Modular Construction of Model Partitioning Processes for Parallel Logic Simulation

Klaus Hering   Gudula Rünger   Sven Trautmann*
Fakultät für Informatik, Technische Universität Chemnitz, Germany
{hering,ruenger,sven.trautmann}@informatik.tu-chemnitz.de

Abstract

Logic simulation of a complex processor model in VLSI design is very time consuming. One possibility to increase the simulation speed is to partition the processor model and assign the resulting parts to simulator instances that cooperate over a loosely-coupled system. For corresponding model partitioning processes, we have developed a distributed framework parallelMAP *implementing a hierarchical partitioning strategy. It is intended to be used as production environment in VLSI design as well as an experimental test bed for algorithm development. In this paper we describe the possibilities* parallelMAP *offers for the modular construction of partitioning processes starting from a set of basic sequential and parallel modules. Experimental experiences are given with respect to IBM processor models comprising from* $1.5 * 10^5$ *to* $2.5 * 10^6$ *elements at gate level.*

1 Introduction

The complexity of Very Large Scale Integration (VLSI) design is growing rapidly. Therefore the use of verification tools is unavoidable in all design phases. Logic simulation is a very time consuming verification method for the design of processor structures. There have been many efforts to accelerate logic simulation which essentially fall into three categories:

- Variation of simulation techniques and introduction of special data structures
- Parallelization of simulation software
- Migration of simulation processes into hardware

These categories are closely related and don't exclude each other. With respect to the first point, time-driven, event-driven and demand-driven simulation techniques are to be mentioned which have been combined with compiled and interpretive implementation methods [1, 2]. A further promising approach is based on the usage of Binary Decision Diagrams (BDDs) as representations of Boolean functions [3]. In this case, simulation speed up comes at the expense of a large need for memory. Examples of parallel event-driven and time-driven logic simulators can be found in [4] and [5], respectively. For these approaches, balancing of load and keeping the communication overhead as small as possible are key issues. The partial or complete realization of simulation processes in hardware making use of hardware accelerators and emulators [6] by far results in the highest speed gain. But in this case a loss of flexibility and observable modeling detail has to be noted. We have parallelized the sequential functional logic simulators *TEXSIM* and *MVLSIM* (IBM) with loosely-coupled processor systems as target hardware [7]. Our parallelization approach makes use of model inherent parallelism. Within a parallel simulation, parts of the original circuit model are distributed to cooperating simulator instances. Thereby, sophisticated model partitioning is a crucial point to reach significant simulation speed up. Corresponding partitioning problems are related to NP-hard combinatorial optimization problems. That's why for the partitioning of complex circuit models heuristics are essentially applied to provide suboptimal solutions. An introduction to the spectrum of partitioning problems in VLSI design is given in [8]. There exists a couple of partitioning tools such as *METIS* [9] and *JOSTLE* [10].
For our special application field we have developed a hierarchical partitioning strategy [11]. Model partitioning appears as BOTTOM UP clustering starting from basic building blocks called *cones*. A prepartitioning phase that is intended to reduce problem complexity leads to *super-cones*. These objects are taken to generate hypergraph structures. Their edges carry information with respect to overlapping and communication relations between model parts. In a second partitioning phase, algorithms can exploit these structures for partition valuation on the basis of a formal model of parallel logic simulation. In general, a partitioning process is concluded by iterative algorithms to

*supported by Deutsche Forschungsgemeinschaft (DFG)

improve partitions. Among other algorithms, we have developed *Parallel Evolutionary Algorithms* [12] for this purpose.
The raising number of algorithms applied in our partitioning project led to the need for efficiently organizing work within a partitioning environment. There were two application scenarios to consider: It should serve as a production environment in industrial processor design processes (IBM) as well as an experimental test bed for the development of partitioning algorithms in the academic area. The resulting *parallelMAP* environment is a realization of the client/server architecture *DRIVE (Distributed Runtime Environment)* [13]. It supports the construction and execution of partitioning processes on heterogeneous computing resources (for instance, workstation clusters in combination with parallel machines).
In this paper we will describe the possibilities *parallelMAP* offers for the modular construction of partitioning processes. We start with an outline of our model partitioning strategy in Section 2. Four phases of a typical partitioning process are presented. In Section 3 the potential of our modular partitioning approach for parallel and distributed execution is shown. Then, in Section 4 the main components of *parallelMAP* are introduced. Section 5 presents experimental results with respect to the runtime of basic modules of our partitioning processes. Section 6 comprises conclusions and our intentions for future work.

2 Hierarchical partitioning process

We work with processor models provided by IBM and have access to their internal structure through the IBM-specific Design Automation DataBase (DADB). First we will introduce some basic terms in context of the DADB. The data type for a processor model is called *proto*. It contains the functional description of the processor model and is generated by compiling DSL/1 or VHDL sources. A proto consists of lists of *boxes* and *nets*. Boxes can embody logical gates, latches, input-pins, output-pins or a combination of boxes. Nets represent connections between boxes. The complete list of models considered in this paper, their function and size is shown in Table 1.
Cones as special circuit areas play an important role in our model partitioning processes. We consider cones which are determined by output-pins or latches (so called head-boxes). A cone belonging to such a head-box b is defined as the set of all boxes of the proto from which a direct path to b exists without intermediate latches (see shaded area in Fig. 1).
Because of the high complexity of the partitioning problem, a hierarchical partitioning strategy has been chosen which allows the splitting of the problem into easier parts. For a complete partitioning process, starting from a proto as representation of the original circuit model up to the generation of parallel simulator input, we distinguish four consecutive phases: *prepartitioning*, *partitioning*, *partition improvement* and *simulation model generation*. Prepartitioning (I) concentrates cones resulting in a specified number of super-cones. This represents a step into a new hierarchy level. The partitioning algorithms (II) assign the previously generated super-cones to blocks. The use of super-cones instead of cones leads to a lower number of elements to assign and therewith to a lower complexity of the partitioning process. The improvement phase (III) conditionally changes the assignment of super-cones to blocks with the objective of increasing the expected speed of parallel simulation processes based on a corresponding model partition. Finally, the simulation models as input for the parallel simulator are generated in phase (IV). In the following we illustrate the mentioned phases in more detail. Algorithms of the prepartitioning phase (I) identify all head-boxes and generate a specified number of super-cones by uniformly distributing the head-boxes (each box representing ”its” cone). This is done without explicit consideration of the cone size and cone overlapping. The *STEP*-algorithm is an example for such an algorithm. As parameter it takes the number of super-cones to be built. This number typically ranges from 100 up to 1000 depending on the model size. Assuming the head-boxes numbered consecutively according to their position in the proto box list, the *STEP*-algorithm assigns the head-boxes $s * \frac{n_{headboxes}}{n_{supercones}}, \ldots, (s+1) * \frac{n_{headboxes}}{n_{supercones}} - 1$ to the super-cone s. This approach takes advantage from the structure of the proto data type defined in the DADB. The *STEP* prepartitioning can lead to super-cones with quite little overlapping effects.
For further partitioning, aspects of overlapping and communication between super-cones are represented by hypergraphs. Within phase (II), hypergraph generation (IIa) using a proto and a prepartition is done as

Name	Function	Boxes	Nets
CLKSTR6	Clock-Chip (Monet)	187120	274781
PICMOFP	Picasso Chip-set, IBM 390 architecture	235721	374837
PU_M5X	Processing Unit (Monet)	252982	355453
MBA98	Memory Bus Adapter (Monet)	512843	510986
ML100M0S	Monet Chip-set, IBM 390 architecture	2657165	4830437

Table 1. Name, function and size of the studied processor models

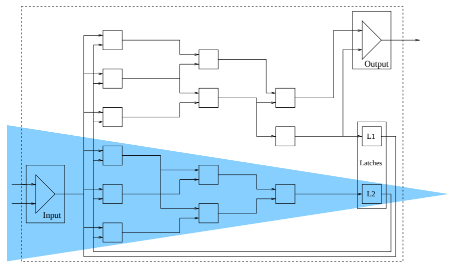

Figure 1. Cone belonging to latch L2

first step. A following partitioning algorithm (IIb) assigns super-cones to a specified number of blocks taking a hypergraph, prepartition and proto (or a subset of them) as input. To exemplify a partitioning algorithm from the second phase we sketch the *MOCC* (Minimum Overlap Cone Cluster) algorithm. We assume the number of blocks to be n:

1. The n super-cones with the largest number of boxes are chosen and assigned to the n blocks.
2. The remaining super-cones are sorted according to their size.
3. The block B with the least number of boxes is chosen.
4. The overlapping areas of B with all remaining super-cones are calculated and the super-cone that contributes most to overlapping is assigned to B.
5. The number of boxes for B is recalculated.
6. If there are remaining super-cones, the procedure is continued at 2.

The algorithms of phase (III) try to improve partitions by moving super-cones from one block to another. As an example we outline the parallel genetic algorithm (PGA). We consider one of a number of identical modules realizing a multiple subpopulation approach:

1. A partition is taken as input and a number of individuals (representing partitions) is generated to form a subpopulation.
2. Genetic operators modify individuals by exchanging small numbers of cones (variation).
3. Using a fitness function (related to estimated parallel simulation times) the quality of the modified individuals is calculated.
4. Some of the individuals are exchanged with other PGA modules involved in the partitioning of the same proto (migration).
5. The best individuals are selected to form the base of the next generation (selection).
6. As long as the predefined number of generations is not reached, the procedure is continued at 2.

After the third phase one final partition is available. Phase (IV) comprises "technical" algorithms which generate for each block the corresponding proto (IVa) and simulation model (IVb) starting from the final partition. So, this phase provides the model-related input for the parallel logic simulator.
Within our partitioning strategy, we allow the *competition of algorithms*. Having on the one side complex circuit models and on the other side a set of heuristics to handle them, in general one can not know in advance which algorithm is to be preferred. In this context, distributed execution of different algorithms together with a selection or merging mechanism for result determination is advantageous. To enable competition within our partitioning processes, some functionality has to be added to broadcast prepartitions and hypergraphs to involved nodes as well as a possibility to compare partitions and choose the one with the smallest expected parallel simulation time.
A complete overview of a partitioning process including simulator input generation and assuming competition of m algorithms in phases (IIb) and (III) is shown in Fig. 2.
A selection of implemented algorithms and a short description can be found in Table 2.

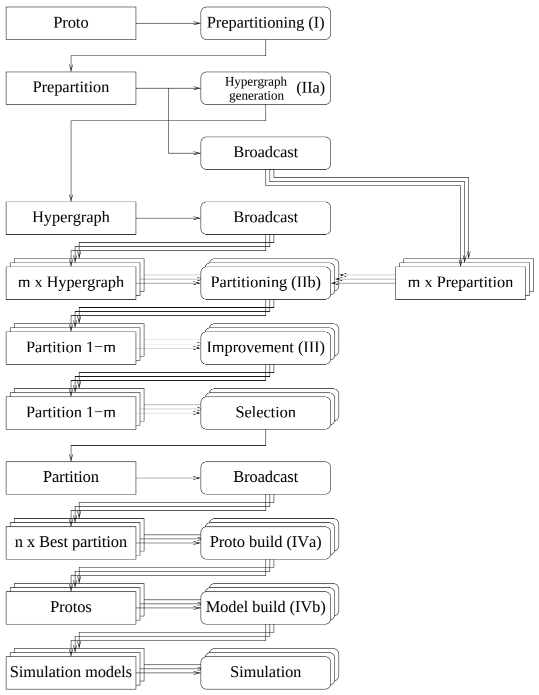

Figure 2. A complete partitioning process

3 Potential of parallel and distributed module execution

In the previous section we have described a sequence of phases forming typical partitioning processes in the context of our parallel logic simulators. Like simulation itself, these processes show a potential of reasonable parallel and distributed execution. One argument supporting this assertion is delivered by the competition of algorithms mentioned in the previous section. Furthermore, if we consider the generation of protos and simulation models as input for the parallel simulators in phase (IV), this can be done for each model part independent of the corresponding steps for other model parts. Then, with our PGA we have a building block of partitioning processes that already represents a parallel algorithm. Finally, taking the aspect of algorithm development into consideration, partitioning processes can be accompanied by a couple of analyzing processes and processes for the generation of intermediate structures which are well suited for distributed realization.
To describe execution dependencies between modules

we make use of *Series Parallel (SP) Graphs*. In Fig. 3 a) three fictitious partitioning processes (without input generation for parallel simulation) are represented. Process 1 describes a sequential partitioning process. In contrast, process 2 shows competition of algorithms with corresponding result selection and/or result merging modules. There are two competing prepartitioning algorithms and two competing sequences of partitioning and improvement algorithms. One of the improvement algorithms is assumed to be parallel (on this level of abstraction without information on a number of nodes to be exploited). Process 3 is built from the former two processes which can be executed in parallel. In the context given, SP graphs

Phase	Name	Function
I	STEP	Algorithm description can be found in Section 2
	SAWTOOTH	The list of head-boxes with their cones is separated by a sawtooth-like function, the rise is adjustable by a parameter
	RANDOM	The cones are arranged randomly
II	MOCC	Algorithm description can be found in Section 2
	nBCC	n Backward Cone Concentration – n overlapping super-cones are combined and assigned to the block with the lowest number of boxes
	STEP	Similar to the prepartitioning algorithm STEP the super-cones are assigned to blocks according to a given order
	RANDOM	The super-cones are assigned randomly to blocks
	SAWTOOTH	Similar to the prepartitioning algorithm SAWTOOTH the super-cones are separated by a sawtooth-like function
III	PGA	Algorithm description can be found in Section 2
	ITALG	Iterative improvement algorithm, searching for adjacent suboptimal partitions
	ADAPTED	Iterative improvement algorithm with problem specific (adapted) cost function
	MIXED	Combination of MINCUT and ADAPTED to achieve a faster solution than using ADAPTED and a more balanced one than using MINCUT
	MINCUT	Min-cut-algorithm for overlapping and communication hypergraph. Because of the two hypergraphs a weighted sum of the cut sizes is used to achieve a better partition
	TABU_ SEARCH	Best adjacent solution is chosen and to prevent loops already visited, solutions are blocked for a specified period

Table 2. Partitioning algorithms which can be used in the different phases

represent a starting point for scheduling processes resulting in a number of execution threads on a virtual set of nodes. For process 2, in Figure 3 b) a transformation result comprising three execution threads is shown. Dashed arrows represent communication relations between instances of parallel algorithms, continuous arrows describe the control flow within the execution threads. We have two parallel algorithms for selection/merging processes (concerning two and three execution threads, respectively). A broadcast operation with respect to the selected prepartition and the generated hypergraph has been inserted. Finally, there is a parallel improvement algorithm with instances appearing within two execution threads.
The scripting language of our partitioning environment *parallelMAP* allows to formulate execution threads as described above based on a set of dynamically linked libraries that provide the modules available for the construction of partitioning processes.

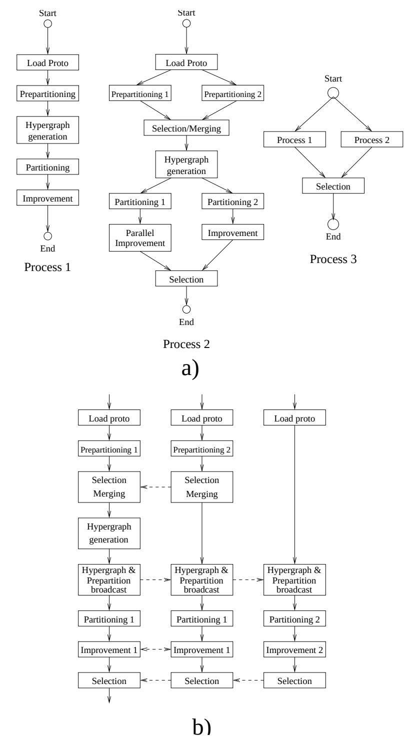

Figure 3. a) Modular structure of partitioning processes given in terms of SP graphs b) Three execution threads belonging to Process 2 with four parallel algorithms involved

4 Partitioning environment *parallelMAP*

This environment has been developed to support model partitioning in the industrial design process as well as algorithm design in the academic area. It is a realization of the distributed runtime environment *DRIVE* [13] that enables scientists and engineers to build, run and monitor parallel programs in an easy way. Programs are formulated using a scripting language that allows the combination of sequential and MPI-based parallel program modules. A *DRIVE* script specifies a set of execution threads such as shown in Figure 3 b). There are three types of basic *DRIVE* components called clients, servers and workers. They are completed by problem specific shared libraries and databases (`DADB` in the case of *parallelMAP*). The base of a *DRIVE* implementation is the server. Clients may connect to, and workers and processing machines are controlled by the server. The server is able to attend to several users simultaneously, each possibly handling a couple of scripts. It also realizes: script-based node allocation, starts of a set of worker instances as framework for parallel script execution, application coordination and the control of the data flow between clients and their corresponding set of worker instances.

The clients provide a comfortable graphical user interface and an editor allowing partially automated script generation. Beyond this, scripts can be started (sent to the server to be arranged for execution) and the output of the workers can be monitored using a socket connection for communication.

Worker instances are always related to an underlying script. From the programming point of view, a set of workers embodies a set of n processes belonging to a MPI-program under execution. Each of the workers has its own socket connection to the server for receiving a sequence of script-based requirements for function execution. Some of the functions are internal functions (implemented within the worker), but the majority of the functions is located in shared libraries. The internal functions are of general importance (for instance, concerning synchronization and communication aspects related to worker components). Functions in the shared libraries are application-specific. In the case of *parallelMAP* they represent algorithms which can be applied in the four phases of model partitioning processes we have described in Section 2. There are distinguished sequential and parallel functions, as well. The execution of the latter ones is connected with a set of related function calls within different workers. Communication between the corresponding function instances is based on MPI. For example, there exists a parallel function to realize *PGA* (see Section 2). Dynamic linking permits the modification of application-specific *parallelMAP* functionality even at runtime.

The general structure of a script consists of three parts: machine reservation, communicator initialization for MPI and the list of queues. Such a list of queues describes a set of possibly parallel execution threads.

5 Experimental experiences

In Section 3 we have outlined the potential of our modular construction approach for partitioning processes with respect to parallel and distributed realization. A variety of partitioning processes described by SP graphs of moderate size is thinkable. To support scheduling processes starting from such graphs, one needs information on the runtime of involved modules under different conditions. Related experimental results will be presented in the following. First we want to describe our experimental environment. As mentioned before we use different processor models (see Table 1).

5.1 Measurement environment

The runtime measurements have been done on an IBM RS/6000 Workstation Model 595 running AIX Version 4.3 equipped with a 135 MHz P2SC processor and 2GB of RAM. To measure the algorithms efficiently we have modified the *parallelMAP* worker's source code. The worker usually does not send runtime information via the server back to the clients. Therefore we have modified the wrapper function which starts functions stored in shared libraries. We obtain runtime values using MPI's `MPI_Wtime` function and send them to clients using the previously established socket connection. Script generation and script execution has been automated. For this purpose a client implementation in *Java* was modified. A *Java* class was implemented that generates new scripts from lists of protos, prepartitioning, partitioning and improvement algorithms by rearranging them. The generated scripts are sent to the *parallelMAP* server, the worker's output with the included runtime information is received and the runtime information is filtered out for later evaluation.

5.2 Results of the measurements

Loading phase The partitioning starts with the loading of the proto. The loading time depends on the size of the proto, i.e. the number of its boxes and nets, as well as the underlying file-systems like *NFS*, *AFS* or a local one. For the protos considered, the loading times range from 4 up to 57 seconds and depend linearly on the number of boxes and nets in the proto.

Prepartitioning phase (I) The runtime of prepartitioning depends on the size of the proto and the prepartitioning algorithm used. The number of generated super-cones has no influence on the runtime of the prepartitioning. This is obvious after looking at the details of the prepartitioning algorithms (see Section 2). The runtimes for the *RANDOM* algorithm are 10 to 20% larger than for the *STEP* or *SAWTOOTH* algorithms. That may be caused by the extra use of the `rand` function for calculating the random numbers.

Hypergraph building (IIa) The next step in partitioning is the generation of the hypergraph. Corresponding runtime values are influenced by the proto used and the previously executed prepartitioning algorithm. Results are shown in Fig. 4. Values for the largest proto (ML100M0S) based on the *SAWTOOTH* algorithm using 500 super-cones and the *RANDOM* algorithm could not be measured because of memory overflow. The shorter the runtime for building hypergraphs, the smaller the generated hypergraphs, and, the smaller the overlapping area of the super-cones and the communication amount between them. For the *STEP* algorithm, the runtime is almost linear with respect to the number of super-cones generated in prepartitioning. As shown in Fig. 4, the *STEP* algorithm leads to the smallest runtime for the hypergraph generation. All the following measurements assume *STEP* as prepartitioning algorithm used.

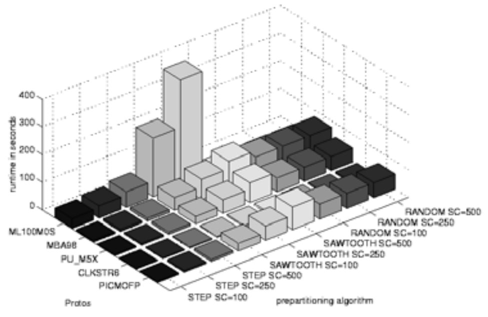

Figure 4. Runtime of the hypergraph generation (IIa) depending on the prepartition algorithm, the number of super-cones (SC) and the proto used

Partitioning phase (IIb) In this phase the prepartition and hypergraph are used as well as the proto to generate a partition. *SAWTOOTH*, *RANDOM* and *STEP* (see Table 2) show the smallest runtime, but they mainly produce partitions connected with very long simulation runtimes.
In general, *nBCC* and *MOCC* produce partitions which are better suited for parallel simulation. Runtime values of the *nBCC* and *MOCC* algorithm considering different standard parameter settings are shown in Fig. 5 with respect to the `PICMOFP` proto.

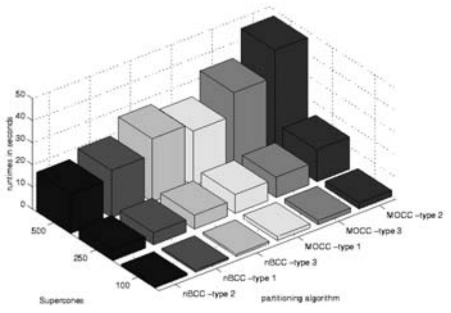

Figure 5. Runtime of $nBCC$ and $MOCC$ (IIb) with different parameter settings for the PICMOFP proto

Improvement phase (III) We consider six different improvement algorithms using the PU_M5X proto. Five of them are iterative improvement algorithms and the sixth is the earlier mentioned parallel genetic algorithm *PGA* (handling 100 or 200 generations). As algorithms of phase (IIb) *nBCC* and *MOCC* are assumed. The results are shown in Fig. 6. Inconsistent results for the iterative improvement algorithms are caused by their truncation condition. The algorithms stop, if no better adjacent solution is reachable. In contrast the runtime of the *PGA* is affected by the proto-size, the number of super-cones and a given number of generations only.

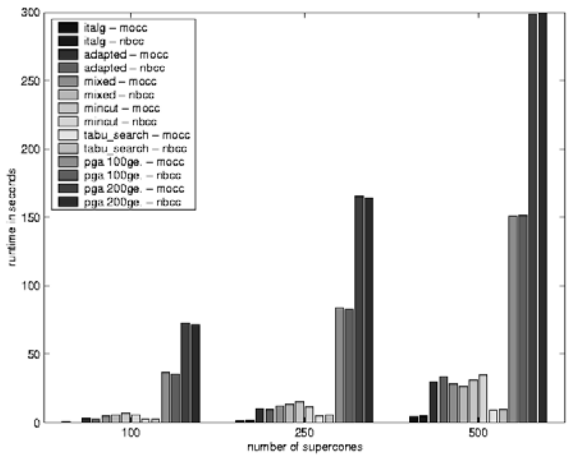

Figure 6. Runtime of different algorithms in phase (III) using $MOCC$ and $nBCC$ as partitioning algorithms in phase (IIb) with respect to the PU_M5X proto

Combination of phases (I - III) Finally, in Fig. 7 we show the share of different phases in the overall-runtime of partitioning processes without consideration of simulator input generation. Results are given with respect to the `PU_M5X` proto. The 90 processes are represented using a top-down division by:

- Partitioning algorithm (*nBCC* or *MOCC*)
- Number of super-cones (100, 250 or 500)
- Subtype of the partitioning algorithm (1, 2 or 3)
- Five iterative improvement algorithms

6 Conclusion and future work

We have presented a modular approach to the construction of model partitioning processes for parallel logic simulation of processor structures in VLSI design. It has been realized within the partitioning environment *parallelMAP* that provides a platform for efficient work of both design engineers and scientists developing partitioning algorithms.
We have shown the potential of our special partitioning processes for parallel and distributed execution. For the representation of partitioning scenarios SP graphs have been used. They provide a basis for scheduling

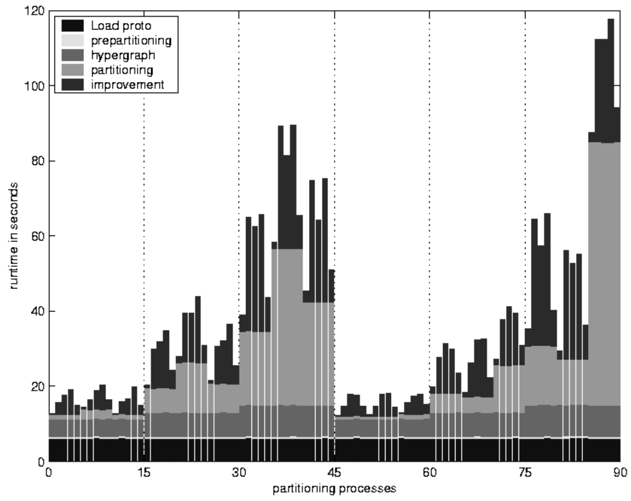

Figure 7. Runtime of whole partitioning processes without simulator input generation using different numbers of super-cones, partitioning and improvement algorithms on the PU_M5X proto

processes leading to descriptions of sets of execution threads. Within *parallelMAP*, such descriptions are given using the *DRIVE* scripting language. To support scheduling, we have provided runtime information with respect to our basic modules for the construction of partitioning processes.

In future work we will investigate relations between the runtime of partitioning processes and the quality of resulting model partitions in terms of expected parallel simulation times. Furthermore, we will include load information concerning the target system for the execution of model partitioning into the process of generating corresponding *parallelMAP* scripts.

References

[1] J. B. Gosling, *Simulation in the Design of Digital Electronic Systems*, Cambridge University Press, 1993.

[2] C. C. Charlton, D. Jackson, and P. H. Leng, “Modelling and simulation of digital logic: An alternative approach,” in *Proc. of IASTED Int. Symp. Identification, Modelling and Simulation*, 1987, pp. 86–90.

[3] C. Scholl, R. Drechsler, and B. Becker, “Functional simulation using binary decision diagrams,” in *Proc. of the IEEE/ACM Int. Conf. on Computer-Aided Design (ICCAD’97)*, 1997, pp. 8–12.

[4] R. Schlagenhaft, M. Ruhwandl, C. Sporrer, and H. Bauer, “Dynamic load balancing of a multi-cluster simulator on a network of workstations,” in *Proc. of the 9th Workshop on Parallel and Distributed Simulation (PADS’95)*, 1995, pp. 175–180.

[5] K. Hering, J. Löser, and J. Markwardt, “dlbSIM - a parallel functional logic simulator allowing dynamic load balancing,” in *Proc. of the Conf. on Design, Automation and Test in Europe (DATE’01)*, 2001, pp. 472–478.

[6] A. Dieckmann, “HW-SW-coverification with emulation, co-simulation and FPGA-based prototyping,” in *Designer’s Forum, Conf. on Design, Automation and Test in Europe (DATE’01)*, 2001, pp. 98–101.

[7] D. Döhler, K. Hering, and W. G. Spruth, “Cycle-based simulation on loosely-coupled systems,” in *Proc. of the 11th Annual IEEE Int. ASIC Conf. (ASIC’98)*, M. E. Schrader, R. Sridhar, T. Buechner, and P. P. K. Lee, Eds., 1998, pp. 301–305.

[8] F. M. Johannes, “Partitioning of VLSI circuits and systems,” in *Proc. of the 33rd Design Automation Conf. (DAC’96)*, 1996, pp. 83–87.

[9] G. Karypis and V. Kumar, “Multilevel algorithms for multi-constrained graph partitioning,” Tech. Rep. TR 98-019, Department of Computer Science, University of Minnesota, 1998.

[10] C. Walshaw and M. Cross, “Parallel optimisation algorithms for multilevel mesh partitioning,” Tech. Rep. 99/IM/44 (Mathematics Research Report), University of Greenwich, Centre for Numerical Modelling and Process Analysis, 1999.

[11] K. Hering, R. Haupt, and Th. Villmann, “Hierarchical strategy of model partitioning for VLSI-design using an improved mixture of experts approach,” in *Proc. of the 10th Workshop on Parallel and Distributed Simulation (PADS’96)*, 1996, pp. 106–113.

[12] H. Schulze, R. Haupt, and K. Hering, “Experiments in parallel evolutionary partitioning,” in *Proc. of Int. Conf. ParCo99, Parallel Computing - Fundamentals & Applications.* 2000, pp. 383–390, Imperial College Press.

[13] K. Hering, H. Hennings, and R. Haupt, “DRIVE: A distributed environment supporting combination of sequential and parallel modules,” in *Proc. of the IASTED Int. Conf. on Parallel and Distributed Computing and Systems (PDCS’99)*, 1999, pp. 637–644.