M. Hanan
A. Mennone
P. K. Wolff, Sr.

Iterative-Interactive Technique for Logic Partitioning

Abstract: A method is developed for partitioning a computer logic design into subsets by combining a constructive method, used for the initial partition, with iterative improvement techniques. These iterative techniques are implemented in an interactive computing environment, which further enhances their efficiency and usefulness. An overview of the system is presented, several algorithms discussed and experimental results given.

Introduction

Partitioning an entire computer logic design into packageable subsets is a fundamental problem in the designing and building of computers. Even with the tendency toward larger subpackage sizes in modern computer design, partitioning continues to be a complex problem. One may visualize future designs of an entire computer on one chip, but even this version carries with it the need to subdivide the chip into regions and allocate the appropriate logic to these regions. Partitioning, then, will continue to be an operation that must be performed some time during the design and manufacture of logic systems.

Manual techniques for partitioning have inherent limitations with regard to speed, capacity and accuracy. Fully automatic partitioning methods overcome these limitations but do not account for all constraints, such as delay, testability, etc. It seems obvious that some combination of manual and automatic techniques and a man-machine approach is needed. The difficulty lies in the fact that these automatic techniques that construct a completed partitioning do not lend themselves to manual intervention.

The key to the success of our implementation of this man-machine approach lies in our combination of (1) a constructive method for the initial partitioning and (2) iterative-improvement algorithms. The constructive initial phase is an automatic, batch-oriented program previously reported in [1], and the iterative-improvement phase is the interactive computing portion of our system. This implementation offers the designer a powerful and easily controlled tool on a real-time basis.

Although the iterative-improvement algorithms implemented are not the most general (and possibly not the most powerful) algorithms that might be derived [2], they have shown that the overall system yields excellent results and that the approach should be further developed.

Comparative analyses of heuristic algorithms are always difficult because no standard problems or solutions are available. However, recently a problem and its solution was published [3] with the hopes of establishing this problem as a standard. We ran our algorithm on this problem and also on another problem, about twice as large as the problem in [3]. We compare our results to those obtained with a previously published algorithm and to a manually produced partition. In this paper we present the results of these comparisons, after discussing some historical background and the components of the interactive system.

Background

Manual division of a computer logic design to subsets, each fitting on a package (i.e., partitioning), has been performed ever since computers have been designed and built. It is relatively easy for a human being to accomplish this task if the amount of data that he has to handle is small, say 500 logic gates. This problem size is within the capability of the human memory system. Most of the design can be "viewed" and value judgments can be made leading to a subdivision of the logic. Usually this initial manual attempt does not satisfy all packaging con-

straints, and rework of the original partition begins. Logic is moved from one package to another, constraints such as the number of I/O pins allowed on each package are changed, or more packages are used, and eventually the partition is completed. This process works but is prone to error and requires lengthy effort, scrap and rework.

The amount of time required for a successful manual partitioning, the penalty to be paid for manual error, the ever-increasing quantities of data to be handled, the increasing complexities of partitioning (e.g., the more stringent constraints due to large scale integration), and many other factors contributed to the motivation for partitioning by some automatic method. Various programming techniques have been tried. See, for example [4], which discusses the partitioning problem and contains an extensive reference list. Also, see [1] and [5] for more references.

After a method is developed to automatically produce a partition satisfying some global constraints, it becomes evident that a technique is needed to permit some modification of this global result to account for specific constraints not easily handled. Recognizing this need, we created two iterative improvement techniques which work well at solving local constraints without destroying the global constraints already satisfied.

The pieces fall into place both logically and chronologically: manual partitioning, automatic partitioning, partitioning improvement, and interactive computing. The basic features of each of the components of the system are discussed next.

Components of the system

The system is comprised of both hardware and software components. We discuss the various software components from the standpoint of operational descriptions and then give a brief view of the hardware used to run these software components.

It has been observed that in the case of heuristic algorithms (in particular, placement algorithms) iterative-improvement algorithms are more powerful, i.e., yield better results, than constructive algorithms alone [6], because an iterative heuristic can be used to improve the results of constructive heuristics. This combination guarantees finding a solution that is at least as good as the constructive method alone. (The penalty one pays for this added power is additional computation time.) Therefore two basic components of our system are a constructive algorithm and interactive-improvement algorithms. For our constructive algorithm we chose ALMS (Automated Logic Mapping System) [1, 7] since it was a system which was readily available to us and has proven to give good results.

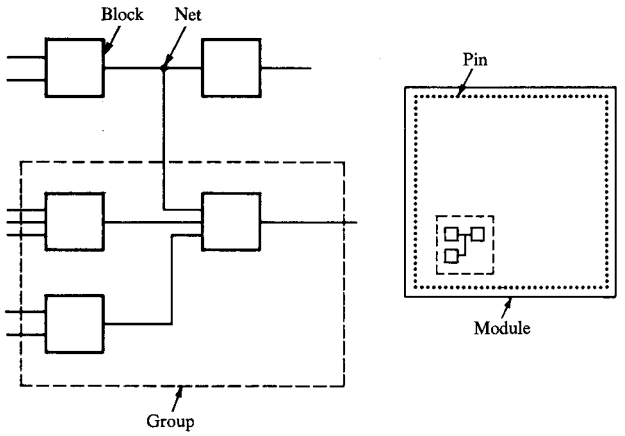

Figure 1 Structure of a logic design consisting of blocks interconnected by a physical network of electrical conductors. Component labels indicate the terminology used in the text.

At this point some definitions are in order. A logic design or logic graph is a structure consisting of blocks interconnected by nets, as indicated in Fig. 1. A block may be a NOR gate, a wired OR (DOTOR), a storage element, a register, etc. A net is a representation of the physical network of electrical conductors connecting those terminals on a subset of blocks that are electrically common. A module is a container that is capable of holding blocks and has provision for making connections from blocks contained on it to blocks contained on other modules. An area attribute is associated with each block and the capability of a module to hold blocks is measured in terms of the area or the number of blocks it can hold. The capacity for connecting nets among modules is specified in terms of the number of pins on the module.

ALMS achieves logic mapping [8] by means of a set of batch-oriented programs that accept as input a description of the computer logic represented as blocks interconnected by nets. Two basic steps are accomplished by ALMS. First, the logic is compressed into groups by the Group Generation Program (GGP). Secondly, these groups are allocated to modules to satisfy certain constraints by the Group Allocation Program (GAP).

For the purposes of this paper it is sufficient to know what GGP does in general. Detailed understanding of its operation can be found elsewhere [1, 7]. GGP groups the logic by a backward trace procedure through the logic structure. The groups formed may contain any number of blocks, which are associated with each other in that they share some nets in common. This "functional" grouping reduces the number of items to be considered by eliminating all connections within groups and creating a structure of connected groups with the nets defined

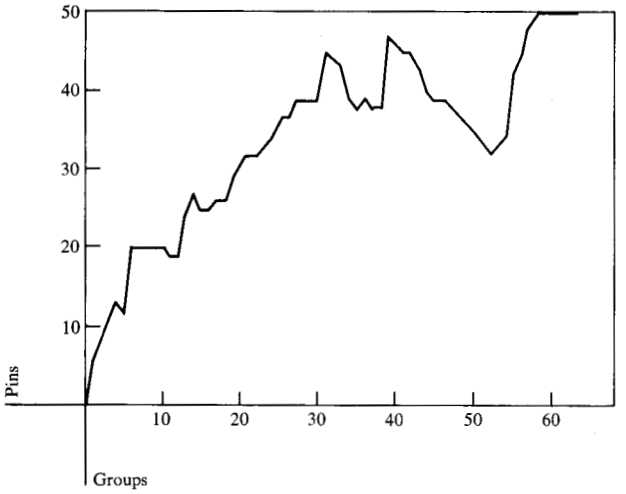

Figure 2 A pin-vs-group growth curve for a single module during the operation of the Group Allocation Program.

between the groups. Hence the original block graph is transformed into a group graph where the groups are superblocks containing one or more blocks.

The second part of ALMS is GAP, of which we give only a brief overview. GAP accepts the group graph output from GGP as input. Other input parameters are the number of modules into which the logic should be partitioned and, for each such module, characteristics such as the maximum amount of area allowed and the maximum number of pins to be used. At each step of the GAP process, a group is assigned to the "best" module such that the maxima for that module are not exceeded. The allocation process is started by "seeding" the modules. After seeding and after each allocation, a candidate set of groups is generated by considering the connectivity of unallocated groups to groups already allocated. A "cost" (which is a function of pins and blocks) of assigning each group of the candidate set to each module is computed, generating a matrix of costs. A heuristic rule (which minimizes cost) is employed to determine which group in the candidate set is assigned to which module. The process terminates either when all groups have been assigned to modules or when there is no group that can be allocated to any module because it would cause the module's maxima to be exceeded. Multiple passes at a solution may be made by allowing GAP to choose new seeds automatically. We consider each pass of GAP as having created a constructive partition and select the result of one of a number of passes as our initial partition. Reference [7] gives a more detailed description of GAP, including the concept of seeding and multiple passes. It is important to note that although the modules are filled in parallel, the groups are assigned to the modules sequentially and that at no time during the allocation process are the module maxima exceeded. These facts will be used during the iterative improvement phase.

For the iterative improvement phase, the basic concept is to start with a given partition (or a partial partition, i.e., one in which not all groups have been allocated), remove a set of groups from one or more of the modules, and reallocate these groups. Two rules, namely, the rule for removing the groups and the rule for reallocating the groups, specify the algorithm. Two specific versions of the removal rule given here are called MULTIPLE and SINGLE. They rely heavily on the characteristics of the allocation algorithm in GAP. In addition the reallocation is done using GAP, i.e., the ALMS allocation algorithm.

As previously mentioned, a characteristic behavior of GAP is crucial in the operation of MULTIPLE; namely, during the GAP allocation process, the pin count never exceeds the maximum pins permissible (P_{max}) for any module. Hence, if we remove the last K groups which were assigned to any module, the pin count on that module must be less than or equal to P_{max}. Also, if the groups are removed in reverse order of allocation, the pin count on any module must eventually decrease to zero (although not monotonically) so that on any given module enough groups can always be removed to reach any pin count less than P_{max}. Figure 2 illustrates the pin-vs-group growth of a module during the GAP process. It is observed, in general, that the removal of groups in reverse order reduces the pin count for the module. The determination of how many groups to remove is made easier by viewing such module growth curves. An illustration of this technique appears in the next section, "Use of the system."

Input parameters to MULTIPLE control the extent of removals and the reallocation. The parameters consist of the number of modules to be changed and for each module changed, the number of groups, blocks and used pins to be removed [9]. These three removal parameters may be used individually or in combination. For example, if we request that five groups and 30 blocks be removed from module 1 then at least five groups will be removed. If after the removal of the five groups only 15 blocks have been taken out, then additional groups will be removed until at least a total of 30 blocks are removed. All removal requests for the module are thus satisfied.

Additional rules control the reallocation of groups to modules. For each module that is changed, an increase or decrease is possible in the maximum number of groups allowed (G_{max}), the maximum number of blocks allowed (B_{max}), and the maximum number of pins allowed (P_{max}).

If the maxima for a module are decreased, enough groups must be removed from that module such that the value of each parameter changed is less than or equal to the new maximum for that parameter before reallocation starts. For example, if seven pins are removed from module 1, decreasing the pin count on module 1 from 43 to 36, then P_{max} for module 1 must not be set below 36. Any, all, or none of the maxima may be changed on any of the modules. There is no requirement that removals must be made if maxima are changed for a given module and, conversely, maxima may be changed without removal so long as the condition stated immediately above is satisfied.

The second of the improvement algorithms is SINGLE. With this method groups are removed and reallocated, one at a time, in the same order as that of the original allocation. As with the MULTIPLE algorithm, there are control parameters which specify the removal rule and the reallocation rule. A group is removed from a module only if the resulting module satisfies the constraints specified by the control parameters. After a group is removed it is reallocated using the GAP allocation process. The process continues by considering each group in turn from the allocation list and terminates when there are no groups that have moved after a complete cycle through the allocation list. Typically the process terminates after two or three cycles. The particular implementation of the SINGLE algorithm chosen for this system attempts, as a major factor, to make moves based on pin requirements. Other parameters could be used as well. The input parameters are:

1) The allowable change in the used pin count on the module from which the group is being removed. If the allowable change is greater than zero, the used pin count is allowed to increase with a removal. If the allowable change is equal to zero, only removals that do not increase the used pin count are acceptable. If a negative value is entered for the allowable change, then only removals which in fact reduce the pin count are made.
2) The allowable change in P_{max} for the module.

Although the order of allocation is identical to the original partition, SINGLE yields an improved partition because the decision to assign a given group A to a given module B is now based on more complete information. The cost of assigning a group to a module during the original GAP allocation is a function of the groups that have already been assigned. If group A were assigned early in the allocation process, only a small portion of the total number of groups to be assigned would have in fact been assigned. Therefore, during the original GAP allocation, assignment was based upon the "best" decision that could be made with the incomplete information then available. In contrast to this assignment,

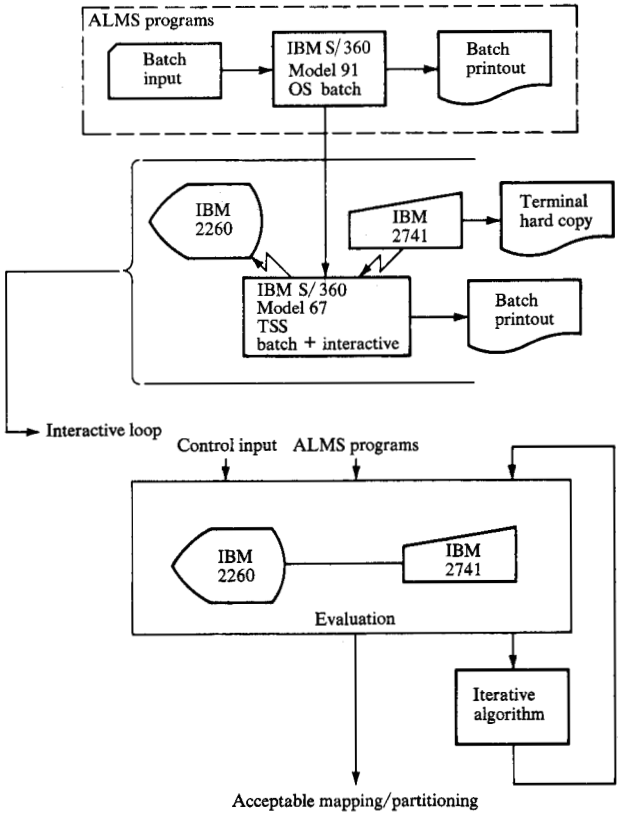

Figure 3 System diagram for the logic partitioning scheme, showing hardware and software components.

during a SINGLE reallocation all other groups have already been assigned to modules and group A may now be reallocated taking this into account. Hence group A may now be assigned to some module other than module B, with a lower cost than the original cost of assigning it to module B. This intuitive rationale for SINGLE may also be applied in part to MULTIPLE.

Our experience indicates that SINGLE is not as effective as MULTIPLE; however, when used iteratively with MULTIPLE it can yield results better than MULTIPLE used alone. Results presented later in this paper bear out this statement.

The hardware used for this system is shown schematically in Fig. 3. The ALMS programs, GGP and GAP, are operational on an IBM System/360, Model 91 running under OS. Our general version of these programs require $\sim$ 450k bytes of core storage. For further data on core requirements see [7].

Batch execution is accomplished with an input deck and the results are displayed on a batch output printer. When a GAP result is chosen for improvement, this result is shipped via a communications link to an IBM System/360, Model 67 running under TSS. (This sys-

Table 1 Four-module partition obtained with ALMS and GAP output. Constraints were 1000 blocks per module and 50 pins per module.

Module No.	*Pins*		*Blocks*	
	Max.	*Used*	*Max.*	*Used*
1	50	46	1000	239
2	50	50	1000	377
3	50	50	1000	223
4	50	50	1000	381
		196		1220

tem was used because it was available to us at the time we began our research. Obviously, the concepts discussed here can be used on any interactive system, such as the VM/370).

The iterative improvement routines, MULTIPLE and SINGLE, the allocation routines from GAP used to reallocate removed groups, and other support software used for interactive keyboard input/output and graphic alphanumeric output are resident in the IBM System/360, Model 67. These also required $\sim$ 450k bytes of core storage. The terminal I/O device is an IBM 2741 and the graphic character output is displayed on an IBM 2260. A history file is also maintained by the interactive portion of the system so that if the user desires, an off-line batch printout of his interactive session can be produced.

Use of the system

The purpose of this section is to show the general aspects of the use of the system by a particular example. The example logic graph contains 1220 gates and 72 primary I/O connections, a "primary connection" being one that is required as input to the logic or as output from the logic. There are 19 DOTOR connections in the logic so that in the block representation of this logic there are 1220 blocks of size 1 and 19 blocks of size 0. Grouping was done by GGP using some manually generated groups. The result of the GGP run was a group graph containing 304 groups that ranged in size from 1 to 77 blocks, with an average of about four blocks per group. GAP was run attempting partitions at five modules with a maximum area (or size) of 350 blocks per module and a maximum of 43 pins per module. We obtained successful results and then further improved them, concentrating on various specific improvement criteria. These results are reported in the next section.

The question arises as to whether a four-module partition with the above constraints can be found. At a maximum of 350 blocks per module, a four-module partition is the minimum possible. That is not to say that a four-module partition exists, because this lower bound does not take pins into account. It is of interest to note that a partition of this logic produced manually by the designers of the logic required four 43-pin, 350-block modules and three 23-pin, 350-block modules. Therefore a four-module solution would eliminate the need for the three 23-pin modules. We ran several passes of GAP in the batch mode with the above constraints. No successful partitions were obtained. Because a four-module partition could not be obtained using ALMS alone, we decided to try a different approach—to relax the constraints when running ALMS and then attempt to bring the constraints down by using interactive partitioning. First, we removed the 350-block constraint and retained the 43-pin constraint but obtained no successful partitions. We then decided to lift the block constraint completely by setting it arbitrarily high to 1000 and raised the pin constraint until an allocation of all groups to four modules resulted. This successful allocation occurred with maximum settings of 1000 blocks per module and 50 pins per module and is shown in Table 1. The maximum used were 381 blocks and 50 pins. We then attempted to improve this result, i.e., return to the original physical constraints of 43 pins and 350 blocks, using our improvement methods.

In this particular example we first illustrate the use of MULTIPLE to attack the block-limit problem. Namely we remove certain groups from some modules, lower the block maximum to 350 on all modules and attempt reallocation under the new constraint. In this case we decided to remove about 20% of the blocks from all of the modules, since this brings the large modules (with respect to blocks) close to their desired level of 350 and provides maneuvering room on the other modules.

Table 2 shows that all groups have been reallocated to modules and that the 350-block maximum constraint has been satisfied. In general it is possible that all groups may not be reallocated to modules, in which case one would try to relax some constraint and reallocate again. As illustrated in Table 2, not only has a successful partition for the 350-block limit been obtained but also the number of pins on the modules has been reduced. This is fortuitous since we were not specifically attempting to lower the pin limit and, in fact, the original maximum of 50 pins had not been changed. However, since we were so close to achieving a 43-pin partition, we attempted another iteration of MULTIPLE but lowered P_{max}.

It is useful at this point to make an observation about the method by which groups are allocated during the GAP execution. This observation was made previously in the section on system components but it bears directly upon our next operation. Figure 1 illustrates the way in which the number of pins for a single module grow

Table 2 Use of MULTIPLE in the block limit problem. Block maximum is lowered to 350. Pin maximum is 50.

Module No.	*Blocks*		*Pins*	
	Original ALMS partition	*After MULTIPLE*	*Original ALMS partition*	*After MULTIPLE*
1	239	290	46	45
2	377	349	50	43
3	223	282	50	36
4	381	299	50	39
	1220	1220	196	163

Table 3 Removal of groups from Module 2 to reach a local minimum with respect to the number of pins.

Removal #	*Module pin count*	*Module block count*	*Remove*
1	43	348	YES
2	44	345	YES
3	45	344	YES
4	45	341	YES
5	45	338	YES
6	45	335	YES
7	45	332	YES
8	45	329	YES
9	44	326	YES
10	45	323	YES
11	46	315	YES
12	48	313	YES
13	36	301	YES
14	34	296	YES
15	34	293	YES
16	34	290	YES
17	33	287	YES
→			
18	34	285	NO

with respect to each group allocated to that module. Examination of this curve shows that there are several local minimum points. By definition, if groups are removed beyond a local minimum point, the pin count increases. Hence our strategy is to remove groups in reverse order until we reach a local minimum on the pin curve and have simultaneously removed a sufficient number of pins, so that we are below the desired maximum. This removal technique is now employed in an attempt to obtain an acceptable partition, i.e., no more than 43 pins per module and 350 blocks per module. Table 3 illustrates the removal of groups from module 2. Note that the removal #18 will increase the pins on module 2 so that a local minimum [10] on the pin count is reached at removal #17, and hence removal #18 is not accepted. This procedure is applied to all four modules in turn and, after the removals for a module are complete, the pin maximum for the module is set to 43. Reallocation then resulted in successful allocation of all groups to modules with the new constraint of 350 blocks and 43 pins on each. Table 4 shows the results.

At this point we have successfully satisfied our primary objective of creating a four-module partition with a maximum of less than 350 blocks and 43 pins. A potential further reduction in total system pins is possible by use of the SINGLE function. By applying SINGLE to this problem and allowing groups only to move from one module to another if they either preserve or reduce the pins used on both modules, we can lower the total pins required and still preserve the original maxima of 43 pins and 350 blocks. A group is removed from a module if it either decreases or preserves the pin count on that module. The best module to which the group should be allocated, i.e., the one resulting in lowest cost, is then found. In some cases the best module is the module from which the group was removed originally. In this case the group is placed back in its original position. This constitutes a valid removal but not a valid move.

Table 4 Reallocation to reduce P_{max} to 43 pins.

	Blocks		*Pins*	
Module no.	*After first iteration of MULTIPLE*	*After second iteration of MULTIPLE*	*After first iteration of MULTIPLE*	*After second iteration of MULTIPLE*
1	290	279	45	40
2	349	346	43	42
3	282	303	36	42
4	299	292	39	35
	1220	1220	163	159

Table 5 Reallocation to reduce total number of pins using the SINGLE function.

	Blocks		*Pins*	
Module no.	*After second iteration of MULTIPLE*	*After third iteration of SINGLE*	*After second iteration of MULTIPLE*	*After third iteration of SINGLE*
1	279	281	40	39
2	346	335	42	36
3	303	320	42	38
4	292	284	35	31
	1220	1220	159	144

Table 6 Summary of example results.

	Manual method		*Unimproved ALMS*		*Final improvement*	
Chip no.	*Blocks*	*Pins*	*Blocks*	*Pins*	*Blocks*	*Pins*
1	147	21	239	46	281	39
2	260	23	377	50	335	36
3	131	23	223	50	320	38
4	188	37	381	50	284	31
5	165	42		196		144
6	222	34				
7	107	29				
		209				

Table 7 Additional results of partitioning using iterative improvement methods. Details of the four improvement methods are given in the text.

	Initial partition		*Improvement (1)*		*Improvement (2)*		*Improvement (3)*		*Improvement (4)*	
Module	*Blocks*	*Pins*	*Blocks*	*Pins*	*Blocks*	*Pins*	*Blocks*	*Pins*	*Blocks*	*Pins*
1	146	42	195	43	224	40	230	43	150	35
2	340	43	280	30	277	30	269	25	334	35
3	158	27	187	42	162	29	147	21	159	34
4	284	31	278	37	277	33	284	31	287	35
5	292	24	280	27	280	23	290	23	290	23
Total pins		167		179		155		143		162

On the other hand, a group may in fact move to a new module, in which case it is both a valid removal and a valid move.

In our example we applied SINGLE in the way mentioned above. Three cycles of SINGLE were required before no further valid moves were possible. There were 29 valid removals and nine valid moves in cycle 1, 28 valid removals and two valid moves in cycle 2, and 27 valid removals and no valid moves in cycle 3. Table 5 illustrates the results of using SINGLE. We observe that the total pins required has been reduced from 159 to 144.

These partitions satisfy our primary constraints of 43 pins and 350 blocks. We can save these results for subsequent improvement or for feeding into other programs. The system has a history-keeping facility and we can print this history offline for use at our desk. At some later date we may decide to make further modification to these results and we use the saved result as a starting point.

In summary, we have taken a section of computer logic previously manually partitioned to seven chips, four of which had 43 pins maximum and three of which had 23 pins maximum (both with a maximum of 350 blocks), and have created a four-chip partition with 43 pins and 350 blocks maximum. Hence we have eliminated the three 23-pin chips completely. Various automatic partitioning attempts had been made unsuccessfully with ALMS to solve this problem. The power of such iterative-interactive techniques is apparent in the results in Table 6. Compared with the manual partition, three chips have been eliminated. Compared with automatic partitioning without iterative improvement, an unfeasible solution was made feasible. In addition, the maximum pin count per module was brought to 39, which is below the 43-pin maximum allowed, and the total system pins were reduced from 196 to 144.

Additional results using iterative improvement techniques

We show several other interesting results using the same 1220-block section of logic discussed in the previous section. For these purposes we choose from among several successful ALMS-produced five-module partitions at 350 blocks and 43 pins. The partition that has the least number of total pins is chosen as best and the results shown under the heading "Initial partition" in Table 7.

Our first objective was to reduce the number of blocks on the densely populated modules, such as module 2, while holding the number of modules constant at five. The motivation for this is that, in general, the more blocks on a module, the more difficult it is to wire that module, assuming that all modules are the same physical size. If the block count of the densely populated modules can be reduced without increasing the number of modules, then the probability of completing the wiring of all the modules is increased.

By applying MULTIPLE to the initial partition, the results shown in Table 7 under "Improvement (1)" were obtained. The maximum block count on any module has been reduced from 340 to 280. This was accomplished by removing approximately 20% of the blocks from each of the modules, reducing all module block maxima to 280, and reallocating the removed groups.

Another objective was to reduce the total number of pins over all modules. The partition shown as "Improvement (1)" in Table 7 was used as the starting point to maintain our previously obtained maximum of 280 blocks per module. The method of attack was to use MULTIPLE and SINGLE, iteratively and interactively. The intermediate results were used to determine the next move. This user system interaction produced the results shown in Table 7 as "Improvement (2)," the total system pins being reduced from 179 to 155. A second attempt at reducing total system pins starting with the initial partition, i.e., permitting modules to grow to 350 blocks, and working interactively, produced "Improvement (3)." In this case the number of pins was reduced from 167 to 143.

Another parameter that may be attacked is the maximum number of pins used on any module. Reducing this maximum below the allowable number leaves spare pins on the modules. These spares may be used for test points or for adding engineering changes to the chips at a later date. Again starting with the initial partition and using MULTIPLE and SINGLE interactively, the maximum number of pins per module was reduced from 43 to 35, leaving at least eight spare pins on each module. This result is shown as "Improvement (4)" in Table 7.

These techniques were also applied to the problem of partitioning the logic graph presented in [3], where Mennone and Russo present a series of partitions for a graph containing 671 blocks, and 77 primary I/O pins. This graph is to be used as a basis of comparison for other partitioning techniques. Their data is the result of repeated use of the batch-oriented ALMS program. Presented are the best results obtained for each data point. Several of these data points were attacked from the position that the maximum number of pins per module be held constant. Our objective was to either lower the total number of pins required and/or the number of modules required. Table 8 shows the results of this effort. In the cases of 18 pins per module and 48 pins per module, interactive use of our technique yielded one fewer module. In the other cases shown, we obtained a decrease in total system pins required [11].

Another tack was taken using this same data—we held the number of modules constant and attempted to lower

Table 8 Partitions with lower requirement for total number of pins and/or number of modules.

Max pins/module	Total pins		Modules used	
	Previous	New	Previous	New
18	697	689	40	39
44	381	361	9	9
48	359	328	8	7
52	349	323	7	7
56	302	295	6	6
60	291	281	5	5

Table 9 Partitions with lower requirement for maximum number of pins, and number of modules held constant.

Modules used	Max pins/module		Total pins	
	Previous	New	Previous	New
7	52	48	349	328
6	56	54	302	315
5	60	58	291	278
4	68	66	259	261

the maximum pins required over all modules. In Table 9 we see that, in the four cases tried, a two-to-four pin reduction in the maximum allowable pins per module was realized after interactive-iterative improvement. Some cases yielded a slightly higher total pin count but this is not surprising, since this parameter was not of primary concern in this experiment. Again the trend is clear and it is felt that similar reductions are possible across the range of results presented in [3].

Computation time

The question of computation time arises whenever heuristic solutions to complex problems are discussed. In the case of our approach, three time factors are important: 1) the CPU time required to execute the algorithms, 2) the response time of the time sharing system, and 3) the elapsed time required to obtain a solution.

In the case of MULTIPLE the amount of CPU time required depends directly upon the number of groups to be reallocated, since the removal process requires only a small fraction of the total CPU time. To give the reader an appreciation of these numbers, one pass of MULTIPLE, in which approximately 25% of 304 groups are removed and reallocated, takes approximately 8-12 seconds of CPU time on an IBM System/360, Model 67 running under TSS. One pass of SINGLE for the same 304 groups requires about 2-3 seconds of CPU time. The CPU time and core storage of ALMS is discussed in [7].

The second factor is response time. Obviously in a time-shared environment, response time is directly related to the load placed on the system by all active tasks. Response time is therefore difficult to quote because of the variability of the user environment on a minute-by-minute basis. In the best case, when the system is lightly loaded, the user may wait several seconds for each second of CPU time. This response time is more than adequate in that it is the user who slows down the time to solution, because more speed is available than he can comfortably and sensibly use. On the other hand, when the system is heavily loaded, the user may wait as much as one minute for each second of CPU time. This response time is less than adequate, and the user senses long periods of inactivity. It has been our experience that during times of average system loading, response time is adequate for a productive terminal session.

The elapsed time to solution is yet another factor. In this regard the choice of an interactive implementation makes the results of various runs available to the user in minutes rather than hours. During one interactive session of several hours the user may accomplish more work than could be done over several days using the same algorithm implemented in a batch environment. There is no need to wait several hours for batch results that will indicate the further batch runs that are to be made. Also fewer runs are needed, since the user pursues only those paths of investigation which look promising.

Extensions

It has been stated that constructive-initial placement followed by an iterative-improvement algorithm is the best approach to the placement problem [6]. We have now extended this approach to a constructive starting solution followed by iterative improvement to the partitioning problem. Previous papers used either a constructive method alone [1] or a random starting solution followed by iterative improvement [5]. Other iterative improvement techniques can be developed. Some possibilities are described below.

It is possible to define other removal rules which do not depend so heavily on the characteristics of the GAP algorithm. For example, we can find sets of groups that are highly interconnected and use these as candidates for removal. The groups can be chosen by determining their contribution to the total number of pins or the total block count on the module.

A simpler method is the pairwise interchange algorithm applied to the groups of a pair of modules. With this method all possible exchange pairs are tried and if the exchange results in an improvement the interchange is accepted. A clever extension of pairwise interchange is discussed in [5].

Still another approach is to use a method similar to Steinberg's algorithms for the placement problem [12]. In this method a set of independent groups (i.e., groups which are not connected to each other) are removed from a set of modules, and the cost of assigning each group to each module can be computed independently of the cost of assigning all other groups in the set. The difference between this and the placement problem is that we do not arrive at the standard assignment problem because of the added constraints of maximum pins and maximum circuits per module.

The intent of this paper is to show the advantages of the combination of a constructive-initial partition with an iterative-improvement algorithm, implemented in an interactive mode, over previously developed algorithms. Generalizations of the removal rule or the iterative-improvement algorithm, e.g., as discussed above, may lead to even better results.

Summary

The results obtained and summarized in Table 5 show success at creating a four-module, 350-block maximum, 43-pin maximum solution where one was not possible by other techniques. The final result was a subsequent improvement, providing a maximum of only 335 blocks and 39 pins, well below the required level. Tables 8 and 9 show improved results when the system was applied to partitioning a computer logic graph that was established expressly for comparing partitioning results. The magnitude of the improvement in Tables 8 and 9 over the partitioning reported in [3] is not as great as the improvement shown in Table 5. This is to be expected, since the original results in [3] had been achieved by numerous repeated applications of the batch-oriented ALMS program and hence were already "excellent" partitions [13].

We have thus demonstrated that for the logic graphs studied, with the combination of a constructive initial partition and an iterative-improvement technique implemented in an interactive mode, it is possible to achieve better partitions more quickly than with other known methods. We have shown the efficiency of the approach; more research is necessary to find the best iterative-improvement algorithms.

Acknowledgment

We thank Carol Thompson for providing the routines that made the IBM 2260 available as an output device for a FORTRAN applications program.

References and notes

1. R. L. Russo, P. H. Oden, and P. K. Wolff, Sr., "A Heuristic Procedure for the Partitioning and Mapping of Computer Logic Graphs," *IEEE Trans. Comput.* **C-20,** 1455 (1971).
2. Our choice of algorithms was based to a large extent on the tools, techniques, and programs immediately available to us. The expedient was to build on the existing implementation of our constructive initial partitioning program.
3. A. Mennone and R. L. Russo, "An Example Computer Logic Graph and its Partitions and Mappings," IBM Thomas J. Watson Research Center, Yorktown Heights, N.Y., Report RC-4319, March 29, 1973. (To appear in *IEEE Trans. Comput.*)
4. U. Kodres, "Partitioning and Card Selection," Ch. 4 in *Design Automation of Digital Systems: Theory and Techniques*, M. A. Breuer, ed., Prentice-Hall, Inc., Englewood Cliffs, N.J., vol. 1, 1972.
5. B. W. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," *Bell System Tech. J.* **14,** 291 (1970).
6. M. Hanan and J. M. Kurtzberg, "Placement Techniques," Ch. 5 in *Design Automation of Digital Systems: Theory and Techniques*, M. A. Breuer, ed., Prentice-Hall, Inc., Englewood Cliffs, N.J., vol. 1, 1972.
7. R. L. Russo and P. K. Wolff, Sr., "ALMS: Automated Logic Mapping System," *Proceedings of the SHARE–ACM–IEEE Design Automation Workshop*, June 28-30, 1971, p. 118.
8. Mapping is the allocation of blocks to modules so that each block is assigned to at least one module. If a given block is assigned to more than one module, this is termed redundancy or replication. If each block is assigned to exactly one module, the mapping is said to be a partition. In this paper we refer strictly to partitions, although these concepts can easily be extended to the more general mapping case.
9. Groups are always removed because these are the basic elements being operated upon. If blocks or pins are specified as the removal parameter, groups are removed until the number of blocks or pins actually taken out is at least equal to the number specified for removal.
10. Note that there is also a local minimum at removal #9. We continue, however, because we have not satisfied the second criterion that the pin count be below 43.
11. All the Mennone-Russo data points were not run because of the limited time available, but the results given here indicate that one would expect similar characteristics for the other data points.
12. L. Steinberg, "The Backboard Wiring Problem: A Placement Algorithm," *SIAM Review* **3,** No. 1, 37 (1961).
13. R. L. Russo, private communication.

Received December 14, 1973

The authors are located at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598.