Hardware/software partitioning of embedded systems with multiple hardware processes

D.C. Hendry
D.S. Sananikone

Indexing terms: Embedded systems, Hardware/software partitioning, Electronic design automation

Abstract: COSYN is an hardware/software partitioning tool for embedded systems with multiple hardware processes. It produces a system hardware/software partition whilst satisfying system constraints, where feasible. COSYN uses a CIPN (coloured interpreted Petri net) to model systems, and to provide simulation data for the partitioning algorithm. The CIPN provides modelling capability for multiple processes and interprocess communication primitives. The partitioning algorithm, which adopts a fine-grained approach to system partitioning (it considers moving nodes at the basic block level), is based on selecting blocks based on their potential speedup and extra hardware requirements, using hardware and software execution time estimators. The interdependence between interprocess communication primitives is exploited to achieve a better hardware/software partition. Results for an input example pdi using different partitioning algorithms are given which illustrate the benefits of the approach presented in this paper.

1 Introduction

Improvements in VLSI technology and increasing use of electronic design automation (EDA) tools in hardware design has brought about an increase in the number of application-specific hardware components in systems design. Functions which were traditionally implemented in software can now be implemented in hardware [1, 2]. This shift in functionality brings about the need for a new approach to systems design to aid the designer in choosing between different implementation methods for particular functions.

Hardware/software cosynthesis [3], or cosynthesis, addresses the following issues:

• the issue of complexity. With the advent of EDA tools in the hardware domain (from transistor level tools, through logic level synthesis tools, and finally to high level synthesis tools) the trend has been to provide tools to address complexity at increasing levels of abstraction. The logical step is the development of system level synthesis tools, or cosynthesis tools.
• the issue of a unified hardware/software design methodology. With the shift in hardware/software functionality and increasing performance expectations the design of a system becomes much harder. Thus tools are required to aid the designer in making design decisions.
• the increasing need to reduce design times, and increase reliability. The use of a unified framework for systems design increases interaction and understanding between hardware and software components in all stages of design.

There are four major issues in cosynthesis: cospecification, cosynthesis framework, system partitioning, and cosynthesis. In this paper we will discuss the system partitioning problem, and COSYN, a tool developed to address the partitioning issue.

2 Related work

Ernst, Henkel, and Benner [4] use the terms fine-grain partitioning and coarse-grain partitioning to categorise different approaches to partitioning. Fine-grain partitioning refers to partitioning at the basic-block level or statement level; coarse-grain partitioning refers to partitioning at the task or process level. Srivastava and Brodersen [5], and Athanas and Silverman [6] use the coarse-grain approach to partitioning. Of more interest, however, are those systems which adopt a fine-grain approach to partitioning; these systems provide a more in-depth exploration of design space and thus offer more potential for better quality designs. These are now discussed.

Ernst, Henkel, and Benner [4] introduce Cosyma, a cosynthesis system for small embedded controllers. The target architecture consists of a Sparc processor communicating with a synthesised coprocessor; the processor and coprocessor execute in mutual exclusion. The system is specified in C, a superset of C, which allows for timing features, task concepts, and task communication. The algorithm uses a simulated annealing algorithm to perform partitioning at the basic block level. Ernst et al. have termed their approach hardware extraction. They use a partitioning cost function that favours functions that can be implemented well in hardware. Hardware is added to the initial software implementation in order to meet performance constraints.

Gupta and Micheli [7] use a hardware-oriented approach to system partitioning. The initial partition is
predominantly hardware implemented. Functions are then moved into software in order to reduce system cost. The target architecture is similar to that used by Ernst et al. Initially all internal points of nondeterminism (data-dependent loops and conditionals) are implemented in software; the remainder of the system is implemented in hardware. Areas of hardware are then implemented in software based on a greedy approach to node selection. However, because all internal points of nondeterminism are implemented in software, this approach is restricted in the nodes it may select.

Knudson and Madson present PACE [8], an hardware/software partitioning algorithm which is used in the LYCOS [9] cosynthesis system. The target architecture they have adopted consists of a single processor communicating with a single ASIC (application specific integrated circuit). The functional specification, described in VHDL (VHSIC hardware description language) or C, is translated into BSBS (basic scheduling blocks). The partitioning algorithm considers all sequences of BSBS, and chooses the combination of sequences that induces the largest speedup. The speedup induced by moving a BSBS sequence to hardware is determined by the difference between the hardware execution time and the software execution time minus the effects of variable communication between the hardware and software component. The time required for variable communication is a product of the number of variables to be communicated and the time taken to move that variable. The partitioning model, however, only considers the case where software and hardware components execute in mutual exclusion.

Jantsch et al. [10] introduce an algorithm for hardware/software partitioning and rapid prototyping. Behaviour is specified in C or C++, and the system is partitioned into a software and hardware component implemented on a Sparc processor and a Xilinx FPGA (field programmable gate array), respectively. Blocks are chosen for implementation based on estimated hardware cost and potential speedup if implemented in hardware. Communication overhead, however, is ignored. The partitioning algorithm is resource constrained; the algorithm finds the blocks which would gain the greatest speedup while still fitting onto the available hardware.

3 Overview of COSYN

An area of interest to codesigners is embedded systems design. Embedded systems are characterised by their use of multiple processes and interprocess communication primitives; often they must perform within a certain time limit. When system requirements exceed the performance available from a software solution a mixed hardware/software system must be used. The approach adopted by COSYN identifies nodes for implementation in hardware based on the potential speedup to be gained, but also utilises the interdependencies that exist between interprocess communication primitives. The original features of COSYN are

- the use of a graph to model the dependency between interprocess communication primitives, and
- a partitioning algorithm which exploits the interprocess communication dependency to achieve a better hardware/software partition.

A system behavioural description is partitioned by COSYN into a software and hardware component, implemented by a single processor and an ASIC or FPGA, respectively. First, the input is translated into basic blocks (these basic blocks consist of three address code (TAC) which specify the behaviour of the block). This is then translated into a coloured interpreted Petri net (CIPN) to model the system. Starting from an initial software solution the system is then iteratively partitioned by moving blocks into hardware until system constraints are met. COSYN selects blocks based on their potential speedup when implemented in hardware. It also takes into account the dependencies that exist between interprocess communication primitives. Block execution times and speedup times are generated by hardware and software estimators. Communication overhead is also taken into account. Fig. 1 shows an overview of COSYN.

![Fig. 1 Overview of COSYN](image)

The input language used to specify the system is a subset of C with extensions to model important features of embedded systems. These are as follows:

- Multiple processes.
- Interprocess communication; Ada like semaphores and rendezvous' primitives are used.
- Timing constraints; these are maximum time, monitor, and data rate constraints. The maximum time constraint constrains the system to execute within a maximum bound; the monitor constraint constrains a section of code to execute within a maximum time; the data rate constraint constrains the rate of data consumption/production to a minimum bound.

4 Target architecture

The target architecture is based on a Motorola MC68040 processor running on a multitasking kernel, executing concurrently with an hardware component. The hardware component may be an ASIC or an
FPGA; COSYN is not targetted towards a particular technology but rather uses the execution time of the functional units in a specified hardware library to determine performance, and thus drive partitioning. In addition, the hardware component is partitioned into processes which can execute concurrently. This is in distinction to the systems presented by Ernst et al. [4], Knudson and Madson [8], and Jantch et al. [19], where the hardware and software component execute in mutual exclusion.

Hardware processes are managed through the use of exception routines and interrupts; the processor initiates an hardware process by executing an exception routine which starts the relevant hardware process; at the end of execution the hardware process generates an interrupt which signals to the processor that the process has finished executing. An interrupt controller and bus arbiter are also specified.

Processor instructions and data are stored in memory accessible by both processor and hardware component. The hardware component also has access to local memory in which data used exclusively by itself is stored.

5 Simulation model

From a behavioural description of the system COSYN synthesises an architectural partition. It uses a model to simulate the system; the model captures system behaviour, and provides performance data for the partitioner.

The choice of a suitable model is dependent on satisfying various criteria:

- It must be capable of providing the necessary timing information so that the partitioner is able to make the necessary partitioning decisions.
- It must permit modelling of both hardware and software processes.
- It must adhere to the data dependencies within a process.
- It must be able to represent communication between processes.

The model must also be able to provide performance statistics on the behaviour of the system. To do this it must have a notion of time. FSMs (finite state machines) and CDFGs (control data flow graphs) do not represent timeliness; FSMs consist of states and transitions, but there is no notion of time; CDFGs consist of control and data dependencies within a graph, but again time is not taken into account. Petri nets, however, are able to model the passing of time and thus are able to provide timing information making them suitable for modelling system behaviour. Petri nets represent a graphical and mathematical tool with which to model systems. They have found use in many applications including performance evaluation [11], communications protocols [12], analysis of multiprocessor systems [13], and concurrent and parallel programs [14]. Briefly, a Petri net can be specified as a 5-tuple \((P, T, I, O, M)\), where \(P\) is the set of places, \(T\) is the set of transitions, \(I\) is the set of input arcs from places to transitions, \(O\) is the set of output arcs from transitions to places, and \(M\) is the marking which defines the number of tokens in each place in the Petri net. A transition has a set of input places which correspond to the places which have arcs directed to the transition, and a set of output places which correspond to the places which have arcs directed from the transition. A transition is enabled when all the input places to the transition have at least one token. The transition is then fired by removing tokens from the input places and placing tokens into the output places.

![Fig. 2 PN translation for interprocess communication primitives](image)

a Semaphore WAIT

b Semaphore SEND

![Fig. 3 PN translation for interprocess communication primitives](image)

a Rendezvous request

b Rendezvous accept
c Rendezvous end
In order to model the system a colour interpreted Petri net (CIPN) is introduced. CIPNs merge two classes of Petri net: interpreted Petri nets [15], and coloured Petri nets [16]. Interpreted Petri nets (IPNs) are an extension of the Petri net concept and allow the behavioural and time modelling of the system; coloured Petri nets are an abbreviation of the Petri net concept and give the ability to model complex systems in a more manageable way. Combining the two gives the ability to model the system behaviour and the interaction between processes, and allows timing measurements to be taken.

From the input description a set of basic blocks [17] is generated. The basic blocks are then transated into a CIPN. For most basic blocks there is a direct mapping to the underlying CIPN. For the interprocess communication primitives, however, the CIPN translation is illustrated in Figs. 2 and 3.

The model is split into three parts: Behaviour, Interface, and Communication. The Behaviour part represents the behaviour of the processes; the Communication part represents the communication primitive netlists (see Figs. 4 and 5); the Interface part represents the interface between the Behavioural part and Communication part.

For each communication primitive a netlist is generated. Figs. 4 and 5 show the semaphore and rendezvous netlists, respectively.

Simulation is initialised by depositing tokens into the places denoting the start of each process. When a token is deposited onto a place the set of operations associated with it are executed. After a delay (determined by the delay associated with the place), the token becomes available. Simulation is advanced in discrete time steps; the time step is determined by the smallest remaining firing delay time of a place in the marking. (The marking of the PN defines the number of tokens in each place. The remaining firing delay of each place denotes the remaining time before the token becomes available. At each time step the set of enabled transitions is generated. Tokens are removed from the input places of the transitions, and placed into the output places. Simulation continues until the set of generated transitions is null.

The partitioning algorithm uses the activation frequency and token residence time of each place, generated by the simulation, to determine nodes for implementation in hardware.

### 6 Estimation tools

COSYN uses estimation tools to generate block execution times. The estimation tools give a fast, but approximate execution time for the block.

The software execution time for a basic block is determined by calculating the sum of the execution times for each TAC statement within that block. A processor characteristics file is used to generate execution times depending on the data types within the three address code.

The hardware estimator determines the execution time of the block and the number of functional units used. An ASAP scheduler is used to determine the minimal number of C-steps (control steps) required for the TAC sequence corresponding to that block. Within the
schedule each three address code statement has a start and stop operation time and three external bus flags: A_WRITE, B_READ, and C_READ. B_READ and C_READ are set if the operands are variables which are to be read in from external memory before the operation; A_WRITE is set if the result is to be written to external memory after the operation (see Section 7.4 for a discussion on the setting of these flags). With respect to the operation times and bus flags an external bus utilisation graph is generated. Fig. 6a and b show a sample bus utilisation graph (a bus read/write will be assumed to take 3 C-steps in this example).

Fig.6 External bus utilisation graph
+ bus free, * bus contention

Fig. 6a shows the bus utilisation from the information provided in the TAC sequence. From the graph, bus contention occurs three times within the schedule (a bus contention occurs when a bus must be used for reading and writing at the same time). For an operation requiring a bus read and a bus write cycle the read cycle must be performed before the operation; the write cycle must be performed after the operation. Fig. 6b shows the final schedule taking into account bus contentions. In the final schedule four C-steps have been added to take into account variable communication; to remove bus contentions the bus read cycles have been advanced while the bus write cycles have been delayed.

7 Partitioning

The partitioning stage attempts to partition the design into an hardware and software component given a set of system constraints. The partitioning stage consists of an iteration of three phases: simulation, analysis, and partitioning; the partitioning stage ends when system constraints are met. From data generated by the simulation of the CIPN, COSYN moves places into hardware based on their potential speedup and frequency of activation (the number of times tokens have been deposited on that place). The extra functional unit area required to implement the place in hardware is also taken into account. The places considered for implementation in hardware are generated by the union of the set of violated constraints. The algorithm also addresses the issue of interprocess communication interdependence by generating associated sets for interdependent communication primitives, and using these sets during the analysis and partitioning stage. The following describes the set generation for system constraints and interprocess communication primitives, and describes the partitioning algorithm used.

7.1 Set generation

For every constraint a set of places is generated. These sets are used by the partitioning algorithm to select places for implementation in hardware. The associated sets are generated as follows:

- Overall time constraint. Each process has a set of places S corresponding to it. For each process i which completes execution after the time limit set by the Overall time constraint the set of places S_i is added to OT, the set of places corresponding to the Overall time constraint.
- Monitor constraints. For a monitor constraint i, M_n, the set of places for Monitor n is composed of all the places occurring between the two Monitor delimiters.
- Data rate constraints. For a data rate constraint i the set D_i consists of places such that p_j exists in D_i where there exists a path from p_j to p_k (a path exists between place A and place B if there exists directed arcs which connect place A to place B; the path may span places and transitions) and there exists a path from p_k to p_l (p_k is the place where the data rate constraint is specified). The set of places P used by the partitioning algorithm is the union of the sets for violated constraints.

7.2 Interprocess communication interdependence

Sets are also generated for the interprocess communication primitives. These are used to generate associated sets in the partitioning phase. The following lists the set generation for each primitive:

- semaphore wait. Consider two processes using semaphore based interprocess communication, as shown in Fig. 7.

```
<table>
<thead>
<tr>
<th>process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>for(a=0;1)</td>
</tr>
<tr>
<td>wait(S)</td>
</tr>
<tr>
<td>send(S)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>process 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>for(t=0;1)</td>
</tr>
<tr>
<td>wait(S)</td>
</tr>
<tr>
<td>send(S)</td>
</tr>
</tbody>
</table>
```

Fig.7 Semaphore communication between two processes

The two processes use the semaphore variable S to determine control over execution in the critical section. If S is larger than 0 the process has execution control; else it must wait until S becomes larger than 0. The only point when the semaphore for Process1 is not available is when process2 is in execution block2; when it is in execution block1 the semaphore is available and Process1 is not required to wait. Therefore the associ-
ated set of places for the place representing the 
WAIT(S) statement in Process1 is the set of places cov-
ered by execution block2 in Process2, and vice versa.
• rendezvous request. Consider a process, process1 
requesting a rendezvous with process2. Fig. 8 shows the 
accept rendezvous statement for process2.

![Fig. 8 Process2 accepting a rendezvous](image)

If process1 is waiting for a rendezvous with process2 
the only point at which process1 is given permission to 
continue execution is at the end of execution block1, 
when process2 executes an ENDRENDZEVOUS state-
ment. Therefore if, at any time, process1 requests a 
rendezvous with process2 it must wait until the end 
of execution block1. In the worst case both block1 and 
block2 must be executed before an ENDRENDZE-
VOUS signal is issued; in the best case the process does 
not need to wait at all (in this case a set does not need 
to be generated as the nondeterministic place will not 
be chosen for hardware implementation). The set of 
places will be generated assuming the worst case; that 
is the set of places covered by block1 and block2.

![Fig. 9 Process requesting a rendezvous](image)

- rendezvous accept. When a process executes a ren-
dezvous accept statement and there are no rendezvous 
requests pending it must wait until a request requests a 
rendezvous. Fig. 9 shows a process requesting a ren-
dezvous.

In the worst case process1 must execute the whole of 
block1 before issuing a rendezvous request; in the best 
case the accepting process does not need to wait. 
Assuming the worst case the set of places associated 
with the accept statement is the set of places covered by 
execution block1.

(Note: with respect to Fig. 9 a place within process1 
is considered to be within the scope of the nondetermi-
nistic place if it exists within execution block1.)

7.3 Communication rules

In addition to the generation of sets for interprocess 
communication primitives, a set of communication 
rules will also be established. The communication 
rules are generated during simulation and determine which 
processes are communicating using a particular inter-
process communication primitive. For example, Fig. 10 
shows the communication rules for three communicat-
ing processes.

![Fig. 10 Example of system with three communicating processes](image)

The communication rules specify which processes 
communicate with each other. This ensures that a non-
deterministic place does not pick up any incorrect asso-
ciated sets. For example, if the associated set for 
process A was required the only set to be picked up 
would be that belonging to process B as the communi-
cation rule A → C does not exist (see Section 7.4). (In 
this example the communication rules are shown as 
pairs of processes; however it is more accurate to show 
them as pairs of interprocess communication primi-
tives.)

7.4 Partitioning algorithm

The partitioning algorithm uses data collected from 
simulation of the Petri net. For nondeterministic places 
the total token residence times for the simulation are 
used; for deterministic places the frequency of activa-
tion is recorded. In addition, for deterministic places 
which are contained within the scope of a nondetermi-
nistic place (see Section 7.2) the activation frequency 
for each place relating to a particular communication 
primitive is recorded. Thus, with respect to Fig. 10, the 
places within the scope of the nondeterministic place 
within process C will have an activation frequency per-
taining to communication with process A, and an activ-
ation frequency pertaining to process B.

The partitioning algorithm is as follows:

- \( P_d \) is the set of deterministic places
- \( P_n \) is the set of nondeterministic places
- \( P = P_n \cup P_d \)
- \( P_{(T)} \) is the total time of the place, \( P_i \in P \)
- (the total token residency time for that place)
- nodes_move is the number of places to be moved at 
each partitioning stage
partition
simulate PN
while constraints not satisfied
    generate P
    nodes_selected = 0
    while nodes_selected < nodes_move
        if  \( \forall P_d, P_n \quad P_{d IT} > P_{n IT} \)
            \( P_i = \text{find_place}(P_d) \)
        else
            \( P_i = \text{find_n_d_place}(P_n, P_d) \)
        move \( P_i \) into hardware
        nodes_selected = nodes_selected + 1
        \( P = P - P_i \)
    regenerate_bus_flags()
simulate PN

The algorithm iteratively selects places to be moved into hardware until performance constraints are met (or until no places can be moved into hardware). It is assumed that the hardware component is large enough to accommodate all the places moved into hardware; the hardware estimator determines functional unit area only, not total silicon area. The nodes move parameter determines the granularity of the partitioning algorithm; the more moves the closer the granularity. Places are first split into nondeterministic sets and deterministic sets. If the place with the highest total time belongs to the deterministic set, a place from that set is chosen for implementation in hardware by the function \( \text{find place}(P_d) \). Each place in the set \( P_d \) is given an index; the place with the highest index is chosen. The index for each place is given by eqn. 1.

\[
P_{\text{index}} = (P_{\text{sD}} - P_{\text{dD}}) \times \frac{P_{\text{freq}}}{\text{weight}(P_i)} \tag{1}
\]

where \( P_{\text{sD}} \) and \( P_{\text{dD}} \) are the software delay and hardware delay, respectively, of the place; \( P_{\text{freq}} \) is the activation frequency of the place. The function \( \text{weight}(P_i) \) penalises places which introduce extra functional units by weighting the place index.

If the selected place \( P_i \) belongs to the nondeterministic set, the recursive function \( \text{find_n_d_place}(P_n, P_d) \) is called and returns a place to be implemented in hardware; the nondeterministic place \( P_i \) is also added to the list primitive_list. (Primitive_list is a list of nondeterministic places which have been selected by the \( \text{find_n_d_node} \) function, and is used in \( \text{find_n_d_node} \) to determine interprocess communication pairs). The function finds the associated sets belonging to the nondeterministic place \( P_i \) using the communication rules. The associated sets are split into two sets and a deterministic set. If the place with the highest total time belongs to the deterministic set a place from that set is chosen (using the index function in eqn. 1). If the place (say, \( P_j \)) belongs to the nondeterministic set and the place is not in primitive_list \( \text{find_n_d_node} \) is called; if the place has an entry in primitive_list the process intercommunication pair \( (P_i, P_j) \) is generated.

The associated sets for \( P_i \) and \( P_j \) are generated and a place is selected from the union of the two sets. For each place \( P_k \) an index is generated as given in eqn. 2.

\[
P_{\text{index}} = (P_{\text{sD}} - P_{\text{sD}}) \times \frac{\text{find_frequency}(P_k)}{\text{weight}(P_k)} \tag{2}
\]

the \( \text{find_frequency}(P_k) \) function finds the activation frequency of the place for the interprocess communication pair. For example, if place \( P_k \) is within the scope of \( P_i \) the function finds the activation frequency of \( P_k \) when \( P_i \) is communicating with \( P_k \).

The regenerate_bus_flags() function sets the bus flags for places implemented in hardware. For a place \( P_i \) selected for implementation in hardware, groups of hardware places preceding the place and proceeding the place are examined, and bus flags are set accordingly. Hardware places are then rescheduled to take into account the change in communication overhead.

(Note: regenerate_bus_flags does not provide a global analysis of variable communication; it considers only groups of adjacent hardware places. It does not take into account hardware groups spanning a group of places implemented in software. This simplifies partitioning at the expense of a less accurate estimation of communication overhead.)

To compare the effect of nondeterministic places on partitioning another algorithm will be specified; this is essentially the same as the algorithm described earlier, but nondeterministic places are not included in the sets, and thus the interdependency between interprocess communication primitives is not taken into account. The first algorithm will be called ND Node; the second will be called Simple.

8 Results
A modified form of the pdi [18] algorithm is used as an example for COSYN. Written using over 1200 lines of code, it contains four processes communicating via semaphores and rendezvous. Figs. 11-14 show the simulation results from the Simple and ND Node algorithms for a desired speedup of 2. (The overall time constraint is set to half the execution time of the initial software partition. The number of places selected per partitioning cycle was set to 1). The x-axes represent the number of simulation runs; the y-axes represent

- input description run-time. This shows the simulated execution time of the system to be partitioned.
- variable communication. The number of times variables must be communicated between hardware and software are shown.
- extra C-steps required. The number of C-steps required to communicate data between hardware and software are shown for each process.
- functional unit area. This shows the functional unit area required.

As can be seen from Figs. 11-14 the input description run time is initially reduced considerably but then as partitioning proceeds the gain in performance by adding hardware is notably reduced; it becomes harder to find performance gains as partitioning proceeds. The variable communication, and extra C-steps required, also increases as partitioning proceeds. The troughs in the curve indicate a software node, which has neighbouring hardware nodes, being implemented in hardware. The effect of this is to reduce variable communication between the hardware and software component and thus reduce communication overhead.

The final partition yields the following results shown in Table 1.

The results given in Table 1 indicate that the final partitions generated by both algorithms are very similar. The partition generated by the ND Node algorithm, however, is considered a ‘better’ partition as its func-
Fig. 11 Results for ND Node algorithm
- Input description time
- Run time
- Variable communication during simulation
- Process 1, + process two
- Process 3, × process four

Fig. 13 Results for Simple algorithm
- Input description time
- Run time
- Variable communication during simulation
- Process 1, + process two
- Process 3, × process four

Fig. 12 Results for ND Node algorithm
- Extra control steps required for communication O/H
- Process 1, + process two
- Process 3, × process four
- FU area of hardware component
- FU

Fig. 14 Results for Simple algorithm
- Extra control steps required for communication O/H
- Process 1, + process two
- Process 3, × process four
- FU area of hardware component
- FU
tional unit area is smaller and only two hardware processes have been generated.

### Table 1: Partitioning results for Simple and ND Node algorithm

<table>
<thead>
<tr>
<th></th>
<th>Simple</th>
<th>ND Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simulation runs</td>
<td>62</td>
<td>61</td>
</tr>
<tr>
<td>Functional unit area (units$^2$)</td>
<td>12</td>
<td>10</td>
</tr>
<tr>
<td>Number of hardware processes</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

The similarity in results can be attributed to the fact that the major interaction is between two processes communicating via semaphores. It is natural that both algorithms select the same places as

- in the case of the Simple algorithm, the most time intensive segments of code occur within these two processes.
- in the case of the ND Node algorithm, the place with the highest total time is nondeterministic, and is dependent on the communication between the two processes described earlier. Thus, a place selected for implementation in hardware will be most likely to come from those two processes.

This similarity is highlighted by the fact that of the 61 places selected by the Simple algorithm, 59 places are similarly selected by the ND Node algorithm (albeit in a different order). For the input example pdi applying an overall time constraint to the system ensures that the Simple algorithm always includes the two processes in its place set. By applying a Monitor constraint to the system, the deficiencies of the Simple algorithm will be highlighted.

The use of a Monitor constraint constrains a particular section of code to a maximum execution time. If a system is specified in which the overall execution time is unimportant, but a particular function is required to execute within a particular time limit then a Monitor constraint is placed upon it. To emphasise the difference between the two partitioning algorithms a Monitor constraint is applied to a process in pdi.

The final partition yields the following results shown in Table 2.

### Table 2: Partitioning results for Simple and ND Node algorithm

<table>
<thead>
<tr>
<th></th>
<th>Simple</th>
<th>ND Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simulation runs</td>
<td>9</td>
<td>11</td>
</tr>
<tr>
<td>Functional unit area (units$^2$)</td>
<td>29</td>
<td>4</td>
</tr>
<tr>
<td>Number of hardware processes</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Although the ND Node algorithm generates two hardware processes compared with the one generated by the Simple algorithm its functional unit area is considerably smaller. The place set generated by the Simple algorithm includes all places within the constraint and so all places to be implemented in hardware are chosen from that set, the ND Node algorithm uses the same initial place set, but also potentially uses the place set associated with the interprocess communication primit-

### 9 Partitioning granularity

Varying the number of places to be implemented in hardware with each partitioning cycle determines the degree of granularity of the partitioning approach (increasing the number of places makes the partitioning coarser-grained; reducing the number of places makes the partitioning finer-grained). This section deals with the effect of adopting a finer-grained approach, and a coarser-grained approach to partitioning.

The results given earlier were achieved by using a one place per partition cycle (one place to be selected for implementation in hardware per partition cycle), and represents the finest-grain approach possible for COSYN. Increasing the number of places per partition cycle produces a coarser-grain approach and leads to a partition that is, at best, the same as the finest-grain approach.

Consider the use of an $n$-place per partition cycle ($n$ is larger than one). After simulation the first place is selected for implementation in hardware. Subsequent places are selected based on information from the previous partition. However, since a place has already been selected for implementation in hardware, the performance data used to select the extra places are now outdated; the performance data relate to the partition before the first place was selected and do not take into account the effect of that place on system performance. This leads to the postulation that the higher the $n$ the ‘worse’ the final partition. To substantiate this COSYN will be run with varying $n$ for pdi with a desired speed-up of 2. Table 3 shows the results.

### Table 3: Partitioning results for varying n-places per partition cycle

<table>
<thead>
<tr>
<th></th>
<th>Places per partition cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Simulation runs</td>
<td>61</td>
</tr>
<tr>
<td>Number of places selected</td>
<td>60</td>
</tr>
<tr>
<td>Functional unit area (units$^2$)</td>
<td>10</td>
</tr>
</tbody>
</table>

From the Table it can be seen that by increasing $n$, the functional unit area and places selected increase, but partitioning time is shortened due to a reduction in number of simulations required; in the $n = 20$ case, there are 12 times less simulations carried out compared to the $n = 1$ case, but functional unit area and number of places selected are larger.

This situation can be improved by varying $n$ for each partitioning cycle. By choosing an initial number of nodes to be implemented (in this case, 20), and then reducing that amount at each partitioning stage, a higher quality design should result compared with the case where $n$ is constant. As partitioning proceeds, the number of nodes selected per partitioning cycle is reduced, thus reducing the probability that more nodes than necessary are selected for implementation in hardware. Table 4 shows the results for different functions $(f(n, s))$ gives the number of nodes to be selected at
The first three functions describe a linear reduction in number of nodes selected; the fourth function describes a geometric reduction. From the Table, the number of simulations required is much lower compared to the \( n = 1 \) case. In fact, the final results for three of the four functions produced comparable results to the \( n = 1 \) case. Choosing a function, however, is not simple, in general. The improvement in performance is not linear to the number of nodes selected, thus there is no relationship between the rate of decrease of nodes selected, and increase in performance. Also, there is no way of knowing whether the design is as good as the finest-grain approach. However, this method does give a solution, where feasible, which takes less simulation time than the finest-grain approach, and produces at least a better partition compared with the case where the number of nodes selected is constant and equal to the initial number of nodes selected.

**10 Conclusions**

The COSYN tool for hardware/software partitioning has been presented. To model the system a CIPN, a class of Petri net which merges coloured Petri net and interpreted Petri net classes, has been introduced. It has been shown that it is feasible to use a CIPN to model multiple processes and interprocess communication, and to provide performance data. The superiority of the partition generated by the ND Node algorithm shows that an approach which takes interprocess communication interdependence into account yields better results than an approach which does not consider the effects of interprocess communication. The effect of granularity on the hardware/software partition was also considered. Results indicate that a finer-grained partitioning approach gives better partitions, but simulation time is higher compared with coarser-grained partitioning approaches. The large simulation time, in relation to coarser-grained approaches, was addressed by setting the number of nodes per partition cycle to an initial value, and then decreasing that number as partitioned progressed. It was concluded that this method produced a result which improved simulation time compared with, and also generated a design which approached the quality produced by, the finest-grain approach.

Future work includes the development of better hardware estimators, use of different architectures, and constraint partitioning using other metrics such as power consumption.

**11 References**

9. ‘LUCS user’s guide’. Department of Computer Science, Technical University of Denmark, DK-2800, Lyngby, Denmark, 1995 (http://www.it.dtu.dk.lucas/users_guide)