Procedure Cloning: A Transformation for Improved System-Level Functional Partitioning

Frank Vahid
Department of Computer Science
University of California, Riverside, CA 92521
vahid@cs.ucr.edu, www.cs.ucr.edu

Abstract

Functional partitioning assigns the functions of a system's program-like specification among system components, such as standard-software and custom-hardware processors. We introduce a new transformation, called procedure cloning, that significantly improves functional partitioning results. The transformation creates a clone of a procedure for sole use by a particular procedure caller, so the clone can be assigned to the caller's processor, which in turn improves performance through reduced communication. We define several cloning heuristics that seek to clone the minimum number of procedures, a goal necessary to obtain the best improvements. We highlight experiments comparing our cloning heuristics and showing partition improvements with cloning.

1 Introduction

Functional partitioning is an increasingly important task for system design environments. In such partitioning, a behavioral specification's functions are assigned to system components, including software processors, custom hardware processors, and memories. Such partitioning must satisfy constraints, like package size and pin limits, while minimizing other metrics, like execution time or power. Recent research has shown benefits of lower cost and better performance when functionally partitioning among hardware and software [1, 2, 3, 4, 5, 6, 7, 8]. Other research focused on functionally rather than structurally partitioning among hardware packages [9, 10, 11, 12, 13, 14, 15] with numerous benefits, like fewer packages, improved performance, faster synthesis, and easier debugging [16]. To gain these benefits, good functional partitioning approaches are needed.

Functional partitioning approaches usually start with a program-like specification of system functionality. Developers of such specifications, like their software developer counterparts, must create modular, readable, and reusable programs. These issues lead to extensive use of procedures, many called from multiple locations. To our knowledge, such multiply-called procedures are handled in earlier approaches either by: (1) Treating each procedure as a single computation, so one instance of a procedure (or of its blocks or statements) is partitioned among components, or (2) Treating each procedure *call* as a computation, so multiple instances are partitioned.

The second approach exposes the largest solution space by essentially creating a dataflow graph, where a distinct graph node for each call shows the different data dependencies per call, similar to using distinct addition nodes for each addition operation for behavioral synthesis. However, a larger space represents a harder partitioning problem. Our experiments show that the number of nodes can increase by almost an order of magnitude (and thus the solution space by an even greater factor), leading to inferior solutions.

The first approach also has a drawback, but it can be solved by cloning. The drawback is that a single procedure instance may be called from procedures on other components, requiring inter-component communication, but in some cases copying the procedure for each call would eliminate this communication. For example, in Figure 1(a), procedure *Main* calls *Slow* 128 times and *Fast* 16 times, which both call *Util* 256 times and *Resource* 64 times. There is inter-component communication between *Slow* and *Resource* and between *Fast* and *Util*. Assume *Resource* requires 200 cycles in software and only 20 in hardware, *Util* requires 20 cycles in both, and each inter-component communication requires 2 cycles. The time for communicating with and executing *Resource* is thus: $(128 * 64 * (2 + 20)) + (16 * 64 * 20) = 200,704$. Likewise, the time for *Util* is: $(128 * 256 * 20) + (16 * 256 * (2 + 20)) = 745,472$ cycles. Moving *Util* to hardware would increase its time to $802,816$ due to more communication. Moving *Resource* to software, while reducing communication, increases its time to $1,845,248$ due to slower execution in software.

The solution is to copy, or clone, certain procedures. Copying *Util* for use by *Fast* in hardware, as in Figure 1(b), reduces *Util*'s time to: $(128 * 256 * 20) + (16 * 256 * 20) = 737,280$. In contrast, copying *Resource* increases time, since the eliminated communication is outweighed by slower execution in software. Thus, we have aimed to clone procedures only when beneficial. Each procedure is initially represented as one node, and then a cloning transformation, guided by a heuristic, clones a subset of procedures, analogous to duplicating gates during circuit partitioning [17]. Cloning heuristics are needed to clone just the right procedures and hence improve partition results, without cloning too many procedures and hence worsening partition results.

In this paper, we define the clone and unclone transformations, describe how estimation techniques must be modified to account for clones, introduce several heuristics for performing cloning before, during and after functional partitioning, and highlight experiments demonstrating cloning's benefits and comparing our heuristics.

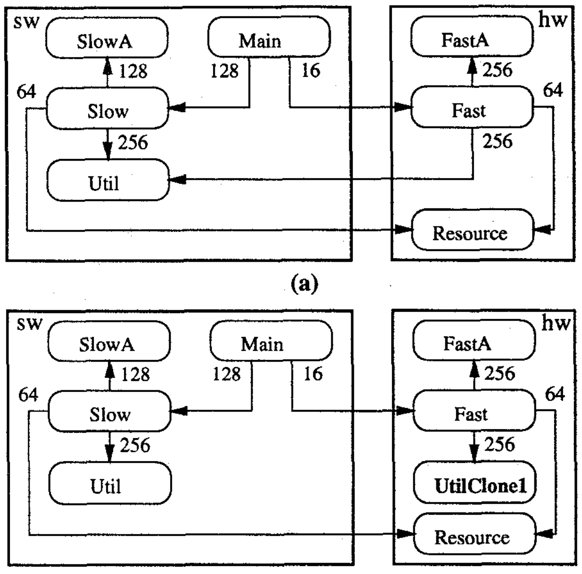


Fig. 1: Cloning example: (a) original partition, (b) reduced communication after cloning.

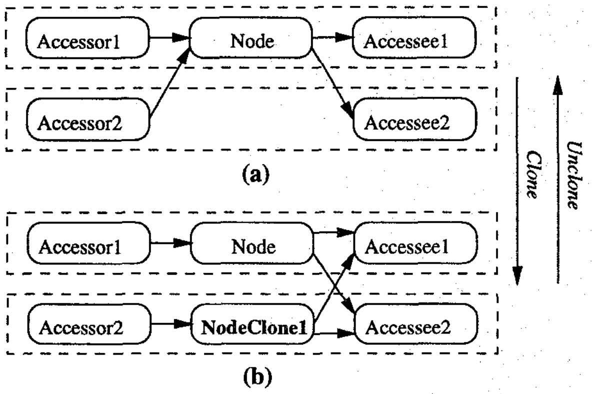


Fig. 2: Cloning an AG node.

2 Clone/unclone transforms

2.1 Internal representation

We convert a specification to an Access-Graph (AG) representation [18, 19], suitable for many system design problems like partitioning and cloning. An AG node represents a behavior (procedure or process) or variable. An AG directed edge represents a behavior accessing another behavior (a procedure call) or variable (a read or write). The edge direction indicates the accessor and accessee, but *not* the direction of the flow of data, which can flow in either or both directions. Each node is annotated with internal computation times (execution excluding communication and accessed object times) and sizes for every implementation type (e.g., 8051 microcontroller or XC4000 FPGA). Each edge is annotated with its access frequency and the number of bits transferred per access. All annotations can be minimums, averages, or maximums. Developed equations quickly compute size, I/O, and execution times (including communication) from the annotations for any partition [18].

2.2 Cloning

We introduce the cloning transformation through a simple example. Consider the AG of Figure 2(a). *Node* has two accessors, *Accessor*1 and *Accessor*2. Cloning *Node* for *Accessor*2 results in the AG of Figure 2(b). *Accessor*2 now accesses its own copy *NodeClone*1, and no longer accesses *Node*. Also, *NodeClone*1 accesses the same nodes (*Accessee*1 and *Accessee*2) that *Node* accesses. Because cloning is intended to allow an accessor to have a copy of the node on its own part, *NodeClone*1 has been created on *Accessor*2's part. In this example, we would probably need to further clone *Accessee*1 and *Accessee*2 for *NodeClone*1, in order to obtain a reduction in communication time and I/O.

More formally, the clone transformation, as applied to an AG, can be defined as follows:

- Input: (1) an AG, (2) a **base** n, which is the AG node to clone having $fanin > 1$, and (3) an **accessor** a, which is an AG node that accesses n via an edge e.
- Output: An AG with new node n_{clone}, which is a copy of n including copies of outgoing edges, such that e connects a to n_{clone}, $n_{clone}.fanin = 1$, and $n_{clone}.part = a.part$.

Note that node cloning is only defined relative to a node's accessor; we must specify the *particular accessor* for which a node copy will be made. Also, if n originally had a fanin of 1, cloning need not be performed because a is already the sole accessor of n.

Also note that **cloning is not inlining**. In inlining, a procedure call is replaced by the procedure's contents. Inlining, like cloning, has the effect of copying a procedure for sole use by the accessor, but it has the added undesirable effect of choosing the procedure's implementation. In particular, a procedure can be implemented either as inlined, as a control subroutine, a custom processor, or a datapath functional unit; inlining before synthesis eliminates the latter three choices for a synthesis tool. Inlining can be very expensive when there are multiple calls to a procedure and/or there is a deep calling hierarchy; hardware or software sizes can grow prohibitively. Cloning, in contrast, keeps the procedures as procedures, so that subsequent synthesis tools can still choose the other three implementation options, usually far more efficient than the inlining option.

We currently only clone nodes representing procedures, not variables, though both node types can have fanin > 1. Cloning a variable node would prevent its accessors from communicating through stored data. Similarly, we do not clone procedures having static local variables, which essentially serve to share data.

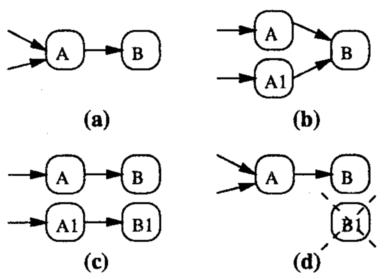


Fig. 3: Sequence leading to a dead node.

2.3 Uncloning

Uncloning will be required by our heuristics. Uncloning is the inverse transformation of cloning, but in some cases requires more work than undoing the changes of the clone transformation, as we shall see.

Consider the AG of Figure 2(b). *Node* has a clone *NodeClone*1, which was created for *Accessor*2. Uncloning *NodeClone*1 to *Node* results in the AG of Figure 2(a). *NodeClone*1 is gone along with its outgoing edges, and *Accessor*2 now accesses *Node* instead.

More formally, the unclone transformation, as applied to an AG, can be defined as follows:

- Input: (1) an AG, (2) a clone n_{clone}, which is the AG node to be uncloned, and (3) an identical node n_{ident}.
- Output: An AG in which n_{clone}'s incoming edges point to n_{ident}, and which does not include n_{clone}, n_{clone}'s outgoing edges, or dead nodes.

A dead node is one that had incoming edges before the unclone, but has none after the unclone. For example, Figure 3 shows a sequence of clone and unclone transforms leading to a dead node. Starting with Figure 3(a), we clone A for one accessor, resulting in $A1$ of Figure 3(b). This clone increases B's fanin to 2, making B a candidate for cloning. Cloning B for $A1$ yields $B1$ in Figure 3(c). Now, if we unclone $A1$ to A, as in Figure 3(d), $B1$ would be dead; uncloning must delete such dead nodes. Deleting a dead node may yield more dead nodes, which must also be deleted.

An identical node n_{ident} is a node with the same base as n_{clone}. Usually, n_{ident} is the base n itself from which n_{clone} was created, but it could instead be a fellow clone of n. We use the terminology of uncloning a node *to* another node, i.e., we unclone n_{clone} to n_{ident}.

3 Estimation modifications

We must account for clones when estimating performance, size, and I/O during partitioning, so that estimates don't become inaccurate. This means realizing that clones on a single part will always be uncloned to one node after partitioning, since all same-part accessors can use that one node without adding inter-part communication. For size estimation techniques where each node is assigned a weight and a part's size is computed as the sum of its nodes' weights, we maintain a list of bases on a part along with a count field indicating the number of nodes with that base. When a node is added to a part, that base's count is incremented; when deleted, the count is decremented. The node's size is added to the part's size only when the count changes from 0 to 1, and is subtracted only when count changes from 1 to 0. (More complex size estimation techniques that consider hardware sharing [20] can be similarly modified).

I/O estimation must be modified similarly. I/O is the number of input/output pins required on a part. For a part, all edges crossing the part's boundary and pointing to same-base nodes should only be counted once, since those nodes will be merged and hence the edges can share the same I/O pins.

Execution-time estimation does not need modification. We compute a node's execution-time for a given partition by adding the node's internal computation time (ict) and its communication time. Communication time is the time spent transferring data to/from accessed nodes, plus the execution-time of those nodes. We focus on partitioning a single large process, or multiple processes that do not contend for processing resources, so slow-downs due to resource contention need not be computed; extensions for multiple contending processes remain as future work. Because a clone's annotations are identical to its base's annotations, we need not modify the execution-time estimation technique for clones. When a clone is moved to a different part, the technique will automatically use a different ict value and different data-transfer times when computing that clone's execution-time.

4 Cloning heuristics

Having defined the clone/unclone transformations and discussed estimation modifications for clones, we now discuss different heuristics for finding the best procedures to clone. We classify cloning heuristics into three categories, each of which shall be discussed:

1. *Pre-partition cloning*: clone a subset of procedures, and then partition the new AG.
2. *Post-partition cloning*: partition the original AG, clone some subset of procedures based on the partition, and then partition again.
3. *Integrated partitioning and cloning*: partition using an iterative improvement heuristic that not only moves nodes among parts, but also clones and unclones nodes.

4.1 Pre-partition cloning

In *pre-partition max-cloning*, we clone procedures until none has more than one accessor, as illustrated in Figure 4(b). Max-cloning provides the largest solution

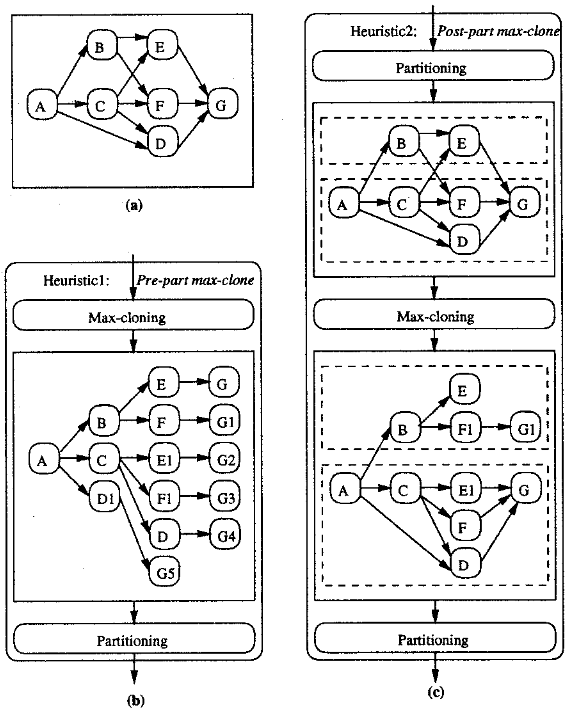


Fig. 4: Two cloning heuristics: (a) input AG, (b) pre-partition max-cloning, (c) post-partition max-cloning.

space to subsequent partitioning. Max-cloning is similar to creating a dataflow graph instead of an access-graph from the specification; since each procedure call involves distinct data, each call requires a distinct node. Max-cloning results in a large increase in the number of AG nodes, meaning that partitioning heuristics, which can not always find the optimal solution due to the NP-completeness of partitioning, may yield poor results and long run-times (as shown in Section 5).

In *pre-partition best-cloning*, we predict which nodes when cloned would best improve the cost after partitioning, much as clustering uses closeness metrics to predict which node groupings would yield the best final partition [20]. Because of the excellent results of the other heuristics, we have not yet investigated pre-partition best-cloning.

4.2 Post-partition cloning

In *post-partition max-cloning*, after initially partitioning, we clone every node (with fanin > 1, of course) *for every accessor on a different part* than the node itself, as long as the node also has an accessor on the same part. Contrast this with pre-partition max-cloning, in which we cloned a node for every accessor, not just those on different parts. Figure 4(c) illustrates post-partition max-cloning.

To understand why we require at least one same-part and one different-part accessor, consider the possibilities for a node with at least two accessors:

1. *All accessors are on the same part as the node*: There is no need to clone since all accessors already have same-part access to the node.
2. *All accessors are on different parts than the node*: If providing a clone on one of the accessor's parts would yield an improvement, then partitioning would have likely placed the node on that accessor's part. Since the node appeared on a different part, the node probably won't fit on the other parts, or the node's ict was lower on the current part but the accessor could not be moved there.
3. *At least one accessor is on the same part and another is on a different part*: The same-part accessor could have prevented the node from appearing on another accessor's part. Cloning will reduce communication and I/O; whether this reduction outweighs the increase in the accessor's part size and the possible reduction in the node's ict will be seen only after repartitioning.

In *post-partition best-cloning*, after initially partitioning, we predict which nodes when cloned would best improve the cost after repartitioning. One approach is to clone those nodes for accessors that immediately improve the partition's cost. However, our experiments found that a single clone rarely reduced cost after partitioning, so future work might focus on looking instead for sequences of clones yielding improvement.

4.3 Integrated partitioning and cloning

Iterative-improvement partitioning heuristics make thousands of changes, moving nodes among parts, using a control strategy overcoming local cost minima without making excessive moves. Modifying the definition of a "change" from just a node move to either a move, clone, or unclone, and thus integrating partitioning and cloning, represents the third cloning category.

The simulated annealing heuristic is a popular heuristic for which such a modification is straightforward. The modification replaces a function *RandomMove*, called in the heuristic's inner loop, by function *RandomChange*. The function has three parameters in addition to the partition itself, representing the probabilities of performing a move, clone, and unclone change, respectively. A move consists of moving a random node to a random destination part. A clone consists of applying the clone transformation of Section 2 to a random node with fanin > 1 and a random accessor of that node. An unclone consists of uncloning a random clone node to its base.

4.4 Max-uncloning

After any of the cloning heuristics, the resulting partition may have multiple same-based nodes (clones) on a single part. These nodes can be uncloned to a single node, shared by the accessors. We define a *post-partition max-unclone* transformation that unclones all same-base nodes on a single part through repeated application of the unclone transformation.

	None				Premax-u				Postmax-u				Integ-u			
Example	HwSize	HwIO	SwSize	Exec	HwSize	HwIO	SwSize	Exec	HwSize	HwIO	SwSize	Exec	HwSize	HwIO	SwSize	Exec
ans	2614	80	520	700	2849	76	483	425	2396	80	507	564	2533	80	478	398
ether	3753	74	1658	3767	10658	159	1019	2248	1238	80	2018	3237	2795	80	1807	3236
fuzzy	16405	180	15500	9530	16405	180	15500	9530	16405	180	15500	9530	16405	180	15500	9530
itv	12202	186	6341	13237	12402	198	6330	12853	12503	203	6293	11637	12565	197	6270	12057
mwt	4800	86	478	2266	4976	73	747	748	4954	99	344	760	4976	118	739	760
Avg % change	0.0%	0.0%	0.0%	0.0%	39.7%	20.2%	2.1%	-29.9%	-13.9%	6.5%	-1.9%	-22.4%	-4.4%	10.2%	10.9%	-26.5%

(a)

Example	None	None2	Premax	Postmax	Integ
ans	79	289	86	170	96
ether	75	258	99	180	109
fuzzy	66	233	66	149	61
itv	119	469	128	280	124
mwt	50	201	67	115	59

(b)

Example	None	Premax	Premax-u	Postmax	Postmax-u	Integ	Integ-u
ans	45	52	46	46	46	46	46
ether	123	142	125	126	125	140	124
fuzzy	70	70	70	70	70	70	70
itv	85	85	85	85	85	85	85
mwt	30	55	35	35	30	33	33

(c)

Fig. 5: Cloning heuristic comparison: (a) costs, (b) runtimes, (c) number of nodes.

5 Experiments

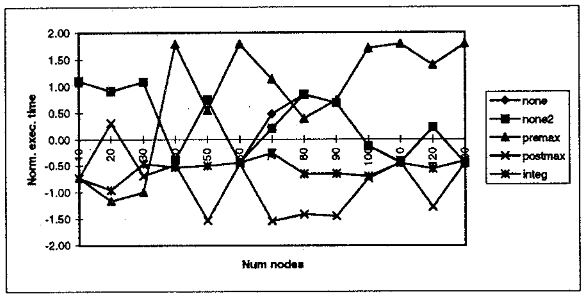
We conducted experiments comparing our heuristics and showing improvements gained by cloning. Examples included a telephone answering machine (*ans*), an Ethernet coprocessor (*ether*), a fuzzy-logic controller (*fuzzy*), an interactive TV processor (*itv*), and a microwave transmitter controller (*mwt*). We automatically annotated each example's AG using UC Irvine's SpecSyn estimators [20], and then partitioned among hardware and software (an 8086 processor and a Xilinx XC4000-series FPGA). Each example had at least one execution-time constraint, usually on a root node, thus the time included communication and times for many accessed nodes. We used a cost function with three terms: the total execution time of constrained nodes, the FPGA I/O constraint violation, and the FPGA size constraint violation; the processor's software size was not constrained. The FPGA constraints were weighed heavily to ensure they would not be violated. Such a formulation seeks to find the best software speedup possible using the FPGA as a coprocessor.

Figure 5(a) summarizes results. *None* represents partitioning without cloning, using simulated annealing. A four-times longer cooling schedule (*None2*) gave identical results. *Premax-u*, *Postmax-u* and *Integ-u* are pre-partition max-cloning, post-partition max-cloning, and integrated partitioning and cloning, followed by max-uncloning, as indicated by the *-u*. *Integ-u* used simulated annealing with clone and unclone probabilities of 0.05 each and hence a move probability of 0.9. We tried smaller move probabilities, but results were inferior. *HwSize*, *HwIO* and *SwSize* represent the number of hardware gates, hardware input/output pins, and software bytes after partitioning, respectively. *Exec* is the total execution time of all constrained functions in the example.

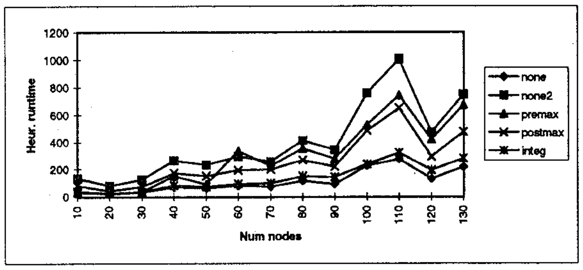
Cloning heuristics greatly reduced execution time, with 20% to 30% reductions. *Premax* increased average hardware size by 40%, while *Postmax* and *Integ* actually decreased average hardware size with only a minor software size increase or even a decrease. Such a decrease of both hardware and software size is possible because a node's hardware and software sizes are somewhat distinct, and hence swapping two nodes between hardware and software parts could actually decrease the size of both parts. A good cloning heuristic enables the partitioning heuristic to find such decreases. Figure 5(b) and (c) display run-times for each heuristic measured in seconds on a 166MHz Pentium, and the number of nodes remaining for each heuristic before and after max-uncloning the final partition.

To better understand heuristic performance as a function of problem size, we applied the heuristics to 13 highly-procedural generated examples ranging in size from 10 to 130 nodes in increments of 10, using generation techniques described in [19]. Figure 6(a) demonstrates the final normalized execution times. *Postmax* was usually the best, followed by *Integ*. *Premax* was best for tiny examples, but as the problem size grew, it quickly became the worst. *None* and *None2* yielded identical costs, demonstrating that the cost reductions couldn't be obtained through better partitioning alone. Figure 6(b) shows heuristic runtimes. *Integ* was only slightly slower than *None*. *Postmax* required about double the time of *Integ*, since partitioning is applied twice. *Premax* is even slower, even though it only applies partitioning once. Figure 7 shows the number of nodes before and after (indicated as *-u*) the final max-uncloning step. *Premax* yields large numbers of nodes; for example, 60 nodes grew to 572 nodes after max-cloning. Numbers for examples 100 through 130 were off the chart so are not displayed; they were 810, 922, 741, and 998, respectively. These large numbers of nodes explain *Premax*'s poor results and long runtimes in Figure 6. Even after max-uncloning, *Premax* yields many nodes, resulting in more hardware and software. *Integ-u* yielded the fewest nodes after max-uncloning, followed closely by *Postmax*.

In summary, both the *Postmax* and *Integ* cloning heuristics yielded excellent cost improvements over regular partitioning. *Integ* was faster, but this might not be true for other improvement heuristics.



(a)



(b)

Fig. 6: Generated examples: (a) cost, (b) runtime.

6 Conclusions

We demonstrated that significant improvements can be gained by combining procedure cloning with functional partitioning. We showed that post-partition max-cloning and integrated partition/cloning are both good cloning heuristics. We also showed the inferiority of approaches that expose a bigger solution space by creating a dataflow graph (pre-partition max-cloning). The success of cloning makes it a very useful transformation in any system-level functional partitioning tool.

References

[1] R. Gupta and G. DeMicheli, "Hardware-software cosynthesis for digital systems," in *IEEE Design & Test of Computers*, pp. 29–41, October 1993.

[2] R. Ernst, J. Henkel, and T. Benner, "Hardware-software cosynthesis for microcontrollers," in *IEEE Design & Test of Computers*, pp. 64–75, December 1994.

[3] S. Antoniazzi, A. Balboni, W. Fornaciari, and D. Sciuto, "A methodology for control-dominated systems codesign," in *Int. Workshop on Hardware-Software Co-Design*, pp. 2–9, 1994.

[4] J. Adams and D. Thomas, "The design of mixed hardware/software systems," in *Proc. of the Design Automation Conference*, 1996.

[5] X. Xiong, E. Barros, and W. Rosentiel, "A method for partitioning UNITY language in hardware and software," in *Proc. of the European Design Automation Conference (EuroDAC)*, 1994.

[6] P. Eles, Z. Peng, and A. Doboli, "VHDL system-level specification and partitioning in a hardware/software co-synthesis environment," in *Int. Workshop on Hardware-Software Co-Design*, pp. 49–55, 1992.

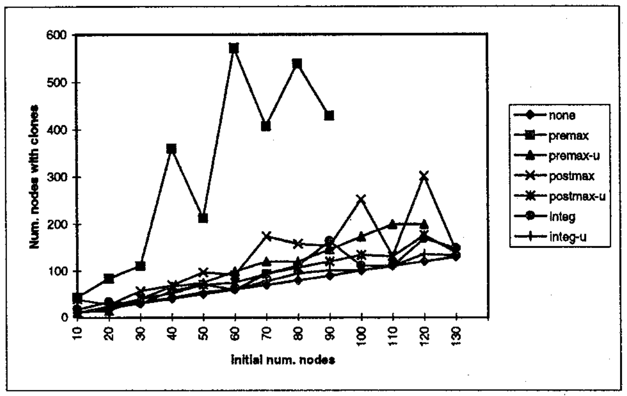


Fig. 7: Generated examples: numbers of nodes.

[7] P. Knudsen and J. Madsen, "PACE: A dynamic programming algorithm for hardware/software partitioning," in *Int. Workshop on Hardware-Software Co-Design*, pp. 85–92, 1996.

[8] A. Kalavade and E. Lee, "A global criticality/local phase driven algorithm for the constrained hardware/software partitioning problem," in *Int. Workshop on Hardware-Software Co-Design*, pp. 42–48, 1994.

[9] E. Lagnese and D. Thomas, "Architectural partitioning for system level synthesis of integrated circuits," *IEEE Transactions on Computer-Aided Design*, vol. 10, pp. 847–860, July 1991.

[10] R. Gupta and G. DeMicheli, "Partitioning of functional models of synchronous digital systems," in *Proc. of the Int. Conference on Computer-Aided Design*, pp. 216–219, 1990.

[11] K. Kucukcakar and A. Parker, "CHOP: A constraint-driven system-level partitioner," in *Proc. of the Design Automation Conference*, pp. 514–519, 1991.

[12] Y. Chen, Y. Hsu, and C. King, "MULTIPAR: Behavioral partition for synthesizing multiprocessor architectures," *IEEE Transactions on VLSI Systems*, vol. 2, pp. 21–32, March 1994.

[13] C. Gebotys, "An optimization approach to the synthesis of multichip architectures," *IEEE Transactions on VLSI Systems*, vol. 2, no. 1, pp. 11–20, 1994.

[14] Z. Peng and K. Kuchcinski, "An algorithm for partitioning of application specific systems," in *Proc. of the European Conference on Design Automation (EDAC)*, pp. 316–321, 1993.

[15] F. Vahid and D.Gajski, "Specification partitioning for system design," in *Proc. of the Design Automation Conference*, pp. 219–224, 1992.

[16] F. Vahid, T. Le, and Y. Hsu, "A comparison of functional and structural partitioning," in *Int. Symposium on System Synthesis*, 1996.

[17] L. Liu, M. Kuo, C. Cheng, and T. Hu, "A replication cut for two-way partitioning," *IEEE Transactions on Computer-Aided Design*, vol. 14, no. 5, pp. 623–630, 1995.

[18] F. Vahid and D. Gajski, "SLIF: A specification-level intermediate format for system design," in *Proc. of the European Design and Test Conference (EDTC)*, pp. 185–189, 1995.

[19] F. Vahid and T. Le, "Towards a model for hardware and software functional partitioning," in *Int. Workshop on Hardware-Software Co-Design*, pp. 116–123, 1996.

[20] D. Gajski, F. Vahid, S. Narayan, and J. Gong, *Specification and design of embedded systems.* New Jersey: Prentice Hall, 1994.