Clustering for improved system-level functional partitioning

Frank Vahid
Department of Computer Science
University of California, Riverside, CA 92521

Daniel D. Gajski
Department of Information and Computer Science
University of California, Irvine, CA 92717

Abstract

Partitioning of system functionality for implementation among multiple system components, such as among hardware and software components, is becoming an increasingly important topic. Various heuristics can accomplish such partitioning. We demonstrate that clustering can be used to merge pieces of functionality before applying other heuristics, resulting in reduced runtimes with little or no loss in quality, and often with improvements in quality. In addition, we show that clustering, when used for N-way partitioning, fills the gap between fast heuristics and highly-optimizing heuristics.

1 Introduction

A system's functional specification may have to be partitioned into several smaller parts for one of several reasons. First, we may not be able to implement the entire system on a single package. Second, we may wish to implement part of a primarily hardware system in software to reduce implementation costs, or part of a primarily software system in hardware to improve performance. Third, we can often improve the synthesis or manual design process by concentrating on smaller parts, yielding reduced size or improved performance, less synthesis runtime and memory use, and the possibility of concurrent synthesis/design. For example, we have found that for a 700-line encryption example in VHDL, functionally partitioning the specification into two parts reduced the total runtime of a particular behavioral synthesis tool from 1166 seconds down to 230 seconds, and reduced the resulting hardware size from 79,000 gates down to 65,000 gates.

For all of the above reasons for which we perform functional partitioning, the goals are similar. Given a functional specification and a set of target parts, the goal of functional partitioning is to assign functional objects to parts, as show in Figure 1, while satisfying constraints on global metrics like performance, part size, and pin constraints. To achieve this goal, we first *functionally decompose* the specification into some set of atomic *functional objects*, each representing a piece of the specification, such as a VHDL process, procedure or variable. We then partition these objects among parts, where a *part* represents a standard or custom processor, comprising a single chip or one of many blocks on a chip. A partition that doesn't meet constraints is said to have a non-zero *cost*. For example, Figure 2(a) provides a partial specification of a fuzzy-logic controller, and Figure 2(b) provides a decomposition of the specification into 12 functional objects, which in turn must be assigned to various parts.

Partitioning is a hard problem and is known to be NP-complete. In current practice, functional partitioning is done manually by a system designer. However, automated and interactive approaches are becoming important for several reasons. First, the trend towards implementation-independent specifications implies that the specification writer and the system designer may not be the same person, which means that the designer may not be able to partition the specification well without extensive examination. Second, recent efforts demonstrate the advantage of C-compiler partitioning approaches in which portions of time-consuming software are automatically compiled to a reconfigurable coprocessor [1]. Third, experienced designers may use automation to meet shorter design times.

Many automated partitioning heuristics have been developed. *Clustering* is a heuristic that successively merges objects based on local closeness metrics, such as the amount of data shared between two procedures, rather than based on global constrained metrics. Other heuristics include iterative-improvement strategies like group migration and simulated annealing, in which a given partition is iteratively modified and global metrics are re-evaluated in the hopes of finding a lower-cost partition. In our terminology, clustering is just one of many partitioning heuristics.

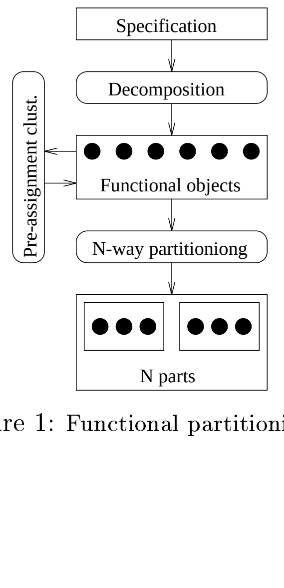

Figure 1: Functional partitioning.

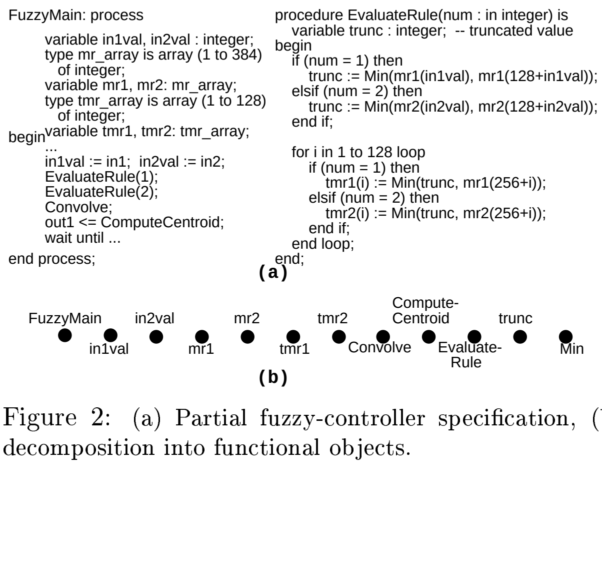

Figure 2: (a) Partial fuzzy-controller specification, (b) decomposition into functional objects.

No one heuristic yields the lowest cost partition in the minimum runtime for all examples. In general, choosing a heuristic involves making a tradeoff between partition cost and tool runtime. We have found that clustering provides a unique cost/runtime point in the tradeoff curve, filling a gap between existing heuristics. Perhaps more significantly, we have found that we can apply clustering before applying any heuristic, merging a few very close objects, in order to reduce subsequent heuristic runtime without incurring much higher partition cost.

Previous efforts in clustering during functional partitioning include clustering of logic operations [2], arithmetic operations [3, 4], and statements [5]. Other related functional partitioning efforts include arithmetic-operation level approaches [6, 7, 8, 9, 10], statement-sequence level approaches [11, 12], and state level approaches [13, 14]. Our work differs from the above works in two ways. First, we investigate various combinations of clustering heuristics with iterative improvement heuristics, rather than just one or the other. Second, we use coarser-grained functional objects, such as processes, procedures and variables, which in turn require new closeness metrics. Such granularity enables shorter runtimes and designer comprehension, and ensures that inter-part communication times don't dominate over computation times. The case for procedural-level granularity during system partitioning has been presented in [15, 16, 17, 18]. We should point out that the procedures being partitioned need not be the same procedures from the original specification; some can be eliminated through inlining, and others can be introduced through exlining [19].

The paper is organized as follows. In Section 2, we briefly describe the procedural-level closeness metrics that we defined for clustering. In Section 3, we demonstrate two roles of clustering during functional partitioning: pre-assignment clustering and N-way clustering. In Section 4, we provide results of a number of experiments. In Section 5, we provide conclusions and future work.

2 Closeness metrics for clustering

In this section, we informally describe the various closeness metrics that we have defined between functional objects. Formal definitions can be found in [20]. We treat a variable as a procedure, which is "called" whenever read or written. Also, we define the metrics between two groups of procedures, rather than just between two procedures. Finally, note that we also consider a concurrent process to be a procedure that simply repeats itself. To avoid confusion, we will refer to procedures, processes and variables as functional objects.

In order to define the closeness metrics, we first considered the three main classes of constrained metrics for an implementation: interconnect, performance and size. Intuitively, each closeness metric should focus on one of those three classes.

2.1 Interconnect

For interconnect, we defined a **connectivity** metric. This metric measures the estimated number of wires shared among two sets of functional objects if they were separated among two different parts. Grouping objects that would share wires should result in less interconnection between parts. This metric requires finding the data objects (e.g., parameters and global variables) accessed by both sets. In some cases, we must estimate the wires by encoding these data objects into bits. Then, the number of wires for a scalar variable is the number of bits, and the number of wires for an array variable is the number of address bits plus the word width. In other cases, the width is already known. This case arises when the *channel* between objects has been assigned to a particular *bus*, where the bus already has a specific number of wires.

In order to combine this metric with other metrics, we wish to normalize it to a number between 0 and 1. To do this, we can divide by the total number of wires accessed by either (not necessarily both) sets of objects. Thus, if two objects only transfer data between themselves, then they will have a closeness of 1. On the other hand, if either of those two objects also transfers data to another object, then the closeness will be less than 1. If they transfer no data between themselves, the closeness will be 0.

2.2 Performance metrics

There are two ways in which interaction among objects can affect performance. First, data may need to be communicated from one object to another – grouping such objects onto the same part should reduce lengthy inter-part communication times and hence improve performance. Second, two objects may be able execute concurrently – grouping such objects onto the same part may force sequential execution and hence hurt performance, and should thus be avoided when possible.

We defined a **communication** metric as the number of bits transferred between two sets of objects. This metric differs from the connectivity metric in that it measures the amount of data transferred between sets of objects, rather than the number of wires used to transfer that data. For example, if two objects communicate 16 bits of data 10 times over an 8 bit bus, then the communication metric would measure $10 \times 16 = 160$ bits, whereas the connectivity metric would measure only the 8 wires of the bus. The frequency of data transfer can be determined through profiling. We compute this metric by summing the bits transferred between the two sets of objects. We normalize by dividing by the total number of bits transferred between either set and any other object.

We noted, however, that some communications do not affect performance constraints. For example, an object A may transfer many parameters to another object, but if A is not constrained, and neither are any of A's calling objects, then the transfer is not particularly important. Thus, we defined a **constrained communication** metric, which differs from the previous metric in that it evaluates to 0 if the communication does not affect the performance of any constrained object. Such a metric might prove especially useful for hardware/software partitioning, since constrained objects are more likely to be assigned to hardware, with other objects being assigned to hardware.

Both of the communication metrics require profiling to obtain data transfer frequencies. At times, though, profiling information is not available. We thus developed an indirect performance metric called **common accessors**. When two sets of objects are accessed by many of the same objects, grouping those sets may improve the performance. This metric requires considering an object to be its own accessor, to encourage grouping an object with the objects that access it. To compute this metric, we count the number of objects that access both sets. We normalize by dividing by the total number of accessors of either set.

Finally, we defined a **sequential execution** as the number of pairs of objects that can execute sequentially, where a pair consists of one object from each set. We normalize by dividing by the number of possible pairs.

2.3 Size

There are two ways in which grouping objects can affect size. First, objects grouped onto the same custom part can often share hardware like a multiplier, thus reducing the overall hardware size. Second, partitioning objects among some number of identical parts usually requires that we maintain a balance among the required size of each group of objects.

We have defined a **shared hardware** metric to measure the hardware shared between two sets of objects $B1$ and $B2$. The shared hardware is computed as: $size(B1) + size(B2) - size(B1 + B2)$. We normalize by dividing by the minimum of $size(B1)$ and $size(B2)$, since the largest sharable amount occurs when one object can be implemented completely using the other object's hardware.

We have also defined a **balanced size** metric. When clustering objects for assignment among parts, we usually want clusters of balanced size. If we don't make an effort to balance size, the above metrics will cluster nearly all objects into one group. One way to encourage balanced sizes is to favor merging smaller objects over larger ones, to prevent any one group from getting too large. The metric is computed as the size of an implementation of both sets of objects. We normalize by dividing by the size of all objects in the specification.

Most of the above metrics make sense whether we are partitioning among hardware or software parts. The exception is the shared hardware metric; we have not found a software equivalent for this metric.

Note that the connectivity, sequential execution, and shared hardware metrics have a similar purpose to metrics previously defined for logic-operation clustering [2] and arithmetic-operation clustering [3, 4].

3 Roles of clustering

We have defined two distinct roles of clustering during system-level functional partitioning: pre-assignment clustering, and N-way clustering. Assignment is the task of assigning each functional object to a particular part. Pre-assignment clustering merges very close objects before any assignment considerations. On the other hand, N-way clustering assigns objects to parts. In pre-assignment clustering, we are not concerned with how many parts among which we will eventually be partitioning; we are only concerned with merging very close functional objects. For example, given 100 objects and 3 parts, pre-assignment clustering might reduce the number of objects down to 75 objects, whereas N-way clustering would assign every object to 1 of the 3 parts. The two techniques share the same mechanics, that of clustering objects based on closeness, but they have very different uses. We now discuss each clustering role in more detail.

3.1 Pre-assignment clustering

In pre-assignment clustering, we merge very close functional objects into a single new object before applying N-way partitioning heuristics, thus reducing runtime and possibly lowering costs. Very close objects usually exist in a specification, since specifications are typically written in a modular manner. Modularity implies that each procedure does not call every other procedure and access every global variable; instead, it implies that a procedure has a small number of neighboring procedures/variables that it accesses. Those objects are sometimes so close, meaning they only deal with one another and not other parts of the specification, that they should never be separated. Pre-assignment clustering combines such objects into a single (hierarchical) object, so that a subsequent partitioning heuristic can not separate them, but instead must treat them as a single object.

There are two advantages to pre-assignment clustering. First, fewer objects obtained by pre-assignment clustering usually means that subsequent partitioning heuristics will require less runtime and memory. Such reductions are very significant if we consider that we may apply partitioning heuristics tens or hundreds of times, once for each possible configuration of parts. For example, we may try partitioning the specification among a configuration of a slow processor and an FPGA, then of a fast processor and an FPGA, then two FPGA's, then two different FPGA's, and so on. We may even use a meta-algorithm that searches for the best configuration of parts by first allocating different configurations, partitioning among each configuration, and then evaluating the cost of each. Pre-assignment clustering is done just once, after which any number of partitioning heuristics may be applied.

Second, pre-assignment clustering may actually lead to better results. By merging objects into a single object, we are essentially pruning a portion of the solution space, i.e., those solutions where the objects are assigned to different parts. If we take care to only cluster objects that should never have been separated, then the pruned solutions would have represented inferior solutions. By removing such solutions from consideration, we increase the chances of a partitioning heuristic for finding a good solution.

For example, Figure 3(a) demonstrates a pre-assignment clustering of some of the fuzzy-controller functional objects. Looking back to Figure 2, we note that *EvaluateRule* is the only accessor of *trunc* and of *Min*. We thus group those three objects into one new object so that a subsequent N-way partitioning heuristic never tries to separate those original objects. Any N-way partitioning heuristic can be applied to the new objects, such as simulated annealing, or even N-way clustering, which we shall now describe.

3.2 N-way clustering

N-way partitioning is the assignment of the specification's functional objects among N parts, such as among 3 FPGA's, or among 2 processors. N-way clustering is one

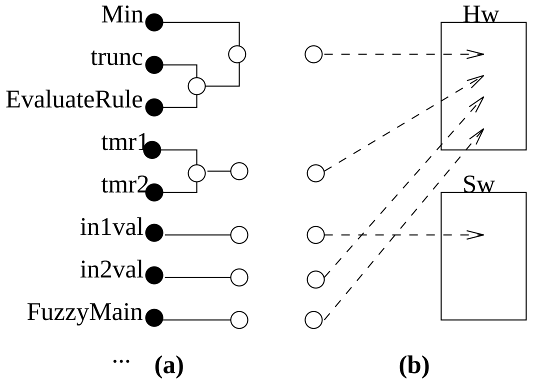

Figure 3: Partitioning functional objects among parts: (a) Pre-assignment clustering, (b) N-way partitioning.

technique for such partitioning, in which we first cluster close objects until the number of remaining clusters equals N, and we then trivially assign each cluster to its own part.

For example, Figure 4 demonstrates N-way clustering of the fuzzy-controller functional objects. Close objects are merged until only two objects remain, corresponding to the two parts among which we are partitioning. No subsequent partitioning is necessary, although we shall see that some iterative improvement proves beneficial.

N-way clustering is one of many alternative techniques for N-way partitioning. However, N-way clustering is particularly useful when estimators of global metrics, which are used by improvement heuristics, are very slow (or non-existent). Such heuristics evaluate global metrics for thousands of different partitions (e.g., in our experiments, simulated annealing averaged over 20,000 partitions per example), so slow estimators make such heuristics infeasible. Clustering provides a reasonable alternative, since we can cluster based on metrics derived directly from the specification, such as connectivity, common accessors and sequential execution.

Note that N-way clustering can be preceded by pre-assignment clustering, just as can any other N-way partitioning heuristic. In other words, the starting objects of Figure 4 could have been the merged objects of Figure 3(b). Such two-stage clustering is in fact an instance of the multi-stage clustering technique described in [4]. We shall see in the next section that each clustering stage requires different closeness metrics.

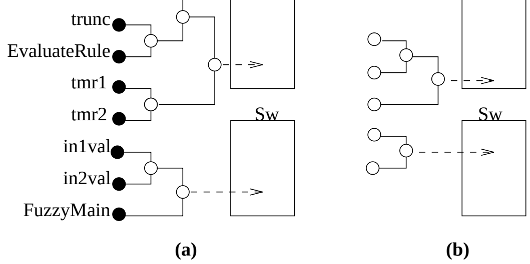

Figure 4: N-way clustering: (a) With original objects, (b) With pre-assignment-clustered objects.

4 Experiments

We now describe results of a number of experiments of pre-assignment clustering and of N-way clustering, and describe how these results help define the role of clustering during system-level functional partitioning.

Clustering results are compared with results of running three iterative-improvement partitioning heuristics on four examples. The *Greedy* heuristic moves objects from one part to another as long as cost reductions are obtained, having a computational complexity of $O(m)$, where m is the number of objects. The *Group Migration (GroupMig)* heuristic [21] yields better results by overcoming some local minima, and is widely used in network (circuit) partitioning. Our implementation has a complexity of $O(m^2 * N)$ (versions for circuit partitioning have achieved a complexity of $O(m * N)$), but the number of iterations often varies from 1 to 7, yielding unpredictable runtimes. The *simulated annealing (SimAnn)* heuristic uses random moves to escape even more local minima, at the expense of generally long runtimes. Annealing parameters included a temperature range of 50 down to 1, a temperature reduction factor of 0.93, an equilibrium condition of 200 moves with no change, and an acceptance function as defined in [21]. Each of the above heuristics started with an initial partitioning generated by a *Random* partitioning heuristic.

The four examples were VHDL descriptions of a volume-measuring medical instrument (*Ex1*), a telephone answering machine (*Ex2*), an interactive-TV processor (*Ex3*), and an Ethernet coprocessor (*Ex4*).

Table 1 provides the results of applying each heuristic on each example. The partition cost C is a unitless number indicating the magnitude of estimated constraint violations [21]. Constraints on hardware size, software size, hardware I/O, and execution time were intentionally formulated such that there would be constraint violations (non-zero cost), so that we could compare how close each heuristic came to achieving zero cost. The runtimes T are the CPU times in seconds running on a Sparc20. Each example was partitioned among *2* ASICs (VTI), *3* ASICs, *4* ASICs, and a hardware/software (*hs*) configuration of one processor (8086) and one ASIC.

4.1 N-way clustering

Our experiments show that N-way clustering alone does not yield better cost or time than the above heuristics, but but that N-way clustering followed by the greedy heuristic yields very good results.

Table 2 provides results of applying clustering to the four examples, under the column *Nclust*. We used a closeness function that was a weighted sum of the normalized connectivity, communication, accessors, and balanced size metrics. These metrics were chosen after performing a 2^k factorial experiment to determine the best combination of metrics for the four examples [20]. For the case of hardware/software partitioning, the more tightly-constrained final cluster was assigned to hardware, the other to software.

In addition to the clustering results, the columns labeled *Nclust_gr*, *Nclust_gm*, and *Nclust_sa* in the table show results of using the partition obtained through clustering as an initial partition for the three improvement heuristics,

Ex	P	Random		Greedy		Groupmig		Simann	
		C	T	C	T	C	T	C	T
1	2	314	0	68	2	40	26	15	147
	3	443	0	50	3	0	20	22	220
	4	428	0	88	7	29	87	16	246
	hs	576	0	61	2	16	21	18	148
2	2	236	0	69	4	43	40	47	146
	3	256	0	25	8	7	198	0	174
	4	234	0	0	11	2	187	0	208
	hs	160	0	0	1	0	7	0	0
3	2	893	0	90	15	68	804	30	403
	3	1081	0	115	32	71	953	63	609
	4	1220	0	141	60	100	2953	94	625
	hs	2115	0	83	12	20	296	20	379
4	2	960	0	105	16	60	655	7	378
	3	1206	0	114	34	114	1064	97	520
	4	1338	0	66	60	39	2036	72	693
	hs	660	0	102	20	23	598	0	268
	Avg	758	0	74	18	40	443	31	323

Table 1: Existing N-way partitioning heuristics

Ex	P	Nclust		Nclust+gr		Nclust+gm		Nclust+sa	
		C	T	C	T	C	T	C	T
1	2	85	12	59	14	13	52	33	161
	3	168	13	96	17	16	90	10	227
	4	218	13	15	20	15	135	7	282
	hs	88	12	66	14	16	47	17	152
2	2	141	26	34	29	34	65	43	175
	3	244	29	16	34	9	214	0	188
	4	339	29	15	43	17	213	0	221
	hs	0	24	0	25	0	39	0	31
3	2	111	147	78	156	78	456	36	517
	3	154	158	142	175	142	804	90	707
	4	141	173	137	200	137	1170	104	827
	hs	147	144	67	151	20	538	20	454
4	2	109	363	62	378	37	976	39	692
	3	155	390	5	422	5	1635	23	906
	4	193	395	37	443	37	2428	34	1069
	hs	102	367	76	378	20	913	0	665
	Avg	150	143	57	156	37	611	29	455

Table 2: N-way clustering results

rather than using a random initial partition. *Time* for each heuristic represents the time for clustering plus the time for improvement.

The table shows that N-way clustering by itself yields very poor costs. Group migration and simulated annealing show small improvements in average cost using the clustered initial partition rather than a random partition; however, the greedy heuristic shows substantial improvement, and fills a gap in the cost/time tradeoff curve. Specifically, consider Figure 5. Note that applying N-way clustering followed by greedy improvement (*Nclust + Greedy*) yields a cost/time point in between the two heuristics of greedy and simulated annealing. This point indicates better cost than the greedy heuristic, but less time than simulated annealing.

Thus, results show that N-way clustering followed by greedy improvement yields a viable alternative to existing N-way partitioning heuristics, and would thus prove beneficial when included in a system-design tool's suite of partitioning heuristics.

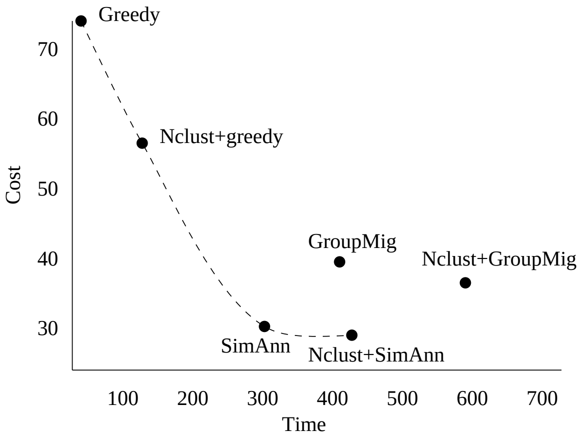

Figure 5: Cost vs. time for N-way partitioning heuristics.

4.2 Pre-assignment clustering

Our experiments show that pre-assignment clustering reduces iterative-improvement partitioning runtime with little or no increase in cost in some cases, and reduces overall cost with no increase in runtime in other cases.

We applied pre-assignment clustering to the four examples, followed by random partitioning and each of the three improvement heuristics. We used a closeness function that was a weighted sum of the normalized connectivity and communication metrics. These metrics were chosen after performing a 2^k factorial experiment to determine the best combination of metrics for the four examples [20]. Through further factorial experiments, we found that the best termination criteria for the clustering process was a closeness threshold of 0.3; in other words, we terminate clustering when no pair of objects has a closeness greater than 0.3.

The plot of Figure 6 summarizes the average effects of pre-assignment clustering on the cost and time of the three improvement heuristics. The pre-assignment clustering lowers the cost obtained by the greedy heuristic, with the same runtime as before. The clustering yields lower runtimes for group migration and simulated anneal-

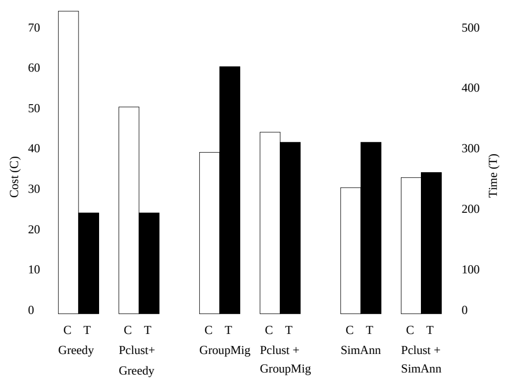

Figure 6: Cost/time effects of pre-assignment clustering.

ing, with nearly no increase in cost; in fact, in 20 cases, there was actually a decrease in cost.

To further demonstrate the effect of pre-assignment clustering on improvement heuristic runtime, we experimented more extensively with the Ethernet coprocessor example. We followed pre-assignment clustering by random partitioning and group migration. Results are shown in Figure 7. The *number of objects* axis represents the number of objects remaining after pre-assignment clustering. Thus, the right end of the axis represents no pre-assignment clustering, meaning all 125 original objects are subsequently partitioned among parts; the center of the axis represents reduction by pre-assignment clustering down to 70 objects. Such reductions were obtained by clustering until reaching a fixed number of objects, rather than using a closeness threshold as earlier. The *cost* axis represents the magnitude of constraint violations of the final partition after group migration was applied to the hierarchical functional objects. The *time* axis represents the computation time (in seconds on a Sparc 2) required by group migration. Results show that the fewer the number of objects input to group migration, the lower the runtime. When we reduced the objects from 125 down to 85, we not only decreased the runtime by 30%, but also found a lower-cost partition. A reduction down to 55 objects decreased runtime by nearly 55%, at the expense of slightly higher cost. Further reduction yields higher-cost partitions, since clustering is then forced to merge distant objects.

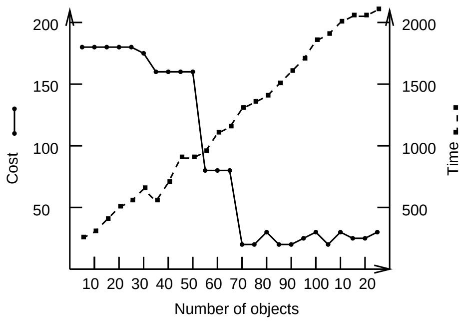

Figure 7: Improvement-heuristic runtime reductions gained by pre-assignment clustering.

5 Conclusions and future work

We have demonstrated two roles of clustering during system-level partitioning. Pre-assignment clustering leads to reduced runtimes of subsequent partitioning heuristics, and in many cases leads to lower cost partitions. Such reduced runtime is especially significant since subsequent partitioning heuristics might be applied hundreds of times. N-way clustering, when followed by a fast improvement-heuristic, complements a suite of iterative-improvement heuristics, and is especially useful when global-metric estimators are slow.

It may be interesting to examine techniques that apply many partitioning heuristics, including clustering, in parallel and select the best result. We could even apply several clusterings in parallel, each with different closeness metrics. Additionally, further investigation of multi-stage clustering may prove beneficial.

References

[1] P. Athanas and H. Silverman, "Processor reconfiguration through instruction-set metamorphosis," *IEEE Computer*, vol. 26, pp. 11–18, March 1993.

[2] R. Camposano and R. Brayton, "Partitioning before logic synthesis," in *Proceedings of the International Conference on Computer-Aided Design*, 1987.

[3] M. McFarland and T. Kowalski, "Incorporating bottom-up design into hardware synthesis," *IEEE Transactions on Computer-Aided Design*, pp. 938–950, September 1990.

[4] E. Lagnese and D. Thomas, "Architectural partitioning for system level synthesis of integrated circuits," *IEEE Transactions on Computer-Aided Design*, July 1991.

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

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

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

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

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

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

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

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

[13] T. Ismail, K. O'Brien, and A. Jerraya, "Interactive system-level partitioning with Partif," in *Proceedings of the European Conference on Design Automation (EDAC)*, 1994.

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

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

[16] D. Thomas, J. Adams, and H. Schmit, "A model and methodology for hardware/software codesign," in *IEEE Design & Test of Computers*, pp. 6–15, 1993.

[17] P. Gupta, C. Chen, J. DeSouza-Batista, and A. Parker, "Experience with image compression chip design using unified system construction tools," in *Proceedings of the Design Automation Conference*, pp. 250–256, 1994.

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

[19] F. Vahid, "Procedure exlining: A transformation for improved system and behavioral synthesis," in *International Symposium on System Synthesis*, 1995.

[20] F. Vahid and D. Gajski, "Closeness metrics for system-level functional partitioning," in *Proceedings of the European Design Automation Conference (EuroDAC)*, 1995.

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