Integrating communication protocol selection with partitioning in hardware/software codesign

Knudsen, Peter Voigt; Madsen, Jan

Published in:
Proceedings of the 11th International Symposium on System Synthesis

Link to article, DOI:
10.1109/ISSS.1998.730610

Publication date:
1998

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

Integrating Communication Protocol Selection with Partitioning in 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 codesign approach which incorporates communication protocol selection as a design parameter within hardware/software partitioning. The presented approach takes into account data transfer rates depending on communication protocol types and configurations, and different operating frequencies of system components, i.e. CPUs, ASICs, and busses. It also takes into account the timing and area influences of drivers and driver calls needed to perform the communication. The approach is illustrated by a number of design space exploration experiments which use models of the PCI and USB communication protocols.

1. Introduction

This paper presents an approach to hardware/software codesign which *integrates* communication protocol selection with hardware/software partitioning. The approach has been implemented as an extension to the LYCOS[6] co-synthesis system.

Most current approaches to co-synthesis consider communication synthesis to be a final step in the co-synthesis trajectory [2][3][7][8]. For instance, [2] presents communication synthesis as an allocation problem to be solved *after* system-level partitioning. However, as the level of communication overhead between system components influences what the best partition is, communication synthesis has to be *integrated* 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.

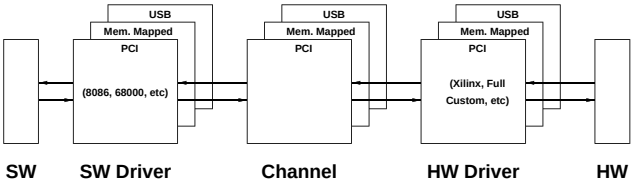

Figure 1. *Communication model overview.*

In this paper we explore how communication protocol selection influences the partitioning process. In particular, we examine a design space extended with protocol selection, area of drivers, and operating frequencies of system components. The approach is based on the communication estimation model that we proposed in [5]. This model represents 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. The structure of the model is illustrated in figure 1. The model allows for fast estimation which is important when being part of an iterative synthesis strategy exploring a very large design space.

Communication models which allow for functional verification have been proposed. For instance, [10] models communication at various levels of abstraction which enables multi-level system simulation to verify correct behavior given the selected communication components/protocols. However, the question of *how* to select the best combination of communication components/protocols together with hardware and software components still needs to be addressed. The aim of this paper is to present an approach which addresses this question.

2. The communication model

This section gives a shortened description of the communication model that was introduced in [5]. In addition, the model is extended with a communication driver area model which is given in section 2.4.

In this paper, we use the model to model communication in a processor/coprocessor target architecture as shown in figure 1, but it 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.) and the time overhead caused by bus collisions are currently not modeled/estimated. Finally 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.

2.1. Channel model

Figure 2 shows the channel model. The channel is assumed to receive n_{cd} words of width w_t from the transmit-

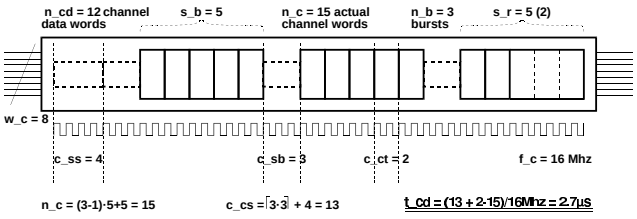

Figure 2. *Channel model*.

ting driver[1]. These words are then transmitted in $(n_b - 1)$ bursts of size s_b and a last (remainder) burst of size s_r. The number of synchronization cycles between bursts is denoted c_{sb} and the number of initial synchronization cycles c_{ss}. Each data element transfer lasts c_{ct} cycles and the channel frequency is f_t. Using these terms, the number of transmitted channel words n_c is calculated as

$$n_c = (n_b - 1)s_b + s_r \quad (1)$$

and the total number of channel synchronization cycles c_{cs} as[2]

$$c_{cs} = \lceil n_b c_{sb} \rceil + c_{ss} \quad (2)$$

The total channel transmission delay t_{cd} is now calculated as

$$t_{cd} = (c_{cs} + c_{ct} n_c)/f_c \quad (3)$$

The number of bursts n_b and the size s_r of the remainder burst depend on the channel burst mode. We model four burst modes as shown in figure 3. The "No" burst mode is

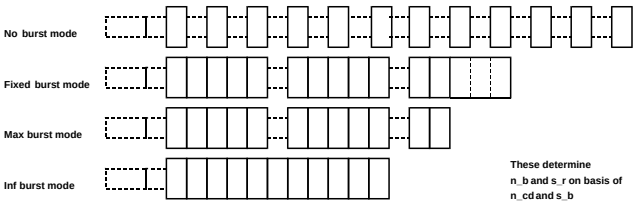

Figure 3. *Burst mode modelling.*

used when synchronization is required between each data element. The "Fixed" burst mode is used when each burst is required to have a fixed size. The last burst may contain slack (unused places) but requires the same transmission time as the other bursts. The "Max" burst mode only has a maximum on the burst size, so here slack will not occur in the last burst which can then be smaller and hence communicated faster than the other bursts. The "Inf" burst mode is used when there is no limit on the burst size so that all n_{cd} words can be transmitted in one large burst. Please refer to [5] for the equations for n_b and s_r.

2.2. Driver transmission/reception delay model

The transmission driver and reception driver models are shown in figure 4 where we assume that the software part is

[1]Throughout the paper we adopt the convention that subscripting in figures is represented by an underscore. For example, n_b in a figure corresponds to n_b in the text.

[2]$\lceil x \rceil$ denotes the smallest integer larger than or equal to x. $\lfloor x \rfloor$ denotes the largest integer smaller than or equal to x.

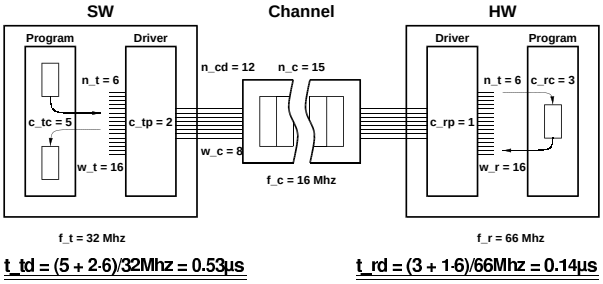

Figure 4. *Transmission/reception driver model.*

the transmitter and the hardware part the receiver. The parameters in the two models are similar, so only the parameters for the transmission driver model will be explained. The driver receives n_t transmission words from the program which spends c_{tc} cycles on the call. The n_t transmission words of width w_t are then converted to n_{cd} channel words which requires c_{tp} cycles for each of the n_t words. Using these terms, the transmission delay t_{td} is calculated as

$$t_{td} = (c_{tc} + c_{tp} n_t)/f_t \quad (4)$$

and, similarly, the reception delay t_{rd} as

$$t_{rd} = (c_{rc} + c_{rp} n_t)/f_r \quad (5)$$

Note that the receiving driver processing delay c_{rp} is the number of processing cycles per *transmission driver input word* which is why it is multiplied with n_t.

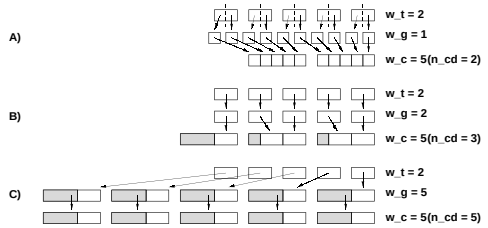

Figure 5. *Different packing granularities. A) Optimal Packing. B) Medium Packing. C) Fast Packing.*

The driver processing delays c_{tp} and c_{rp} depend on the amount of processing the driver needs to perform per driver transmission word. In [5] we show how different ways of packing/splitting data on the channel, in cases where the channel data width is different from the transmitting/receiving processor data width, may influence these numbers. As shown in figure 5 for the packing case (with $n_t = 5$), we use a parameter w_g (the *packing granularity*) to describe the degree of packing. More optimized packing will in general require more processing in the drivers so in our models, the values for c_{tp} and c_{rp} are larger for smaller values of w_g. As apparent from figure 5, the packing granularity will also influence the number of channel data words, n_{cd}, that the transmitting driver produces. Please refer to [5] for a full treatment of packing/splitting and for an equation for n_{cd}.

2.3. 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 of the three delays to

$$t_m = \max(t_{td}, t_{cd}, t_{rd}) \tag{6}$$

and calculate the total transmission delay as

$$t_t = t_m + 2\frac{t_m}{n_t} \tag{7}$$

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

2.4. Area model

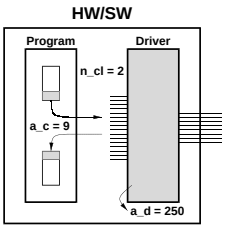

Figure 6. *Area model.*

Figure 6 illustrates the communication area model which is an addition to the communication model in [5]. a_c is the area overhead associated with each driver call, n_{cl} the number of such calls and a_d the area of the driver itself. The area unit is "bytes" if the considered processor is a software processor and an area unit decided by the communication library designer when the processor is a hardware processor. The total hardware area/software object code size associated with all communication to/from the considered processor is now calculated as

$$a_t = a_d + n_{cl}a_c \tag{8}$$

This equation captures both of the following cases:

1. *Non-inlined communication.* a_c is set to the number of bytes or the hardware area required to call the software/hardware driver and a_d is the area of the driver. This is the normal case.

2. *Inlined communication.* The driver is not actually called, but inline expanded. This also implies that c_{tc} and c_{rc} are zero. a_c will be set to the area of the driver and a_d will be zero.

2.5. Modelling Examples

Figure 7 shows how the PCI and USB channels can be modeled with our approach.

The PCI bus [9] is modeled as a 32 bit channel where each transfer takes 1 cycle. The PCI bus supports burst transfers with maximum, fixed as well as unlimited size. We assume a maximum size (**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.

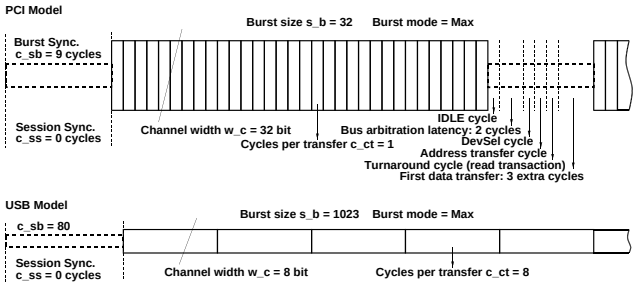

Figure 7. *PCI and USB modelling.*

The USB [1] (Universal Serial Bus) is a new personal computer interconnect that can support simultaneous attachment of multiple devices. Transactions on USB are always framed into 1ms quanta which correspond to 1500 bytes at full speed, i.e. 12 MHz. Though USB is a serial bus, we model it as a 8 bit wide channel, i.e. as being able to transfer 1 byte at a time, where each transfer then takes 8 cycles. Approximately 10 bytes (which translates to 80 cycles) are used for protocol overhead when using isochronous transfers. The maximum data payload size is 1023 bytes, so we set the burst size to 1023 and use the **max** burst mode as bursts are allowed to be smaller than 1023 bytes.

3. Design space exploration experiments

In the following, we first describe how the partitioning algorithm PACE[4] in the PALACE partitioning and design space exploration tool in LYCOS has been been extended to utilize the communication model. Then follows three experiments that demonstrate how communication protocol tradeoffs can influence partitioning and design space exploration results.

3.1. Extension of the PACE partitioning algorithm

PACE is a dynamic programming algorithm for hardware/software partitioning in the binary case, i.e. where we wish to partition an application onto a target architecture consisting of a software processor and a hardware coprocessor. The communication model can be used for more general target architectures but this simple architecture is sufficient for demonstrating the kinds of tradeoffs that are involved when protocol selection becomes part of the search space.

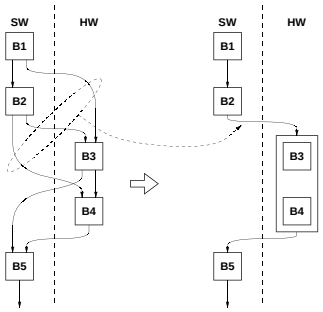

Figure 8. *Deriving effective R/W-sets for communication estimation in* PACE.

Simply put, PACE models the application as consisting of a sequence of basic scheduling blocks (BSBs) to be placed in either hardware or software[3]. The algorithm is characterized by being able to recognize that multiple adjacent blocks when put in hardware only need to communicate the *effective* read/write-sets of variables from/to software, as illustrated in figure 8. The variables communicated *between* adjacent hardware blocks are not considered to contribute to the communication overhead.

As for communication overhead calculation, PACE has been extended to utilize the new communication estimation routines when calculating the reception/transmission delays for each sequence of adjacent hardware blocks.

As for speedup calculation, the algorithm has been extended to account for the hardware and software operating frequencies, that the communication model also takes into account, so that it recognizes that better speedups are achieved when the operating frequency of the hardware part is increased.

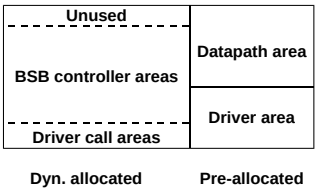

Figure 9. *Extended* PACE *area model.*

Finally, the algorithm has been extended to take account of the hardware area occupied by the hardware communication drivers and by each call to the drivers, as expressed in equation 8. Figure 9 shows the hardware model that PACE now incorporates. The full box represents the total area of the hardware chip. The right parts of the box represent preallocated area for the datapath (multipliers, adders, etc.) and for the communication driver. The left parts of the box represent the area that is available for partitioning. For each sequence of BSBs that is put in hardware, there will be a contribution associated with implementing the corresponding hardware controller and two contributions associated with the transfer of variables to/from the driver, respectively. The two last (a_c) contributions are visualized in figure 6 as well.

[3]Please refer to [4] for a more thorough description.

Note that register file area, interconnect area, etc. are currently ignored in the area model.

3.2. Experiment details

In all the experiments, we performed partitioning on the same application as in [4]. The application calculates eigenvectors which are used to obtain local orientation estimates for cloud movements in a sequence of Meteosat thermal images and consists of 448 lines of behavioral VHDL which have been converted to a Control/Dataflow Graph (CDFG) containing 1511 nodes and 1520 edges. This CDFG was automatically divided into 167 BSBs. For the software processor, we used a Motorola 68000 model and for the hardware coprocessor, we used an Actel ACT 3 FPGA library to estimate hardware datapath and controller area. The total hardware area was set to 2000 logic/sequential modules. The hardware datapath contained, among other modules, three adders/subtracters, three dividers and three multipliers and occupied a data path area of 1189. All variables are 16 bit wide. For the communication channels, we used three models:

pci-fastp which models the 32 bit wide 33 MHz PCI protocol and whose drivers incorporate "fast" packing as shown in figure 5C. The driver processing (packing) delay per transmission word was set to $c_{tp} = c_{rp} = 3$. The driver area is set to $a_d = 500$ and the driver call area to $a_c = 11$.

pci-optmp which models the 32 bit wide 33 MHz PCI protocol and whose drivers incorporate "optimal" packing as shown in figure 5A. The driver processing (packing/unpacking) delay per transmission word was set to $c_{tp} = c_{rp} = 7$. The driver area is set to $a_d = 500$ and the driver call area to $a_c = 11$.

usb which models the 12 MHz USB protocol and whose drivers split/unsplit each 16 bit transmission word into/from two 8 bit channel words using $c_{tp} = c_{rp} = 2$ cycles. The driver area is assumed to be half as big as the PCI driver and is set to $a_d = 250$ and the driver call area to $a_c = 11$.

3.3. Experiment 1

In the first experiment, the channel frequency was fixed at the "native" frequency of the chosen channel (33 MHz for PCI and 12 MHz for USB). The software and hardware frequencies were set equal to each other and set to the following frequencies in turn: 12 MHz, 33 MHz, 66 MHz and 99 MHz. In this way, the channel (at least for the higher hardware/software frequencies) can be expected to become a larger and larger bottleneck as it is the slowest part that determines the communication throughput, according to equations 6 and 7. For each of the three channels and each of the chosen software/hardware frequencies, we performed hardware/software partitioning and noted the total resulting system execution time as reported back from the partitioning tool. The result is shown in figure 10.

First we note that the USB protocol is the best choice (resulting in lowest system execution time and therefore in best

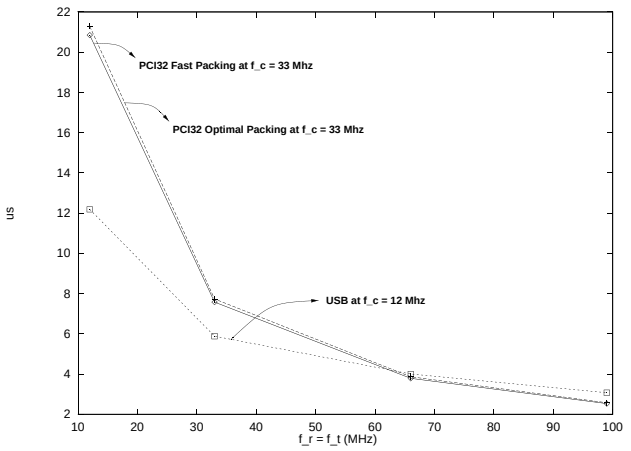

Figure 10. *System execution times for different protocols and varying but equal SW/HW frequencies.*

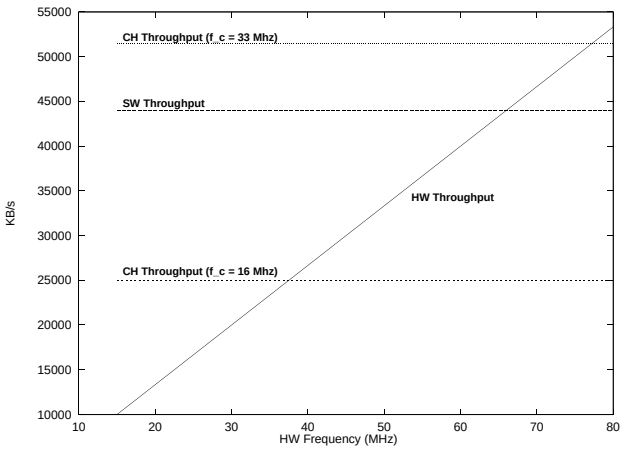

Figure 11. *SW, HW and channel throughputs.*

speedup) for frequencies up to about 66 MHz. This could be somewhat surprising as it is the protocol with the lowest throughput. The reason is that its driver area is smaller than the driver areas of the PCI protocols. This results in more available hardware chip area so that more BSBs can be moved to hardware and benefit from hardware speedup. However, as the hardware/software frequency is increased to above 66 MHz, the low throughput USB channel becomes too large a bottleneck, and the PCI channels become better choices, though the available BSB area is smaller for these.

For the PCI channels, we see that the `pci-fastp` protocol is slightly better than the `pci-optmp` protocol for all frequencies. This could also be somewhat surprising as you would expect the `pci-optmp` that packs data on the channel, and therefore has better channel throughput than the `pci-optmp` protocol, to perform better at higher frequencies where the fact that it spends more cycles on packing data is counterbalanced by the fact that the processors run at a higher speed. But this is only true if the channel is the communication bottleneck. For the PCI driver experiments, throughput data output from the partitioning tool showed that the channel was only the communication bottleneck in the `pci-fastp` 99 MHz experiment and the effect of this was negligible. For the other frequencies, the hardware and software drivers were the communication bottlenecks, and here the `pci-fastp` driver had the advantage of spending less cycles on packing than the `pci-optmp` driver.

Clearly, a tool that can help the designer analyze these tradeoffs is needed.

3.4. Experiment 2

In the second experiment, we examined the relation between driver/channel throughputs and the system execution time more closely. The experiment, which consisted of two parts, was performed for the `pci-fastp` protocol. In the first part, the channel frequency was set to 16 MHz and in the second part to 33 MHz. In both parts, the software frequency was fixed to 66 MHz and the hardware frequency was varied between 15 and 80 MHz in steps of 5 MHz.

Figure 11 shows the resulting software, channel and hardware throughputs for these experiments. These numbers were calculated by the partitioning tool on basis of equations (3), (4) and (5) (a fictive transfer of 200MB of data was used for the calculations). The software and hardware throughput graphs are the same for the two parts of the experiment as only the channel frequency is changed. As expected, the hardware driver throughput rate increases linearly with the hardware frequency. For the 16MHz channel frequency case, we get a constant channel throughput of 24976 KB/s and for the 33 MHz case, a constant channel throughput of 53333 KB/s.

As a consequence of equation 7, the effective communication throughput is approximately equal to the minimum of the software driver, channel and hardware driver throughputs. Therefore we have for the 16 MHz PCI channel frequency case that the effective throughput is limited by the "HW Throughput" graph and the "CH Throughput" graph with a cutoff at 37.5 MHz. In the 33 MHz PCI channel frequency case, the effective throughput is limited by the "HW Throughput" graph and the "SW Throughput" graph with a cutoff at 66 MHz.

The implication of this unlinear effective throughput with respect to system performance after partitioning is seen in figure 12 which shows the resulting system execution time as a function of the hardware frequency. Note that both graphs in the figure flatten out at their respective cutoff frequency where the communication overhead becomes dominant.

We see that for hardware frequencies below the first cutoff frequency of 37.5 MHz, the slow PCI channel and the fast PCI channel perform equally well. The reason for this is that the effective throughput is solely determined by the hardware driver throughput in this frequency range. So for this frequency range, the best choice of communication channel is the slow PCI channel (if price and performance are the only components of the cost function).

For hardware frequencies between 37.5 MHz and 66 MHz, the fast PCI channel is now the best choice, as the choice of that channel means that the increasing hardware driver throughput can be utilized. Choosing the slow PCI

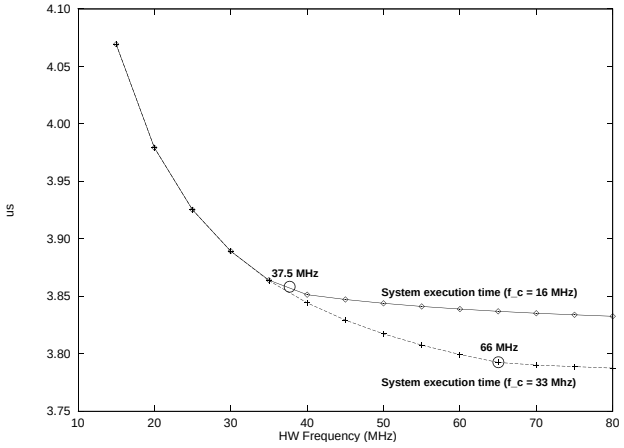

Figure 12. *System execution times.*

channel would limit the effective throughput to that of the PCI channel itself.

Above 66 MHz, the channel throughput becomes the communication bottleneck if the slow PCI channel is chosen and the software driver throughput becomes the bottleneck if the fast PCI channel is chosen. As the software driver has a higher throughput than that of the slow PCI channel, choosing fast PCI channel will also result in the best performance in this hardware frequency range.

3.5. Experiment 3

The last experiment was a simple experiment aimed at investigating the tradeoff between internal and external implementation of the hardware drivers. The previous experiments have assumed that the drivers occupied area on the hardware chip, but the designer may also choose to implement the communication drivers on a separate chip (thus obtaining a larger available area for partitioning), if the price of this is not too high compared with the performance gain.

	pci-fastp	*pci-optmp*	*USB*
Internal Driver	119.20%	114.75%	**182.22%**
External Driver	**787.35%**	646.58%	279.82%

Table 1. *Speedups for internal versus external hardware driver implementation.*

In this experiment, the software and hardware frequencies were fixed to 33 MHz and the channel frequency to the native frequency of the channel. Table 1 shows the results of partitioning sessions performed with internal versus external implementation of the hardware drivers. For the external implementation, the hardware driver areas were set to zero but the hardware driver call areas were of course still added to the BSB sequence areas.

We see that for internal implementation of drivers, we obtain the best speedup for the USB protocol. This is because the USB drivers are smaller, leaving more available area for partitioning, as we also saw in experiment 1. For external implementation of the drivers, the `pci-fastp` protocol is now the best choice as the available area for partitioning will be the same regardless of protocol choice and the `pci-fastp` protocol is the one with the best performance.

Experiments like the above can be used to guide the designer to the choice between protocol and driver implementation that gives him the best tradeoff between price and performance.

4. Conclusion

We have presented a communication model which includes both modeling of the channel and the drivers. The model has been integrated with hardware/software partitioning as to extend the design space. Design space exploration experiments have shown the importance of including a detailed communication model as well as the importance of integrating communication protocol selection with selection of hardware and software components.

In particular, the experiments have shown the important effect of incorporating the influence of drivers on area and performance. Also, we have seen that driver performance may be the actual throughput bottleneck rather than the channel. We have seen that the adjustment of the operating frequency of a single system component alone influences what the best choice of communication protocol is, in a way that it requires careful analysis to determine. Operating frequencies of system components are clearly important parts of the design space, and the impact of these must be analyzed for both system components and communication channels and utilized in the design space exploration phase to determine the best system configuration.

References

[1] http://www.teleport.com/˜usb/docs.htm.

[2] J.-M. Daveau, T. B. Ismail, and A. A. Jerraya. Synthesis of System-Level Communication by an Allocation-Based Approach. In *Proceedings of the 8th ISSS*, pages 150 – 155, 1995.

[3] M. Eisenring and J. Teich. Domain-specific interface generation from dataflow specifications. In *Proceedings of the 6th Codes/CASHE*, pages 43–47, 1998.

[4] P. V. Knudsen and J. Madsen. PACE: A Dynamic Programming Algorithm for Hardware/Software Partitioning. In *Proceedings of the 4th Codes/CASHE*, pages 85 – 92, 1996.

[5] P. V. Knudsen and J. Madsen. Communication Estimation for Hardware/Software Codesign. In *Proceedings of the 6th Codes/CASHE*, pages 55 – 59, 1998.

[6] 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.

[7] J. Madsen and B. Hald. An Approach to Interface Synthesis. In *Proceedings of the 8th ISSS*, pages 16 – 21, 1995.

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

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

[10] 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.