnaround cycle is required between the address transfer and the data transfers in order to avoid bus contention. After completion of the burst, an additional IDLE cycle is required. The address transfer and the data transfers each last one clock cycle (assuming zero wait state transfers), except for the first data transfer which lasts 4 clock cycles. We see that the number of synchronization cycles per burst is $c_{sb} = 2 + 0 + 1$(DevSel cycle) $+ 1$(turnaround cycle) $+$ 3(extra cycles for first data transfer) $+ 1$(IDLE cycle) $= 8$. Using (7) and (8), we can now calculate $n_b = \lceil n_{cd}/s_b \rceil = \lfloor 1000/32 \rfloor = 32$ and $s_r = n_{cd} - (n_b - 1)s_b = 1000 - 31 \cdot 32 = 8$. As we set the number of synchronization cycles per session, c_{ss}, to zero, we can now use (6) to calculate the number of actually transmitted channel words, n_c, and (9) to calculate the number of channel synchronization cycles c_{cs}:

$$\begin{aligned} n_c &= (32 - 1) \cdot 32 + 8 = 1000 \\ c_{cs} &= \lceil 32 \cdot 8 \rceil + 0 = 256 \end{aligned}$$

As the number of transmission cycles per channel word is $c_{ct} = 1$, we now use (2) to calculate the channel transmission delay to

$$t_{cd} = (0 + 1 \cdot 1000 + 256 + 0)/(33 \cdot 10^6) = 38\mu s$$

□

2.6. Data packing/splitting

In this section we show how the number of channel data words n_{cd} is determined for various packing/splitting schemes.

2.6.1 n_{cd} equation: packing ($w_t \leq w_c$)

We generalize the process of packing the n_t smaller driver input words of width w_t into the n_{cd} larger channel data words of width w_c to be a two-step process:

1. First split the input words into n_1 fragments of bit width $w_g, w_g \leq w_c$. If $w_g \geq w_t$, one input word is put on each fragment as shown in figure 5.C.

2. Then pack as many as possible (n_2) of these fragments onto each channel word.

The reason for introducing the intermediate first step is that we can then model optimal as well as fast packing with the same equation, as shown below.

Each driver input word occupies $\lceil w_t/w_g \rceil$ fragments of width w_g, so we need to pack a total of $n_1 = n_t\lceil w_t/w_g \rceil$ fragments. Each channel word can hold $n_2 = \lfloor w_c/w_g \rfloor$ fragments. The number of required channel words is thus $\lceil n_1/n_2 \rceil$ which expands to

$$n_{cd} = \lceil \frac{n_t\lceil w_t/w_g \rceil}{\lfloor w_c/w_g \rfloor} \rceil, \quad w_g \leq w_c, w_t \leq w_c \tag{10}$$

Figure 5 gives an example of data packing for three different values of w_g:

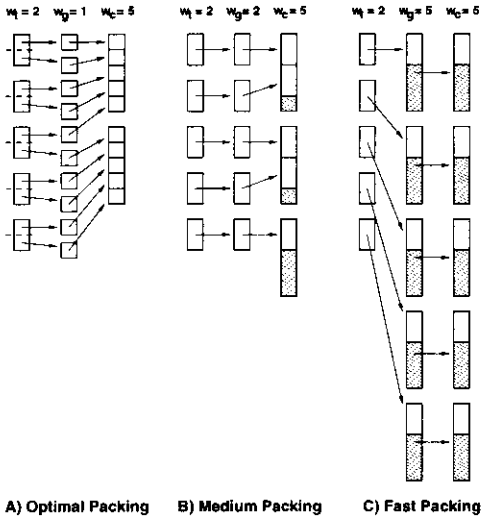

Figure 5. *Packing with different granularities.*

Optimal packing ($w_g = 1$). Optimal packing is achieved by packing the driver input words in a bit-wise manner. This corresponds to setting the packing granularity w_g to 1. Slack is only possible in the last channel word. (10) reduces to

$$n_{cd} = \lceil (n_t w_t)/w_c \rceil \tag{11}$$

Medium fast packing ($w_g = w_t$). Medium fast packing is achieved by packing the driver input words in a per input-word manner, i.e. only as many whole input words that can fit in a channel word are put on each channel word. This corresponds to setting the packing granularity w_g equal to w_t. Slack can now occur in each channel word. (10) reduces to

$$n_{cd} = \lceil n_t/\lfloor w_c/w_t \rfloor \rceil \tag{12}$$

Fast packing ($w_g = w_c$). Fast packing is achieved by packing each input word onto a single channel word. This corresponds to setting the packing granularity w_g equal to w_c. Slack will occur in every channel word if $w_c > w_t$. Naturally (10) reduces to

$$n_{cd} = n_t \tag{13}$$

2.6.2 n_{cd} equation: splitting ($w_c \leq w_t$)

Figure 6 gives an example of data splitting for two different values of w_g.

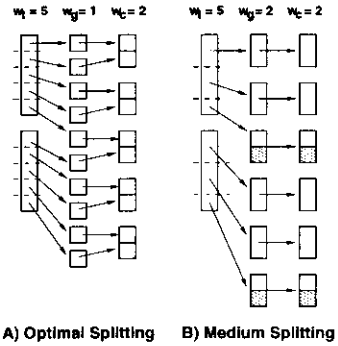

Figure 6. *Splitting with different granularities.*

We follow the same two-step approach as in the packing phase and find that equation for n_{cd} becomes identical to (10). This means that (10) covers packing as well as splitting with the only requirement that $w_g \leq w_c$.

This implies that the equation for optimal splitting ($w_g = 1$) is identical to (11) and the equation for medium fast splitting ($w_g = w_c$) is identical to (12). There is no "fast splitting" ($w_g = w_t$) case as we cannot in general fit a whole driver data word into the smaller channel words (only when $w_t = w_c$).

2.6.3 Resulting n_{cd} equation

The final equation for n_{cd} which covers both packing and splitting can now be written as

$$n_{cd} = \lceil \frac{n_t\lceil w_t/w_g \rceil}{\lfloor w_c/w_g \rfloor} \rceil, \quad w_g \leq w_c \tag{14}$$

This equation models both fast, medium fast and optimal packing/splitting, depending on the parameter w_g. The packing/splitting time in general depends on w_g, so the transmission processing delay c_{tp} in (1) and the reception processing delay c_{rp} in (3) are not actually constants but functions of w_g:

$$c_{tp} = F_{tp}(w_g), \quad c_{rp} = F_{rp}(w_g) \tag{15}$$

The communication model library should provide separate values of c_{tp} and c_{rp} for each supported value of w_g or provide the functions F_{tp} and F_{rp} as expressions.

Example 2: (Bit level serial communication modelling). We consider serial RS-232 communication using a serial

communications controller, for instance a Zilog Z8530 SCC [7] which is configured to perform 8-bit asynchronous communication using 1 stop bit and 1 parity bit. We set the baud rate to 19600, and assume that we wish to write $n_t = 1000$ words of bit width $w_t = 32$. We consider each channel data element to be a single bit, so $w_c = w_g = 1$. (14) gives us the number of channel data words, n_{cd}:

$$n_{cd} = \lceil (1000\lceil 32/1 \rceil)/(\lfloor 1/1 \rfloor) \rceil = 32000$$

We model the channel transfers to consist of bursts of size $s_b = 8$ and set $b_m =$ **fixed**. There will only be three synchronization cycles per burst (for the implicit start bit and the stop and parity bits) as there is no need to reconfigure the SCC for a write operation each time we transfer a byte and there is no delay between burst (byte) transfers as we can reload the write register while the previous byte is being transferred, so $c_{sb} = 3$. Equations (7) and (8) give us $n_b = \lceil n_{cd}/8 \rceil = 4000$ and $s_r = s_b = 8$. We assume that the SCC is already properly configured and set $c_{ss} = 0$. (6) now gives us $n_c = (4000 - 1) \cdot 8 + 8 = 32000$ and (9) gives us $c_{cs} = \lceil 4000 \cdot 3 \rceil + 0 = 12000$. Each data element (bit) transfer lasts $c_t = 1$ clock cycle and the channel clock frequency is $f_c = 19600$. We can now use (2) to calculate the channel transmission delay to

$$t_{cd} = (1 \cdot 32000 + 12000)/19600 = 2.2s$$

□

2.7. Design space exploration

The preceding examples have focused on demonstrating the modelling capabilities of the communication model. We here give an example of how the model can be used in the design space exploration phase of system level co-synthesis.

Example 3: Consider communication of ($n_t = 1000$) 16 bit words ($w_t = 16$) via an $f_c = 32$ Mhz channel of width $w_c = 32$ using a proprietary protocol with no burst mode ($b_m =$ **fixed**, $s_b = 1$) and a multiplexed address/data bus with one address transfer per data transfer ($c_{sb} = 1$) and one clock cycle per transfer ($c_{ct} = 1$). The receiving processor operates at clock frequency $f_r = 200$ Mhz.

First we consider a configuration where we use a fast (and expensive) $f_t = 100$ Mhz transmitting processor that can pack two 16 bit values on each 32 bit channel word using seven processing cycles per transmission word ($w_g = 16, c_{tp} = 7$). The receiving processor also uses $c_{rp} = 7$ cycles per transmission word to unpack the received channel words. All other parameters are set to zero. For this configuration we find that $n_{cd} = 500$, $n_b = 500$, $s_r = 1$, $c_{cs} = 500$ and $n_c = 500$ and can now calculate the transmitting driver delay, channel delay and receiving driver delay to ($\mathbf{t_{td} = 70\mu s}$, $\mathbf{t_{cd} = 31.25\mu s}$, $\mathbf{t_{rd} = 35\mu s}$). We see that the transmitting driver is the communication bottleneck ($t_m = t_{td} = 70\mu s$) and find, using (5) the resulting transmission delay to be $\mathbf{t_t} = 70\mu s + 2 \cdot (70\mu s/1000) = \mathbf{70.14\mu s}$.

We now consider a configuration where we use a slow (and cheaper) $f_t = 50$ Mhz transmitting processor that only packs one 16 bit value on each 32 bit channel word (i.e. fast packing) thus using only three processing cycles per transmission word ($w_g = 32, c_{tp} = 3$). The receiving processor also uses $c_{rp} = 3$ unpacking cycles per transmission word. All other parameters are the same as in the previous configuration. We now find that $n_{cd} = 1000$, $n_b = 1000$, $s_r = 1$, $c_{cs} = 1000$ and $n_c = 1000$ which results in ($\mathbf{t_{td} = 60\mu s}$, $\mathbf{t_{cd} = 62.5\mu s}$, $\mathbf{t_{rd} = 15\mu s}$). Here, $t_m = 62.5\mu s$ which results in a total transmission delay of $\mathbf{t_t} = 62.5\mu s + 2 \cdot (62.5\mu s/1000) = \mathbf{62.6\mu s}$.

We can conclude that in this case the best choice of transmission processor is the cheap and slow processor, even though it does not utilize the full bus bandwidth and channel transmission time is larger than before. The fact that it spends less time on packing data makes it the better choice. Though being artificial, the example demonstrates that the performance of the drivers have to be balanced with the performance of the channel in order to find the best system configuration. □

3. Conclusion

We have presented a high level communication estimation model suitable for design space exploration in co-synthesis and have demonstrated its modelling capabilities and intended use. Future work will focus on extending the model to include bus arbitration/acquisition delay in case of buses with multiple drivers and to integrate the communication estimator with partitioning and design space exploration in the LYCOS system.

References

[1] J.-M. Daveau, T. B. Ismail, and A. A. Jerraya. Synthesis of System-Level Communication by an Allocation-Based Approach. In *Eighth International Symposium on System Synthesis*, pages 150 – 155, September 1995.

[2] J. Madsen, J. Grode, P. V. Knudsen, M. E. Petersen, and A. Haxthausen. LYCOS: the Lyngby Co-Synthesis System. *Design Automation for Embedded Systems*, 2(2):195 – 235, 1997.

[3] J. Madsen and B. Hald. An Approach to Interface Synthesis. In *Proceedings of the Eighth International Symposium on System Synthesis*, pages 16 – 21, 1995.

[4] S. Narayan and D. D. Gajski. Protocol Generation for Communication Channels. In *Proceedings of the 31th DAC*, pages 547 – 548, 1994.

[5] PCI Special Interest Group. *PCI Local Bus Specification*, Revision 2.1, June 1995.

[6] J. Zhu, R. Dömer, and D. D. Gajski. Syntax and Semantics of the SpecC Language. In *Proceedings of the SASIMI Workshop*, pages 75 – 82, 1997.

[7] Zilog, Inc. *SCC/ESCC And ISCC Family Of Products User's Manual*, 1997.