Automated Task Allocation and Processor Specification Strategies for Multi-Computer Systems

Jim Beck, Dan Siewiorek

EDRC 18-50-94

September 2,1994

Automated Task AUocation and Proc3essor Specification Strategies for Multi-Computer Systems

Jim Beck **Dan Siewiorek**

Department of Electrical and Computer Engineering
and the Engineering Design Research Center,
Carnegie Mellon University

Abstract

This paper considers the coupled design problems of task allocation and processor specification for embedded multi-computer systems. Two unique problem representations are proposed. The first representation involves multi-dimensional bin packing while the second is based on graph partitioning. Automated solution strategies are developed and evaluated for both representations. The paper concludes with a discussion of the results, pending research, and areas of future work.

Contents

Contents

1 Problem Definition

1.1 Overview

The research reported in this document is aimed at automating a portion of the design process for a specific class of computers, namely embedded, multi-computer systems. Such systems have several distinguishing features. First, they are *embedded.* This implies a stand-alone system that is dedicated to a single function and that executes a single set of software routines. This is opposed to general purpose systems that can be programmed to solve a wide variety of problems. Second, as the term *multi-computer* implies, they consist of a network of loosely-coupled, autonomous processors. The processors communicate with each other at a high level via message passing over a communication network. The communication network could be arbitrary, but this research only considers broadcast bus-based systems.

When designing such systems, the software application must first be decomposed or *partitioned* into a set of communicating software tasks. The tasks ate then statically *allocated* to processors in the system. At the same time, the hardware requirements of the processors must be determined and *specified.*

Each software task has a demand for the resources available on the processor to which it is assigned. The demand imposed by a software task can occur across many dimensions, such as throughput, memory and I/O channels. Accordingly, each processor must be specified and designed to meet the cumulative demand of the software tasks assigned to it. A set of processor specifications and an assignment of tasks to processors that satisfy all task requirements without over-utilizing any of the hardware components are said to *befeasible.*

Often, the specification of processors is complicated by the desire to optimize an *objective function.* An objective function could be any measurable system parameter, such as cumu-

lative cost or power consumption. If an objective function is given, then the design goal is to find a set of processor specifications and an assignment of tasks to processors that are *feasible* and that optimize the *objective function.*

The task allocation and processor specification problems are *coupled,* since their solutions are *mutually constrained.* Because of feasibility considerations, the assignment of tasks to processors *constrains* the specification of each processor. Likewise, the specification of a processor *constrains* task allocation, since it limits the subset of tasks that can be assigned to it. The research described in this report investigates automated ways of solving these coupled problems. Again, the goal is to obtain a set of processor specifications and an assignment of tasks to processors that *axe feasible* and that optimize a stated *objective function.*

The rest of this section is organized as follows: Sections 1.2 and 1.3 describe in detail the software and hardware models and assumptions. Section 1.4 describes how task assignment is modeled. Section 1.5 discusses feasibility. Section 1.6 considers objective functions. Section 1.7 concludes with a concise restatement of the problem.

1.2 Software Model

The software model that is used is a Data Flow Graph (DFG). The nodes in the graph correspond to tasks and the arcs represent inter-task communication. This type of model is frequently used to represent signal processing applications [4], but is not limited to that domain. An example DFG (taken from [4]) is shown in Figure 1.

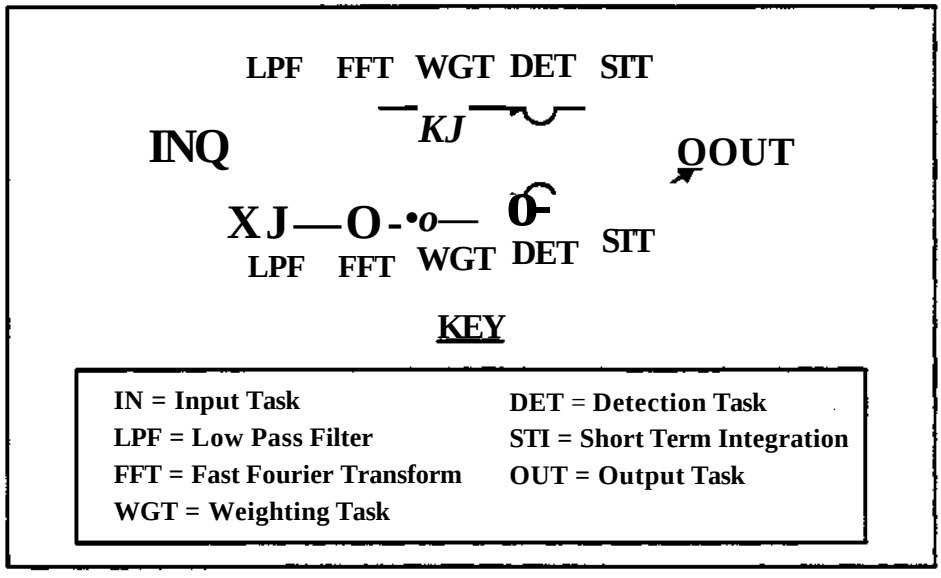


Figure 1
ExampleDFG

There are some hidden caveats associated with the model. First, only software applications that have statically predictable task sequences are supported. This excludes things like recursion, for example. Second, the tasks themselves are assumed to have statically predictable resource and communication requirements. This is required for static task allocation. If this was not true, the effectiveness of a static assignment of tasks to processors would be workload-dependent.

1.2.1 Task Model

The resource demand imposed by a task can occur across many dimensions, such as throughput, memory and I/O. The task model must capture the resource requirements that are pertinent to the design problem at hand. Such requirements will be application specific. For example, computationally intensive signal processing applications are primarily concerned with throughput, while demand for all other resources is secondary. Conversely, automotive applications have modest throughput requirements while sys-

tern cost, which is sensitive to the mix of memory and I/O in each processor, is a dominant concern.

In general, the task model will contain a *vector* of resource *requirements.* Each vector element corresponds to a global resource available in the processor. The design automation techniques that have been developed are compatible with any such well-formed task model.

During the development and verification of the design automation algorithms, the task model shown in Figure 2 was used.

Task = (α, β, χ, δ, ε, ϕ, γ)	
a = Period (3 = Xput Requirement X = Code Size 5 = Data Size	e = Digital I/O Channels $ = Analog I/O Channels y * PWM Output Channels

Figure 2
TaskModel

The a-value represents the task's invocation period. The other parameters constitute its demand vector. Specifically, the p-value is the CPU throughput requirement, the %- and 8-values are memory requirements and the £-, ⌖- and y-values are I/O channel requirements.

122 Communication Model

Likewise, a model is needed for inter-task communication. Unlike the task model, the communication model is one-dimensional. It merely specifies the exchange of data

between tasks. The communication model used during the development of the design automation algorithms is shown in Figure 3.

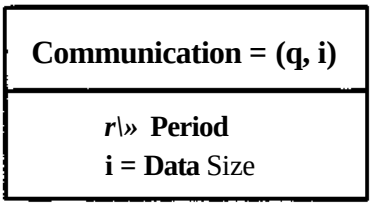


Figure 3
CommunicationModel

1.3 Hardware Model

The hardware model consists of an arbitrary number of heterogeneous processors communicating via message passing over a broadcast bus.

1.3.1 Processor Model

like the task model, the processor model is multi-dimensional. When specified, a processor contains a *vector* of resource *capacities.* The processor model, therefore, consists of all sets of valid capacities that can be selected for each vector element. The union over all sets of valid capacities represents all possible processor specifications, and hence the hardware design space for a processor. Again, the processor model should be chosen based on the design problem at hand, and it should be compatible with the task model.

The processor model used during the development and verification of the design automation algorithms is shown in Figure 4.

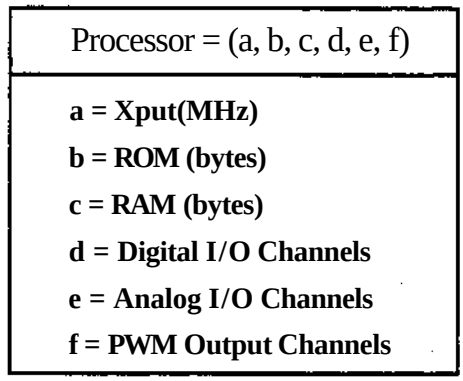


Figure 4
ProcessorModel

1.3.2 Broadcast Bus Model

like the communication model, the broadcast bus model is one-dimensional. The model used during algorithm development was based on a Controller Area Network (CAN) link [5]. It is shown in Figure 5.

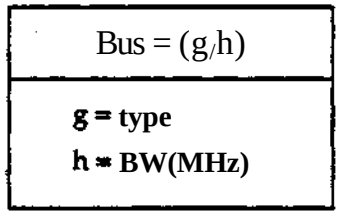


Figure 5
Broadcast Bus Model

1.4 Assignment Model

The set of tasks assigned to a given processor constitutes the *task set* for that processor. A set of constraints is needed to determine whether the execution of the task set on the processor is feasible; this is discussed in detail in Section 1.5.

Assigning communicating tasks (i.e. tasks joined by an arc in the DFG) to the same processor results in intra-processor communication. Similarly, assigning tasks to different processors generates inter-processor communication. Intra-processor communication is *free,* since it involves sharing data within the same, local data space. Inter-processor communication, however, requires message passing over the broadcast bus, as illustrated in Figure 6.

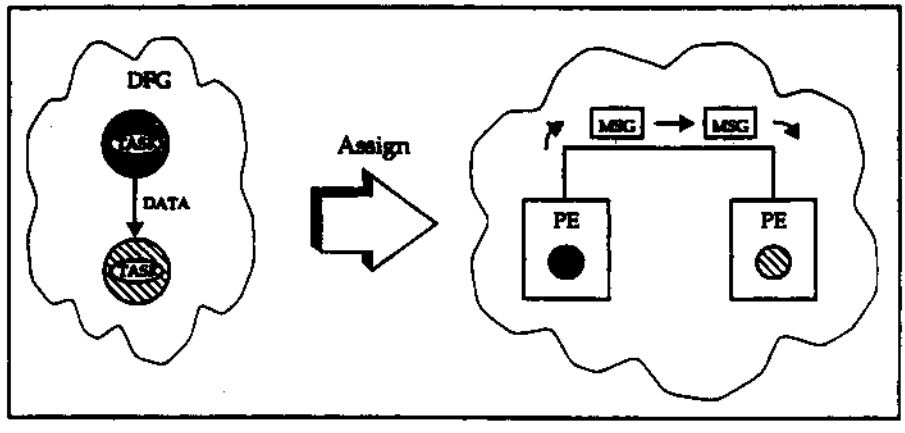


Figure 6
Inter-Processor Communication results in Message Traffic

The set of all inter-processor communications forms the *task set for* the broadcast bus. The amount of time (i.e. bus cycles) needed to carry out any given transfer will depend on the bus being used. Again, a set of constraints is needed to determine whether the bus can handle the transmission needs of the task set; this is also discussed in Section 1.5. Regard-

less, the broadcast bus has a finite capacity for message traffic which, alpng with the communication requirements specified in the DFG, place additional constraints on feasibility

1.5 Feasibility

A set of processor specifications and an assignment of tasks to processors that satisfy all task requirements without over-utilizing any of the hardware components are said to be *feasible. A* set of *constraints* is needed to define the feasibility condition. Such constraints will be specific to the hardware components and operating systems used in the design. To preserve generality, the design automation techniques can accommodate any set of concise, computationally tractable constraints. The set of constraints that were used when developing the design automation algorithms are presented in the subsections that follow.

1.5.1 Processor Utilization Constraint

The processor utilization constraint insures that the throughput of each processor is not over-utilized. It is shown in Figure 7.

$$\forall (i,k)\left(\sum_{i \Rightarrow k} \frac{\beta_i}{\alpha_i} \le a_k\right)$$

i € DFG Graph Nodes
k 6 Processors
B| = Xput required for task i
a; = Period for task i
^{a}k = Xput capacity of processor k
(i =>*k)* = task i assigned to processor k

Figure 7
Processor Utilization Constraint

1.5.2 RAM and ROM Utilization Constraint

Processor memory is distinguished as being either RAM or ROM. Constraints are needed to ensure that the amount of RAM and ROM available in each processor is not over-utilized. These are shown in Figure 8.

$$\forall(i,k)\left(\begin{array}{l}\sum_{i\Rightarrow k} \chi_i \le b_k \\ \sum_{i\Rightarrow k} \delta_i \le c_k\end{array}\right)$$

i € DFG Graph Nodes
k 6 Processors
Xi = ROM required for task i
$i = RAM required for task i
bfc = ROM capacity of processor k
c_k = RAM capacity of processor k
(i =>k) s task i assigned to processor k

Figure 8
RAM and ROM Utilization Constraints

1.5.3 I/O Channel Utilization Constraints

I/O channels are distinguished as being either digital, analog or pulse. Simple constraints are needed to ensure that the availability of each I/O channel type is not over-utilized. These are shown in Figure 9.

$$\forall (i,k) \left(\begin{array}{l} \sum_{i \Rightarrow k} \varepsilon_i \leq d_k \\ \sum_{i \Rightarrow k} \phi_i \leq e_k \\ \sum_{i \Rightarrow k} \gamma_i \leq f_k \end{array} \right)$$

i 6 DFG Graph Nodes
ke Processors
q = Digital I/O required for task i
<fc = Analog I/O required for task i
Yi = PWM Outputs required for task i
dfc = Digital I/O on processor k
e^ = Analog I/O on processor k
fk = PWM outputs on processor k
(i =>k) = task i assigned to processor k

Figure 9
I/O Channel Constraints

1.5.4 Bus Bandwidth Utilization Constraint

A constraint is needed to ensure that message transfers over the bus do not over-utilize the available bandwidth. Before this can be done, however, a model is needed that predicts transmission time for data transfers of arbitrary size. Such a model was derived for the CAN bus in [2] and is reproduced in Figure 10.

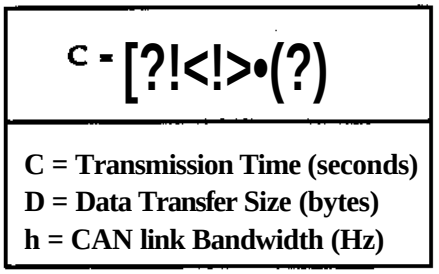


Figure 10
Transmission Time Model for CAN Bus

The bus bandwidth utilization constraint is shown in Figure 11.

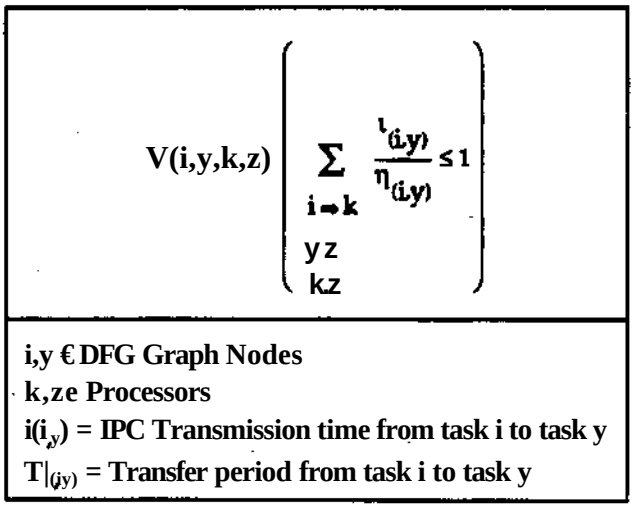


Figure 11
Bus Bandwidth Constraint

1.6 Objective Functions

Solution of the processor specification problem is often complicated by the desire to optimize an *objective function.* An objective function can be any easily computed figure of merit that is relevant to the design. The design automation techniques that were devel-

oped can accept any concise, computationally tractable objective function stated in terms of the processor specifications.

The objective function that was used when developing the design automation algorithms was based on cumulative system cost. The system was assumed to consist of a network of heterogeneous, custom, single-chip microprocessors. The cost of each microprocessor was based on a non-linear function of its die size, and die size was specification-dependent. The die size contribution for each variable in an example processor model is shown in Figure 12. The cumulative system cost function that was used is shown in Figure 13.

Module	Size	Area ($mils^2$)
Core	-	7000
ROM (bytes)	Ik	1800
	2k	2400
	4k	3200
	8k	4400
	16k	6800
	24k	9600
	32k	11000
	64k	20000
RAM (bytes)	256	1100
	512	1900
	Ik	3400
	2k	6600
	4k	11800
	8k	22000
	16k	38500
Digital I/O	0	0
	32	550
Analog I/O	0	0
	8	2200
PWM Outputs	0	0
	2	2000
CAN Interface	-	2900
Routing	-	30% of Total Area

Figure 12
Die Size Contributions far Processor Model Variables

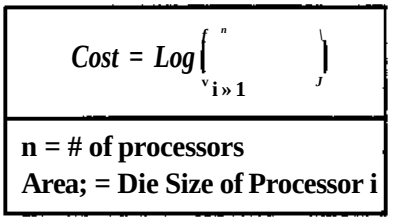


Figure 13
CostFunction

The area and cost functions were adapted from models for an existing family of dedicated processors. They were chosen simply because they provided a realistic set of design trade-offs. Other functions can be substituted into the optimization algorithms.

1.7 Summary

This section has introduced and described the problem being considered, along with the models and assumptions that were used. A concise statement of the problem is shown in Figure 14.

Given:
1). A software application represented by a DFG
  a). Task Model
  b). Communication Model
2). A hardware architecture
  a). Processor Model
  b). Broadcast Bus Model
3). A set of Feasibility Constraints
4). An Objective Function

Do:
1). Determine the number of processors
2). Specify all processors
3). Assign each node in the DFG to a processor

Such That:
1). All feasibility constraints are satisfied
2). The objective function is optimized

Figure 14
Problem Definition

2 Representation One: Packing-Based

2.1 Problem Representation

Once a processor has been specified, only a subset of the DFG nodes can be assigned to it without violating any feasibility constraints. This leads naturally to a *packing-based* representation of the problem defined in Section 1.

Each processor can be viewed as a *bin* having a vector of resource capacities. Similarly, each task is an *object* with a vector of resource requirements. Likewise, the communication bus can be treated as a *scalar bin* with a capacity equal to its bandwidth. The task allocation problem becomes a matter of *packing* the multi-dimensional *objects* into the multi-dimensional *bins.* Feasibility requires that none of the bins, including the scalar bus bin, overflow. Solution of the coupled design problems amounts to developing a method of

successively or incrementally specifying the processors and invoking a packing algorithm to perform task allocation. The processor specification that optimizes the objective function and that can be successfully packed is chosen as the solution.

Based on this problem representation, an algorithm for finding the optimum solution to the coupled design problems is shown in Figure 15. This algorithm is clearly exponential and computationally intractable. Thus, heuristic techniques will be investigated that are tractable and that return near-optimal solutions.

```
For i = 1 to #Graph Nodes [
   For All Specification Combinations for i Processors I
      U(Packable){
         If (Cost < Best) [
            Update Best
         }
      }
   }
}
Return Best
```

Figure 15
OptimumAlgorithm

The rest of this section is organized as follows. Section 2.2 investigates multi-dimensional bin packing and task allocation. Section 2.3 introduces and evaluates four distinct solution techniques based on the packing paradigm. Section 2.4 concludes with a discussion of the results.

2.2 Multi-Dimensional Bin Packing

2.2.1 Goal

The goal of this section is to state the task allocation problem as a *multi-dimensional bin packing problem*. Solution techniques are then proposed and investigated. The major result of this section is a heuristic algorithm that performs task allocation.

2.2.2 Background and Problem Definitions

Bin packing is a well understood and investigated NP-complete problem [7], [10]. It has been previously used to model the task allocation problem for global and distributed memory multiprocessors [4]. The work reported in [4], however, only considered bins with *scalar*, uniform capacities. Thus, task allocation decisions were *one-dimensional*, based solely on throughput requirements.

The task allocation problem now being considered must balance the demand for resources across many dimensions. This requires a *multi-dimensional* extension to the bin packing problem, as well as algorithms that solve this new problem.

To understand the connection between bin packing and task allocation, first consider the task allocation problem that results when communication between tasks is ignored. This problem is isomorphic to the packing problem defined in Figure 16. As the figure indicates, the decision problem amounts to whether or not a set of vector objects can be packed into a set of vector bins without overflowing any bin. Clearly, this packing problem is just a *multi-dimensional extension* to the bin packing decision problem. Before, bins were characterized by a single, *scalar capacity*. Now, however, a *vector* of resource *capacities* is needed to represent a bin.

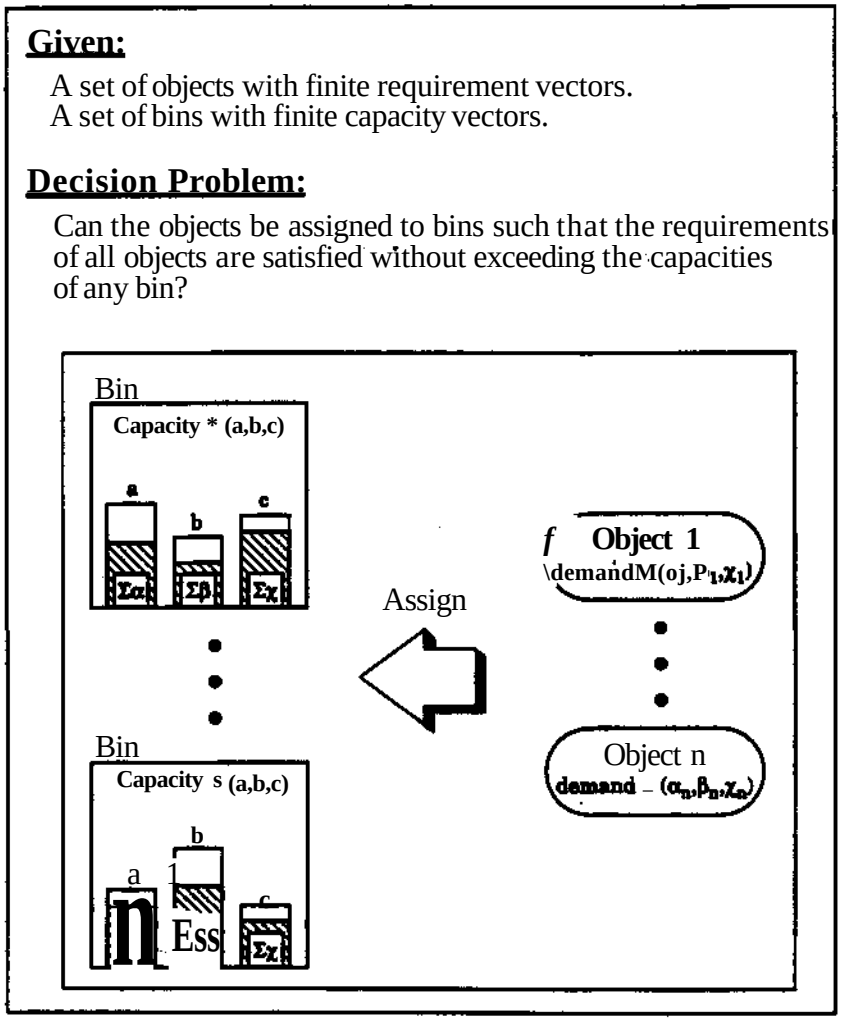


Figure 16

Multi-Dimensional Bin Packing Problem (without Communication)

Next, assume that communication between tasks is not ignored. Now, the task allocation problem is isomorphic to the packing problem defined in Figure 17. In this case, an additional scalar bin is used to model the bandwidth capacity of the bus. Whenever communicating tasks (objects) are assigned to different processors (bins) then a portion of the bus bandwidth (scalar bin) is consumed. The decision problem amounts to whether or not a set of vector objects can be packed into a set of vector bins without overflowing any bin,

including the scalar one. Note, of course, that demand for the scalar bin is actually a function of the assignment of vector objects to bins.

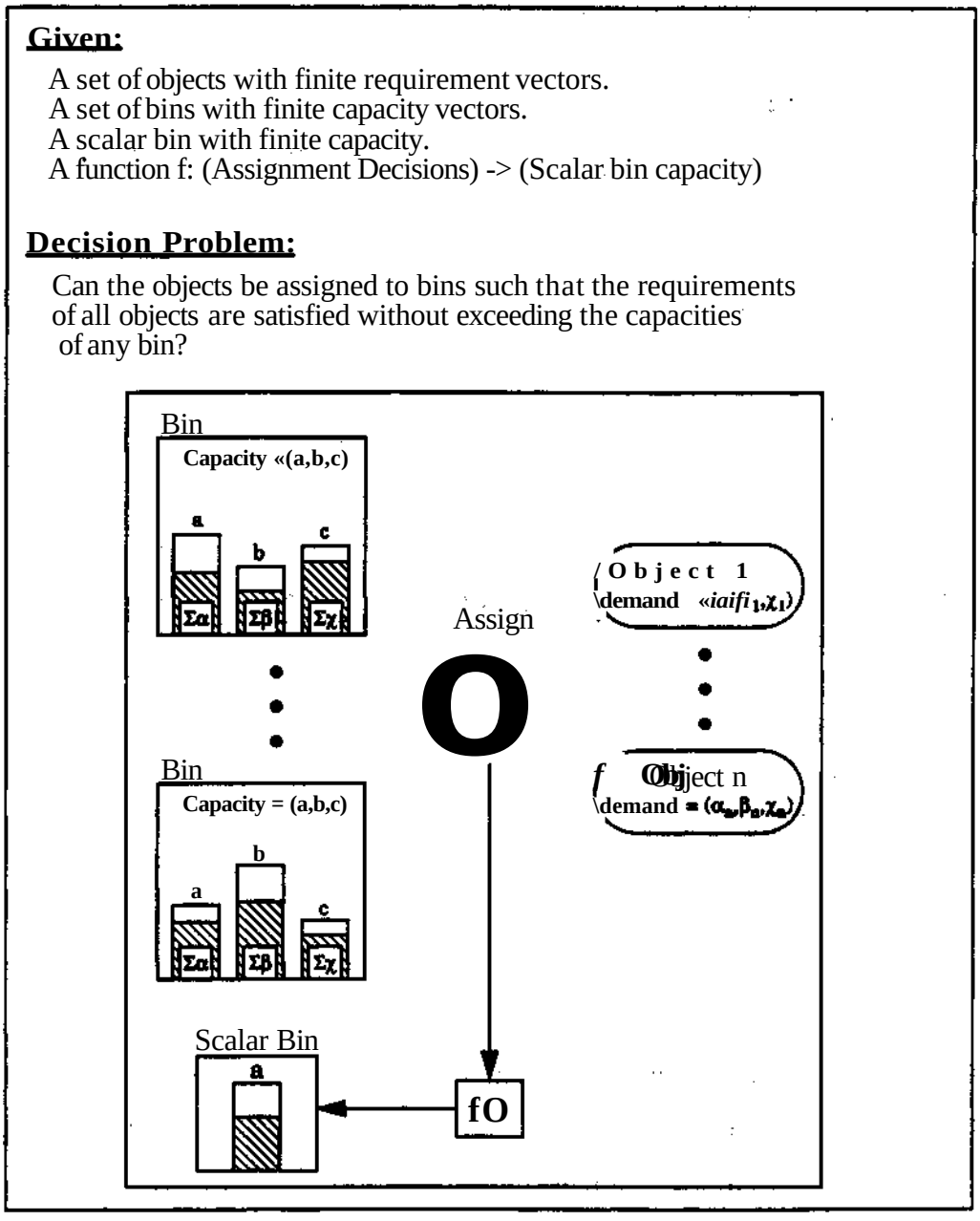


Figure 17
Multi-Dimensional Bin Packing Problem (with Communication)

The multi-dimensional bin packing problems defined in Figures 16 and 17 are unique, meaning that no similar extensions to the bin packing problem have been found in the literature. The Figure 17 problem will be used to represent task allocation. Solution techniques for this problem will be proposed and investigated.

2.2.3 Heuristic Algorithms

Algorithms for finding the optimum solution to the packing problem defined in Figure 17 are intractable. Thus, effective heuristic algorithms were needed. A set of candidate algorithms was created. They were inspired by the classic first- and best-fit heuristic solutions to the bin packing problem, as well as the techniques reported in [4]. All of the algorithms are one-pass, greedy algorithms. Each one chooses an object, one by one, and assigns it to a bin. This continues until all objects are assigned to bins and the packing is complete, or else a set is left of objects that will not fit into any of the remaining bins. In this case, the algorithm fails. Each algorithm is defined by a five character acronym, as defined in Figure 18.

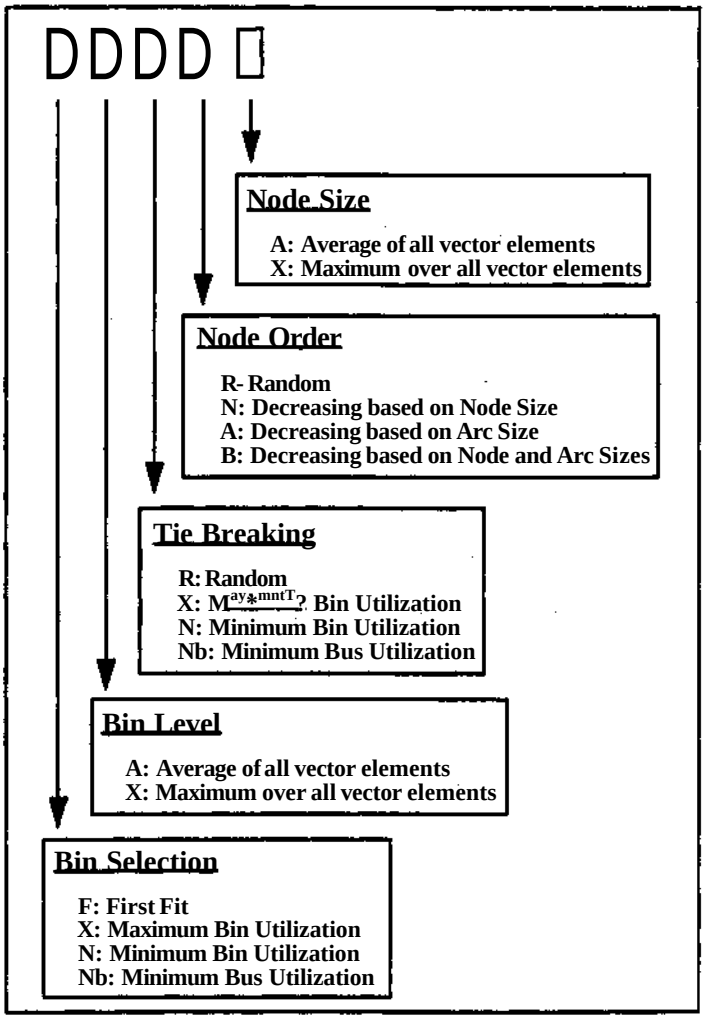


Figure 18
Candidate Heuristic Algorithms

The first character of the acronym specifies the method of bin selection. Four possibilities were considered: choosing the first bin into which the object fits, the bin with the minimum utilization level, the bin with the maximum level, or the bin that minimizes the level of the scalar bin (i.e. bus bandwidth). The second character specifies the method used for determining the utilization level of a bin. Two possibilities were considered: either the average or the maximum level over all vector elements.

If two or more bins are found to be equally good according to the bin selection policy, then tie breaking is invoked. The third character indicates the tie breaking strategy. Four strategies were considered: random tie breaking, tie breaking based on the maximum or minimum bin utilization levels or tie breaking based on the level of the scalar bin (i.e. bus bandwidth).

The order in which the nodes are selected to be assigned to bins has a pronounced effect on the solution. The fourth character indicates the node ordering scheme. Four ordering schemes were considered: random ordering, decreasing order based on *node size*, decreasing order based on *arc size* and decreasing order based on *node* and *arc sizes*. *Node size* was based on the resource requirements of the node (task). *Arc size* was based on the cumulative scalar bin (i.e. bus bandwidth) requirements of the arcs connected to the node in question. Decreasing on *node* and *arc sizes* chooses the node with the maximum size, *node* or *arc*, at each point in the ordering process.

The fifth and last character specified how *node size* was determined. Two possibilities were considered: the average or the maximum utilization requirement across all vector elements.

2.2.4 Experimentation Strategy

A set of experiments was undertaken to gauge the effectiveness of the heuristic algorithms. The packing problem shown in Figure 19 was used for all experimentation. Note that this problem is just a specific instance of the problem type defined in Figure 17. Basically, the problem amounts to packing an input DFG into an arbitrary number of "unit-PEs", which communication over a 1 Mbps CAN bus. Intra-PE communication is free, while inter-PE communication consumes bus bandwidth, based on the CAN bus transmission time model presented previously in Figure 10.

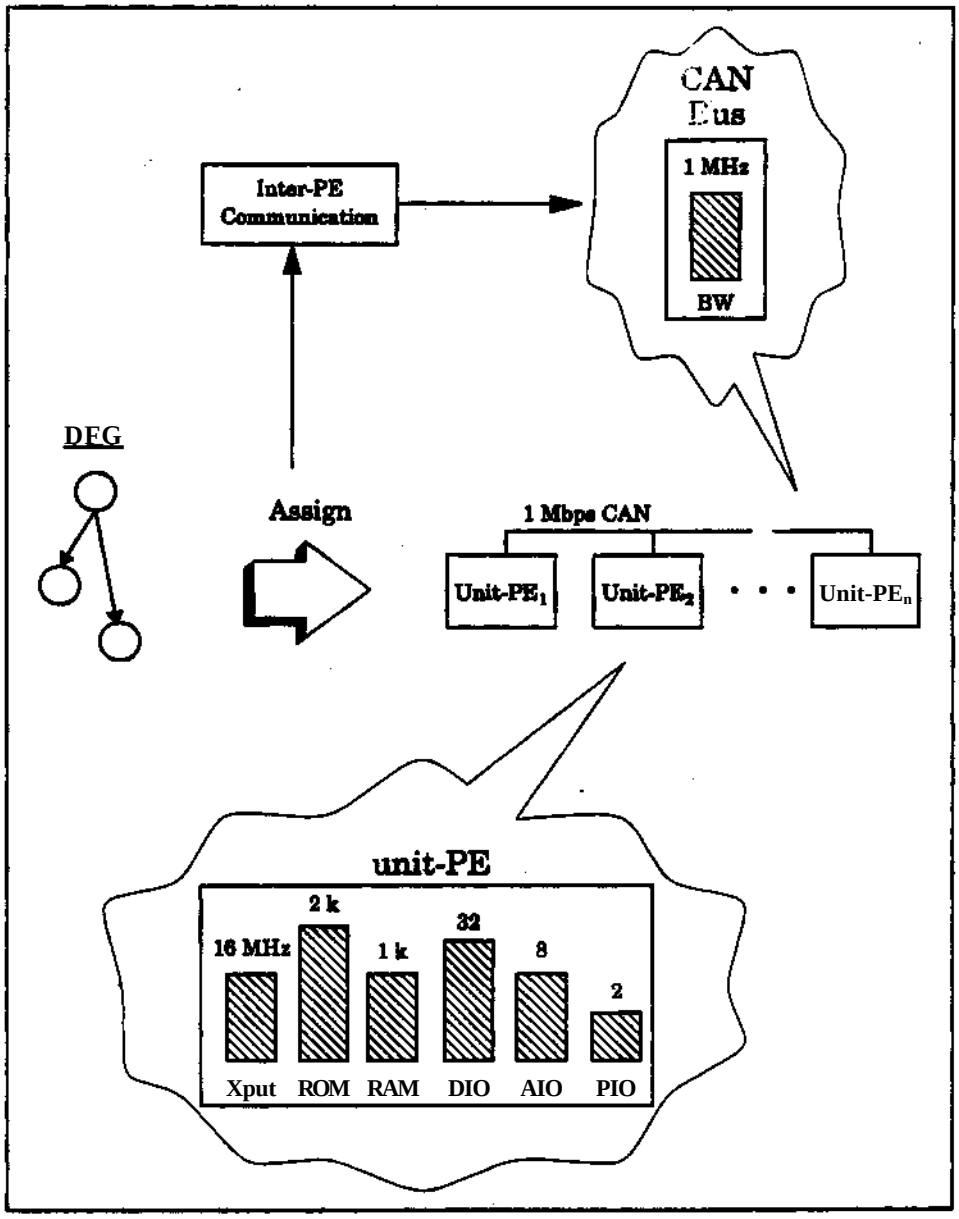


Figure 19
Packing Problem Instance used for Experimentation

Experimentation consisted of four phases. The goal of each phase is summarized in Figure 20.

Phase One:

Investigate the effect of node ordering.

Phase Two:

Investigate the effect of the bin selection policy.

Phase Three:

Investigate the effect of tie breaking.

Phase Four:

Compare the best heuristic algorithm against the optimum algorithms.

Figure 20
Four Experimentation Phases

2.2.5 Inputs

Sixteen DFGs were used as inputs for the experiments. Eleven were randomly generated, three were real and two were contrived. The randomly generated DFGs were produced in an automated way, based on user-supplied probability distributions for the graph variables. The real DFGs were adapted from data obtained from the characterization of an automotive powertrain controller, reported in [1]. They were used without modification, except that the ROM requirement for each task was scaled by (1/10). This was done to allow certain large tasks to fit into the "unit-PEs." The contrived DFGs were hand-crafted.

The graph values for all DFGs, stated as utilization percentages of "unit-FE" capacities, are summarized in Appendix A.

2.2.6 Output Format

When a heuristic algorithm was applied to an input DFG, three metrics were used to gauge its effectiveness:

1. Number of Bins ("unit-PEs") in the Solution.
2. Scalar Bin (bus BW) Utilization Level for the Solution.
3. Run Time for the Algorithm.

Metrics 1 and 2 were plotted on x-y graphs, as shown in Figure 21. The method used to calculate upper and lower bounds on the number of bins is described in Appendix B. The number of bins required for a packing was normalized to the lower bound. All run times were measured on a DECstation 3100 engineering workstation.

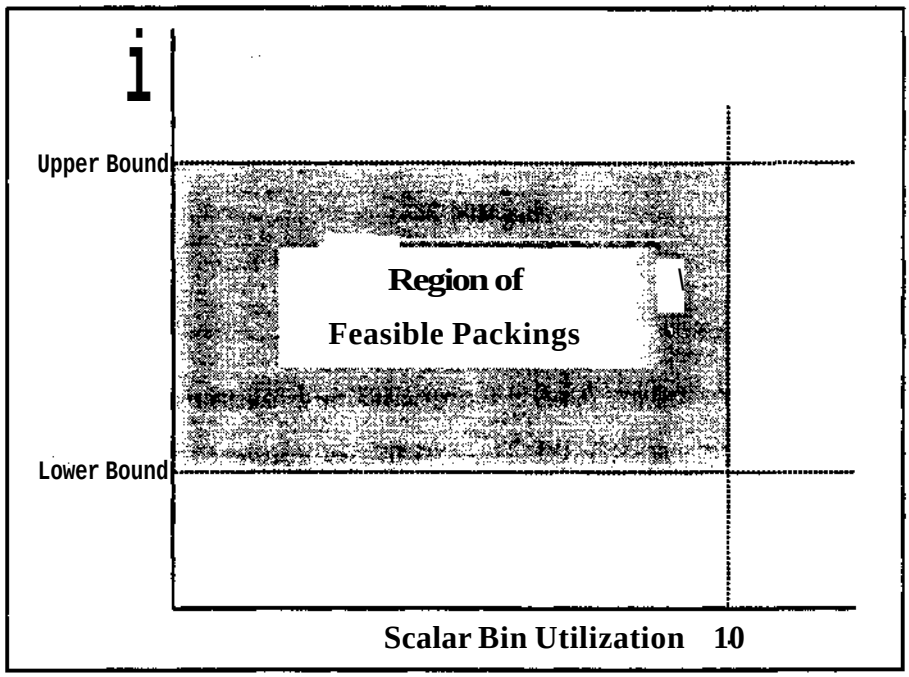


Figure 21
Output Format

2.2.7 Results

Results for all four experimentation phases are summarized in Appendix C. The discussion that follows is based on those results.

The goal of the first phase of experimentation was to determine the effect of node ordering on the heuristic algorithms. This was done by comparing first-fit algorithms with different ordering schemes. Specifically, the following six algorithms were compared:

1. **F-R- (Baseline)**[1]
2. **F-NX**
3. **F-NA**
4. **F-A-**
5. **F-BX**
6. **F-BA**

The results revealed several tilings. First ordering based on node size tended to decrease the number of bins that were needed. Unfortunately, this did not lead to wise utilization of the scalar bin, which in several instances, prevented a feasible solution from being found. Second, ordering based on arc size tended to decrease the scalar bin utilization level. This allowed a feasible solution to be found for all DFGs, however the solutions often required slightly more bins. This was due to poor packing caused by excessive fragmentation. Ordering based on node and arc sizes, however, worked quite well. This scheme appeared to exploit the benefits of the other two (i.e. fewer bins and effective use of the scalar bin) without suffering from their weaknesses (i.e. not finding a solution).

The effect of basing node size on the average or maximum vector element was found to be marginal and inconclusive. Furthermore, there was no significant run time variation across algorithms. This is intuitive, since node ordering only requires the nodes to be sorted once before packing begins.

Summarizing, the results from phase one indicated that node ordering based on decreasing node and arc sizes was the most effective technique, and nodes size based on the maximum vector element was preferred for simplicity. This scheme was used for all subsequent experimentation phases.

1. **As defined previously: (F-R-) = First-fit bin selection and random node ordering.**

The results of phase one are summarized in Figure 22.

Observations:

1. **Ordering on node** size **requires fewer bins but results in poor scalar bin utilization.**
2. **Ordering on arc size uses the scalar bin wisely but requires more bins.**
3. **Ordering on node and arc sizes balances 1 and 2.**
4. **The effect of whether node size is based on the maximum or average vector element is marginal and inconclusive.**
5. **There is no significant run-time variation across algorithms.**

Conclusion:

1. **Only (???BX) algorithms will be considered in subsequent phases.**

Figure 22
Summary of Phase One Results

The goal of the second phase of experimentation was to investigate the effect of the bin selection policy. This was done by comparing the effectiveness of the following seven algorithms:

1. F-BX (baseline)[2]
2. XXRBX
3. XARBX
4. XNRBX
5. NXRBX
6. **NARBX**
7. Nb-RBX

2. As defined previously: (F-BX) = First-fit bin selection, decreasing order based on node and arc sizes and node size based on the maximum vector element.

The results revealed several things. First, selecting the least utilized bin performed worse than the baseline, first-fit. This is intuitive. Selecting the least utilized bin is a poor packing heuristic, since it never tries to completely fill a started bin. This leads to excessive fragmentation and thus more bins compared to first-fit. Second, selecting the most utilized bin performed no better than first-fit. This was a surprising result. The implication is that there is no analog to the *best-fit decreasing* algorithm, which has been shown to be so effective for the classic bin packing problem. The reason for this is also intuitive: there is no notion of what constitutes a *best fit* for a multi-dimensional object being placed in a multi-dimensional bin. A bin assignment that is the *best fit* for a particular dimension may actually impair the packing across other dimensions. To understand this, consider applying the (XXRBX) algorithm to the 1- and 2-dimensional examples shown in Figure 23.

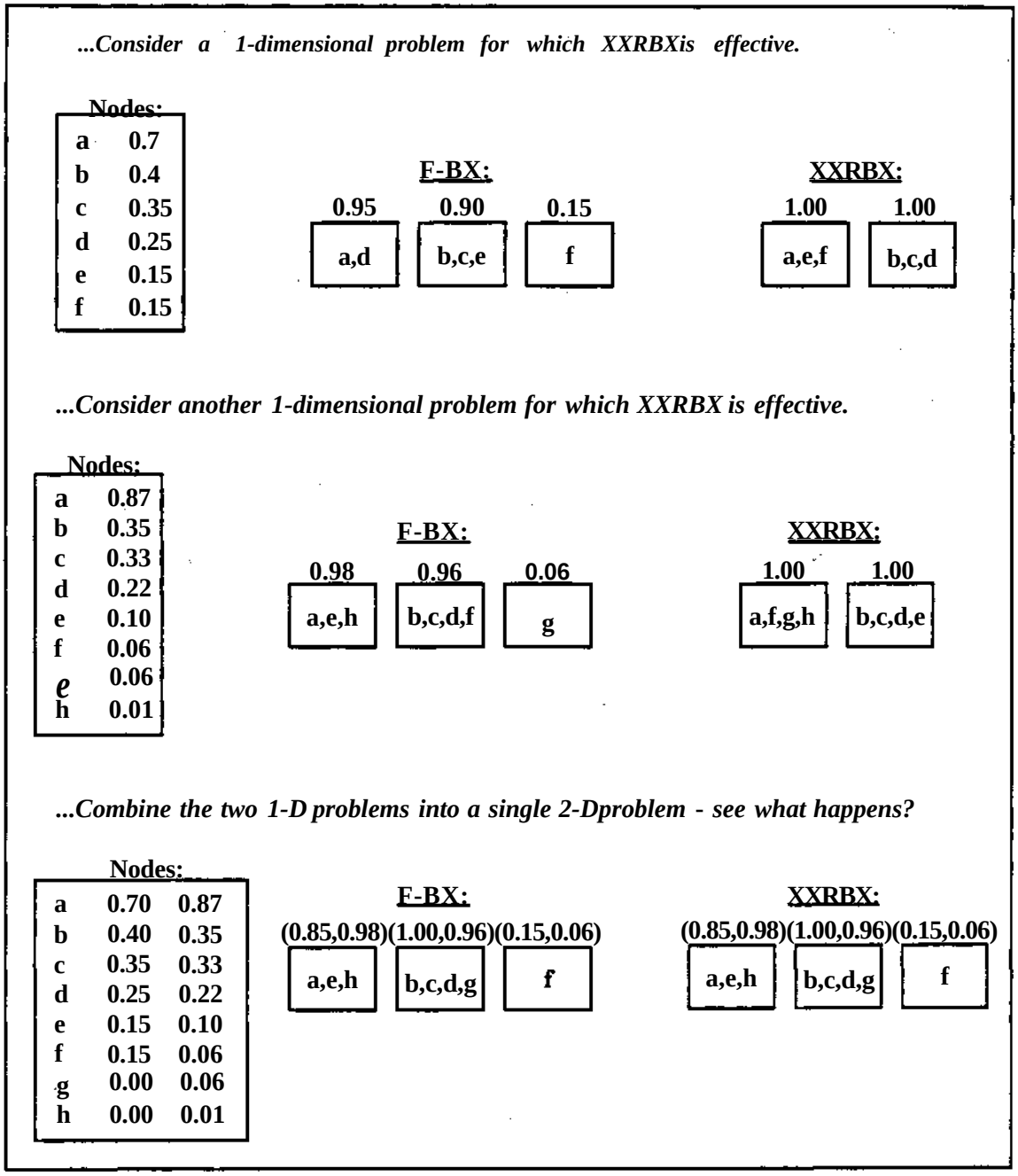


Figure 23
Example showing the ineffectiveness of multi-dimensional best-fit algorithms

As the example indicates, interactions across dimensions nullify the effectiveness of best-fit packing heuristics.

Selecting the bin that minimizes utilization of the scalar bin (i.e. bus) was found to be superior to first-fit. On average, this technique required no more or less bins than first-fit (for the reason given above), but it did return packings with significantly lower scalar bin utilization levels. In fact, this method is a wise choice for task allocation since, coupled with the node ordering scheme found in phase one, it leads to a natural and dynamic clustering of heavily communicating tasks which conserves bus bandwidth.

As in phase one, basing node size on the average or maximum vector element had a marginal and inconclusive effect. Therefore the maximum vector element was preferred for simplicity.

The baseline algorithm, first-fit, did have a measurable run time advantage over the other algorithms. However, based on its superior performance, selecting the bin that minimizes usage of the scalar bin was preferred and this was the technique that was used in all subsequent experimentation phases.

The results of phase two are summarized in Figure 24.

Observations:

1. On Average, selecting the least-utilized bin requires more bins than first fit with a comparable scalar bin utilization level.
2. On Average, selecting the most-utilized bin requires no fewer bins than first fit with a comparable scalar bin utilization level.
3. On Average, selecting the bin that minimizes scalar bin utilization produces lower scalar bin levels than first-fit with a comparable number of bins.
4. The effect of whether bin level is based on the maximum or average vector element is marginal and inconclusive.
5. (F-BX) has a run-time advantage over all other algorithms.
6. The run-times of all other algorithms are comparable.

Conclusion:

1. Only (Nb??BX) algorithms will be considered in subsequent phases.

Figure 24
Summary of Phase Two Results

The goal of phase three was to determine the effect of employing a tie breaking strategy. To accomplish this, the following two algorithms were compared:

1. Nb-RBX (baseline)[3]
2. NbXXBX

As the results indicate, there was no measurable benefit for tie breaking. Again, the reason that the tie-breaking schemes were ineffective is due to inter-dimensional interactions that render node utilization-based bin selections no better than random selection.

3. As Defined Previously: (Nb-RBX) = Selection of bin that minimizes scalar bin utilization, decreasing order based on node and arc sizes, node size based on the maximum vector element and random tie breaking.

The goal of the fourth and final phase of experimentation was to provide a consistency check, by comparing the solution returned by the best heuristic algorithm (Nb-RBX) against the optimum solutions. The solution requiring the fewest bins was found by an algorithm named QPT-BINS. This algorithm did an exhaustive search over all possible assignments of objects to bins, beginning with one and incrementing the number of bins until a solution was found. Likewise, another exhaustive search algorithm, OPT-BUS, was used to find the solution that least utilized the scalar bin. Thus, three algorithms were compared:

1. **Nb-RBX (heuristic)**
2. **OPT-BINS**
3. **OPT-BUS**

As the results indicate, the heuristic algorithm, (Nb-RBX), perfonns well compared to the optimum algorithms. It also has a significant run time advantage. In general, note that OPT-BINS and OPT-BUS are both exponential algorithms. They have time complexities of O(Processors^{Nodes}) and 0(2^^) respectively, making them computationally intractable for the vast majority of the problem space.

The results of phases three and four are summarized in Figure 25.

Observations:

1. Tie-breaking has no measurable effect.
2. (Nb-RBX) returns near-optimal solutions.

Conclusion:

1. (Nb-RBX) is the most effective heuristic algorithm.

Figure 25
Summary of Phase Three and Four Results

2,2.8 Conclusion

A unique extension to the bin packing problem, *multi-dimensional bin packing*, was derived which is isomorphic to the task allocation problem. Heuristic algorithms were proposed to solve this problem. The performance of the algorithms was compared on sixteen input DFGs with respect to three figures of merit. The (Nb-RBX) heuristic algorithm was found to be the most effective. Furthermore, this algorithm was shown to produce near-optimal results with a significant run time advantage over the optimal search algorithms. The (Nb-RBX) algorithm will therefore be used as a fast and efficient technique for performing task allocation.

2.3 Solution Techniques

2.3.1 Goal

The goal of this section is to develop and evaluate solution techniques, based on the (Nb-RBX) packing algorithm, for the coupled design problems defined in Section 1. Recall that solution of the coupled design problems amounts to finding a set of processor specifications and an assignment of tasks to processors that satisfy the feasibility constraints and

optimize the objective function described in Section 1. The major results of this section are the algorithms that are developed.

2.3.2 Overview of Solution Techniques

The previous section developed an efficient heuristic packing algorithm for performing task allocation. Solutions to the coupled design problems were obtained by combining this algorithm with incremental or successive processor specification strategies. Specifically, four solution techniques have been developed. They are listed in Figure 26.

Algorithm	#ofPEs	Processor Specification	Ifeflk Allocation	Description
SW	Dynamic	Constraint-Driven	Nb-RBX	Shrink-Wrapping
ENUM	Static	Search	Nb-RBX	Exhaustive Search
SA	Static	Simulated Annealing	NUIBX	Simulated Annealing
DA	Dynamic	Incremental Refinement	Nb-RBX	Design Advisor

Figure 26
Four Solution Techniques for Problem Representation One

As the figure indicates, all techniques use the same packing algorithm (Nb-RBX) to perform task allocation. Two of the techniques, SW and DA, dynamically determine how many processors to use; the others, ENUM and SA, require this as an input. SW and ENUM were developed as strawman approaches while the SA and DA algorithms are the main results of this section. All of the algorithms were implemented in C++ using AT&T compiler version 3.0.

2.3.3 Shrink-Wrapping Algorithm (SW)

A strawman algorithm was created based on the notion of *shrink-wrappitig.* It works as follows. First, an *upper bound* on the number of processors is obtained using the method derived in Appendix B. Next, the processors are specified by choosing the *maximum* possible value for each hardware design variable. The (Nb-RBX) algorithm is then invoked which attempts to pack the DFG graph into the processors.

If the packing algorithm returns no solution, then one of two situations has occurred. Either no solution exists for the given DFG and hardware architecture, or else a solution exists that the heuristic packing algorithm failed to find. Note, however, that if the latter case occurs, it is due entirely to the inefficiencies of the packing algorithm since the hardware architecture was set to the maximum number of maximally-specified processors.

If the packing algorithm does return a valid solution, the hardware is then *shrink-wrapped* to match the task allocation that was found. *Shrink-wrapping* of the hardware implies two things. First, any unused processors in the design are removed. Second, all of the design variables in the used processors are set to the smallest values that do not violate any of the feasibility constraints. In order for shrink-wrapping to be computationally tractable, the hardware design variables and the feasibility constraints based on them must be mutually independent. This is true for the variables and constraints that were presented in Section 1. In general, this is a reasonable assumption.

The solution obtained with the SW algorithm is driven by the task allocation returned by (Nb-RBX). The number of processors in the design is determined dynamically as a by-product of the task assignment decisions that were made. Shrink-wrapping returns the. lowest complexity hardware architecture that will support this task allocation. As such, no attempt is made at optimizing the objective function. This algorithm is robust, however, in the sense that its only limitations on finding a solution when one exists are those inherent in the (Nb-RBX) packing algorithm. Also, since task allocation (i.e. packing) and

processor specification are both performed only once, this algorithm establishes a lower bound on run time.

A flowchart of the SW algorithm is shown in Appendix D.

2.3.4 Exhaustive Search Algorithm (ENUM)

An exhaustive search algorithm, ENUM, was created as a strawman. First, the number of processors to be used in the design is input to the algorithm. Next, the algorithm cycles through all possible specification combinations for the given number of processors. For each specification combination, the (Nb-RBX) algorithm is invoked which attempts to pack the DFG into the processors.

If the (Nb-RBX) algorithm returns no solution, then one of two situations has occurred. Either no solution actually exists for the given DFG and processor specifications, or else a solution exists that the packing algorithm failed to find. Once again, if the latter case occurs, it is due entirely to the inefficiencies of the packing algorithm.

If the (Nb-RBX) algorithm returns a solution, die objective function is invoked to determine its relative merit. At each step, a copy is maintained of the best, feasible solution seen thus far. Upon completion, the best observed solution is returned by the algorithm.

Unlike the SW algorithm, ENUM actually attempts to optimize the objective function. Because of the exhaustive search, however, it has exponential time complexity. This results in long run times, even for problems of modest size, like the SW algorithm, ENUM is also robust, in the sense that its only limitations on finding an existing solution are those inherent in the packing algorithm. Furthermore, since all feasible processor combinations are compared, this algorithm establishes an upper bound on the degree of optimization obtainable with a (Nb-RBX)-based solution technique.

A flowchart of the ENUM algorithm is shown in Appendix D.

2.3.5 Simulated Annealing Algorithm (SA)

An algorithm, SA, was created that uses *simulated annealing* ([9],[11]) to optimize processor specifications and the (Nb-RBX) algorithm to perform task allocation. The execution of these two algorithms is *interlaced.*

As the simulated annealing algorithm progresses, a *move* function is executed that randomly perturbs the processor specifications. After a random move is attempted, the (Nb-RBX) algorithm is invoked which tries to pack the new processors. If a feasible packing was found, the processors are shrink-wrapped. The simulated annealing *costfunction* is then used to rate the new solution attempt. This cost function consists of two terms. The first term represents the design objective function. The second term represents a penalty that is incurred if the (Nb-RBX) algorithm failed to return a solution. The magnitude of the penalty is a function of how far the (Nb-RBX) algorithm progressed before failing. By trying to optimize its cost function, the simulated annealing algorithm actually tries to return a solution that is feasible (i.e. packable) and that optimizes the design objective function.

Since simulated annealing is a general solution strategy, three portions of the annealing algorithm were tailored to the problem: the *move* function, the *cost* function and the *annealing schedule,* as described below.

2.3.5.2 *Move Function*

***A move* function was needed that creates a new set of processor specifications by randomly perturbing the current ones. The *neighborhood* of the current state is defined as the set of states that are reachable within one move. A requirement of the move function is that it must be able to traverse the entire design space through an arbitrary sequence of moves begun from any starting state. Furthermore, the neighborhood generated by a**

move function should be large enough to allow large hops through the design space in relatively few moves.

Two candidates for the move function were investigated, as shown in Figure 27.

Function 1:	**Function 2:**
1. Randomly pick (pe P). **2. Randomly pick (ve DV).** **3. Randomly increment or decrement v in p.**	**1. Randomly pick (pe P).** **2. Randomly pick (ve DV).** **3. Randomly pick (ce VS_v).** **4. Set v in p to c.**
P = {Set of Processors} **DV = (Set of Design Variables}** **VS_i = {Value Set for Design Variable i|**	

Figure 27
Move Functions

The first function takes a randomly selected design variable from a randomly selected processor and randomly increments or decrements it The second function takes a randomly selected design variable from a randomly selected processor and sets it to a randomly selected element from the set of all possible values.

The premise behind the first function was that it forced incremental change which was thought to improve optimization. The second function, however, had a larger neighborhood. Initial experimentation was conducted and the second function was fouiul to universally outperform the first. Accordingly, it was adopted as the move function.

2.3.5.2 Cost Function

The cost function rates the worth of each candidate solution. The function that was developed is shown in Figure 28.

$$\text{Cost} = \text{Log} \sum_{i=1}^{n} \text{Area}_i + r\,(15 \times UT)$$

n = # of processors
UT = # of unassigned tasks
Area; = Die Size of Processor i

Figure 28
Cost Function

The cost function consists of two terms. The first term is identical to the design objective function being optimized. The second term represents a penalty that is incurred if the (Nb-RBX) algorithm fails to find a feasible packing. The magnitude of the penalty depends on how far the (Nb-RBX) algorithm progressed before failing. Each task that has not yet been assigned to a bin when the packing algorithm fails contributes a fixed amount to the total penalty. Thus, if the (Nb-RBX) algorithm can pack most of the DFG into the processors, less of a penalty is incurred, resulting in a lower cost. The fixed amount incurred by each unpacked task was determined empirically through experimentation. It was found that the penalty term must be large enough to distinguish an infeasible solution from potentially costly feasible ones. The idea of incorporating penalty terms into the cost function was adapted from [12].

2.3.5.3 Annealing Schedule

The annealing schedule determines when and how the algorithm begins, accepts randomly generated solutions and terminates. As such, it is crucial to the operation of the algorithm. An improperly designed annealing schedule can result in poor optimization and/or excessive run times. Four specific issues must be solved by the annealing schedule:

1. **Starting Temperature**

2. **Equilibrium Detection**
3. **Temperature Decrement Function**
4. **Termination Condition**

Each of these items is discussed in detail. The approaches used were adapted from [6], [8], [9] and [11].

The starting temperature must be hot enough to *melt* the system. When the system is *melted,* a randomly generated solution with higher cost is accepted as often, on average, as one with lower cost. This implies that the starting temperature should be reasonably higher than the standard deviation of the cost function. Accordingly, the starting temperature defined in Figure 29 was used [8].

T° « 20 x a
a = Standard Deviation of Cost Function

Figure 29
Starting Temperature

Before the starting temperature can be calculated, an estimation of the standard deviation of the cost function is needed. This is obtained by collecting cost statistics during a random walk through the design space. The length of this random walk should be dependent on problem size. Since the actual problem sizes are exponential, however, a heuristic

was needed that increases with problem size in a bounded way. The heuristic that was used is shown in Figure 30.

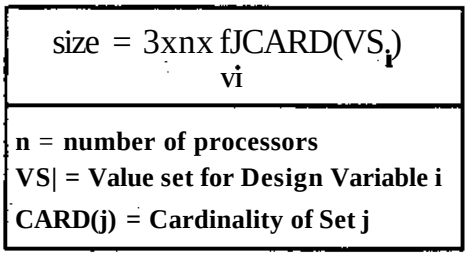


Figure 30
Problem Size Heuristic

When the simulated annealing algorithm is running, transition to a lower temperature is allowed once equilibrium is established at the current temperature. Therefore, a method of detecting equilibrium is necessary. The method used was adapted from [8]. It is based on the observation that once equilibrium is established, the ratio of the number of accepted states with costs that fall within a defined probability interval of the average to all accepted states approaches a constant value. If the probability distribution for cost is assumed to be normal, which is a fair assumption at high temperatures, then a target ratio can be established as shown in Figure 31.

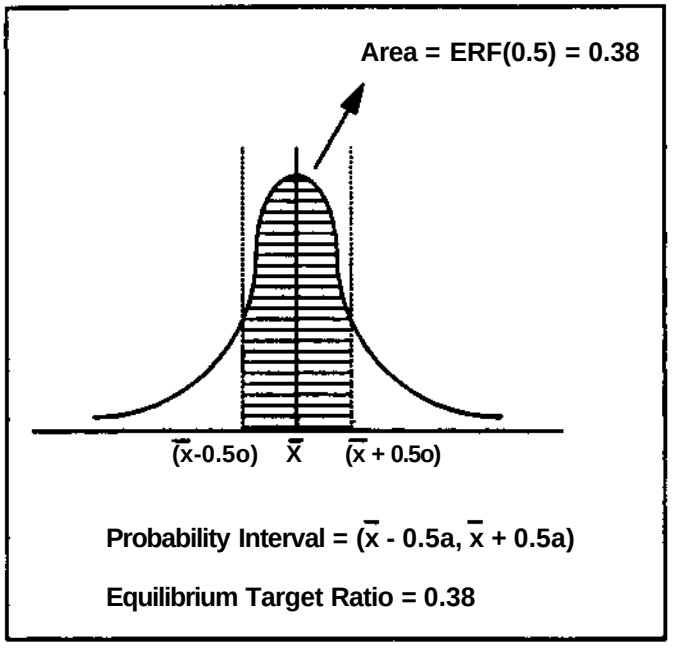


Figure 31
Equilibrium Target Ratio

Before transition to a lower temperature is allowed, the target ratio must be met over a sample size equal to the heuristic problem size defined in Figure 30. Algorithmically, this **is equivalent to requiring an *inside-interval threshold* to be met before an *outside-interval tolerance* is exceeded. The values for the *inside-interval threshold* and *outside-interval tolerance,*** based on the target ratio defined in Figure 31, are given in Figure 32.

Inside-Interval Threshold = (0.38) x size

Outside-interval Tolerance = (1 - 0.38) x size

Figure 32
Inside-Interval Threshold and Outside-interval Tolerance Values

To determine whether or not the *inside-interval threshold* has been met or the *outside-interval tolerance* has been exceeded, it is necessary to know whether or not the cost of each randomly generated solution falls within the defined probability interval around the average cost. Furthermore, this implies that a running calculation of cost statistics must be maintained. These statistics are calculated over a *window* of the most recently accepted costs. The size of the window was determined empirically and set to ten.

As temperature decreases, the number of states accepted at any given temperature also decreases. Thus, the target *inside-interval threshold* may never be reached at low temperatures, trapping the algorithm at an above freezing temperature. To avoid this situation, a maximiun niunber of moves at any given temperature was established. If this number is reached before equilibrium is detected, then the temperature is automatically decremented. The maximum number of moves was set to an empirically determined function of heuristic problem size; it is defined in Figure 33.

Max Moves = 7 x (Outside-Interval Tolerance)

Figure 33
Maximum Move Criterion

When the annealing temperature is decreased, the *temperature decrementfunction* is invoked to determine what the new temperature should be. The *temperature decrement function* that was used is shown in Figure 34 [8].

$$T_{EXP} = T \times e^{X} P \left(\sigma^2 \_ \right)$$

$$r = \begin{cases} 0.1T & (T_{EXp} < 0.1T) \\ T_{EXP} & (MT < T_{BXp} < 0JT) \\ 0.5T & (T_{EXp} > 05T) \end{cases}$$

Figure 34
TemperatureDecrementFunction

This function performs exponential temperature reductions. Reductions were bounded, however, to fall within 10% and 50% of the current temperature. This was done to avoid quenching when little cost variation is encountered, and extremely long run times when cost fluctuates wildly.

The last aspect of the annealing schedule to be considered is the termination condition. The algorithm should terminate when no lower cost solutions are found and the temperature has dropped to a level where no higher cost solutions are accepted. A simple method was used to detect this; the algorithm terminates when no solution has been accepted for four consecutive temperatures.

A flowchart of the SA algorithm is shown in Appendix D.

2.3.6 Incremental Design Advisor Algorithm (DA)

An algorithm, DA, was created that is driven by an *incremental design advisor*. The algorithm works as follows. First, a *lower bound* on the number of processors is obtained using the method derived in Appendix B. Next, the processors are specified by choosing the *minimum* possible value for each hardware design variable. The (Nb-RBX) algorithm is then invoked which attempts to pack the DFG into the processors.

If the (Nb-RBX) algorithm should happen to fail, meaning a task is encountered that will not fit into any of the existing bins, then the *incremental design advisor* is invoked. The design advisor examines the current hardware configuration and partial packing state, and then makes a hardware specification change. When the advisor completes, the (Nb-RBX) algorithm is re-invoked and continues from where it left off. In this way, the algorithm returns a solution that is obtained incrementally from a series of packing attempts and specification refinements.

The success of this approach hinges on two assumptions. First/note that when the design advisor is invoked, it suggests the best possible hardware modification based on an incomplete, *local* view of the design. Thus, like MICON [3], the solution is based on a *series* of *locally optimum* design decisions. Therefore, like MICON, the algorithm assumes that a solution obtained in this manner is a fair approximation of the optimum solution. Second, note that the starting place for this algorithm is the *minimum complexity* hardware configuration. Each time the design advisor is invoked, the *complexity* of the hardware specification is incrementally *increased.* Since the design objective function given in Figure 13 increases monotonically with hardware complexity, the algorithm essentially begins with the most desirable hardware configuration and then moves to an incrementally less desirable configuration every time the design advisor is invoked. Thus, for optimization to occur, the algorithm assumes that a monotonic relationship exists between the design objective function and hardware complexity. This is a fair assumption for most, but not all objective functions.

The operation of the design advisor is described next. When the design advisor is invoked, it is given a set of specified, partially packed processors and a task that will not fit into any of them. Based on this information, a set of candidate hardware changes is created. This is done by sequentially forcing the task into each existing processor and then *shrink-wrapping* the hardware to meet the feasibility constraints of the new partial

packing. Additionally, another candidate hardware change is created by adding a new processor, assigning the task to it, and then *shrink-wrapping* this new configuration. This is illustrated in

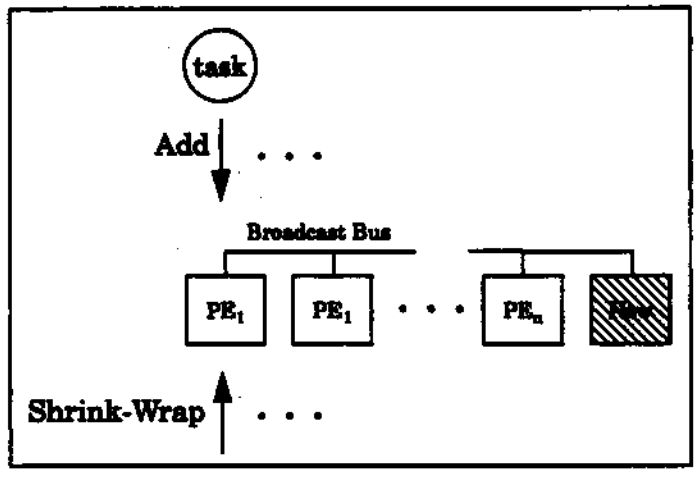


Figure 35
Candidate Hardware Changes

Once this set of candidate hardware changes has been created, the relative merit of the alternatives must be computed and compared. Relative merit was based on two things:

1. **Relative change in the objective function.**
2. **Relative change in the bus utilization level.**

For the candidate hardware changes involving the existing processors, the objective function change was defined as the percentage difference between the old and new configurations. However, for the candidate change containing the new processor, things were handled differently. In general, adding a new processor tends to be more costly than modifying an existing one. Thus, if only relative cost differences were considered, modification of an existing processor would almost universally be preferred to adding a new one. Note, however, that this situation is pessimistic. It assumes that the task being assigned to the new processor must bear the entire overhead burden associated with it.

This is generally not true, however, because if the new processor is subsequently used by the packing algorithm its overhead will be amortized across additional tasks. To account for this, an attempt was made to amortize the overhead burden of the new processor when computing the objective function change. This was done by multiplying the processor overhead by the maximum of the reciprocal of the unpacked tasks and the maximum utilization level across all dimensions for the task being assigned to it. The method used to calculate objective function changes is summarized in Figure 36.

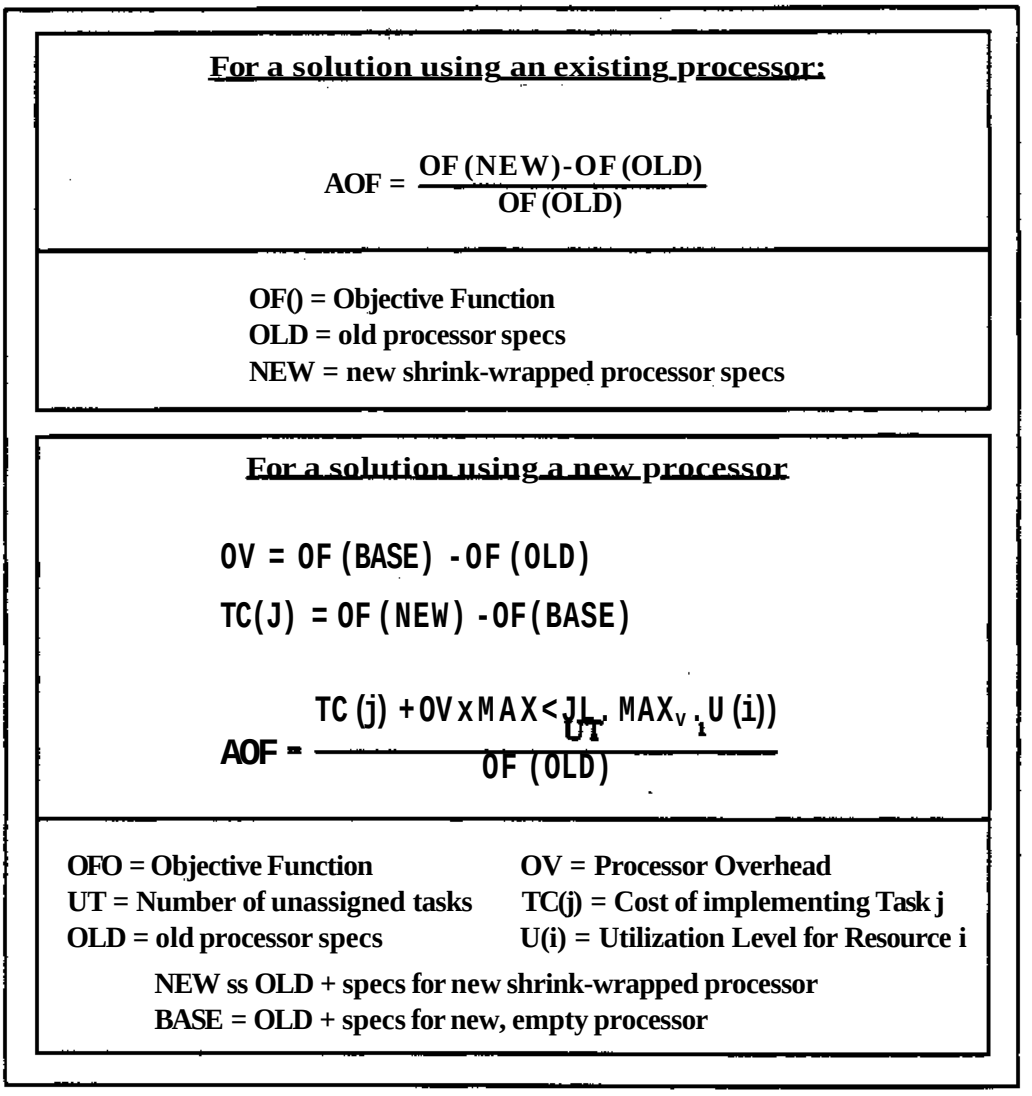


Figure 36

Calculating the objective function change for candidate solutions

To quantify the relative change in bus utilization, the notion *of penalty points* (from MICON, [3]) was used. This is shown in Figure 37.

$$\Delta BW = \frac{u_{<OLD>}}{BW} \times \frac{u_{<NEW>} \sim u_{<OLD>}}{BW}$$

BW = Bus Bandwidth
U(i) = Bus Bandwidth Utilization for solution i
OLD = Old processor specs
NEW = New processor specs

Figure 37
Bandwidth Change for Candidate Solutions

When computing the relative merit of a hardware change, the changes in objective function and bus utilization must be properly balanced. Favoring the objective function leads to greater optimization. However, if bus bandwidth is not managed wisely, then the algorithm may fail after several invocations of the design advisor. Therefore, a method was needed to quantitatively weight these terms. After several candidate weighting schemes were tried, the penalty function defined in Figure 38 was found to work very well.

Penalty » (1+ABW) [l+exp(k(l+AOF))]

DOF = Objective Function Change
DOF = Bus Bandwidth Change

Figure 38
DA Penalty Function

Essentially, the penalty function weights the objective function and bus bandwidth changes. The weighting is controlled by a unit-less parameter, k. When (k<l), the exponential term vanishes and the penalty function is dominated by bus bandwidth. Conversely, when (k>l), the exponential term increases and the objective function dominates.

Conceivably, any value of k in the range (-1 < k < 1) could have been used. Experimentation showed, however, that communication-intensive inputs required small values of k for robustness, while non-communication-intensive inputs needed higher k values for good optimization. To solve this dilemma, an iterative approach was taken. Internally, the DA algorithm is invoked four times, with k=[-3,-l,l,3]. The best solution from the four iterations is returned.

Flowcharts for the DA algorithm and die design advisor are shown in Appendix D.

2.3.7 Experimentation

Experimentation was undertaken to determine the effectiveness of the algorithms. The same DFGs, summarized in Appendix A, were used as inputs. The objective function and set of feasibility constraints presented in Section 1 were also used.

Once again, the goal of the algorithms was to arrive at a set of processor specifications and an assignment of tasks to processors that are feasible and that optimize the objective function. Accordingly, two metrics were used to gauge algorithm performance:

1. **Objective Function value of the returned solution.**
2. **Run-time of the algorithm.**

Since two of the algorithms dynamically determine die number of processors while the other two require this as a *static* input, a fair way of comparing their performance was needed. A method for comparing the static ENUM and SA algorithms with the dynamic SW algorithm was developed that hinged on the following two properties of the SW algorithm:

1. **SW is more *robust* than the static algorithms, ENUM or SA.**
2. **A solution returned by SW will always require *no more* processors than a solution that can be found using either static algorithm, ENUM or SA.**

Based on this, the SW algorithm was first applied to each DFG. Then the static algorithms, ENUM and SA were applied using the same number of processors found in the SW solutions. This allowed direct and fair comparisons to be made.

Similarly, a method was needed for comparing the performance of the design advisor. Thus, a static version of the DA algorithm, called DA_STAT, was created. A comparison between the design advisor approach and the other three algorithms was obtained by applying the DA_STAT algorithm to the DFGs using the same number of processors. Next, by comparing the results of DAJJTAT with the original, dynamic DA algorithm, the effects of the processor creation portion of the design advisor were isolated so that its performance could be judged in relation to the baseline (static) approach.

2.3.8 Results

Results for the algorithms are summarized in Appendix E. The discussion that follows is based on these results.

The ENUM and SA algorithms were substantially slower than SW, DAJ5TAT and DA. Because ENUM and SA are both search algorithms (exhaustive and probabilistic, respectively), they have greater time complexities than the simpler, heuristic-based approaches. Accordingly, they had substantially longer run times for all of the inputs.

For small problems (number of processors < 3), SA was slower than ENUM but for larger problems SA outstripped ENUM by many orders of magnitude. This was because time complexity grows exponentially with problem size for ENUM but not for SA. In fact, this is the primary motivation for using a simulated annealing algorithm over exhaustive search.

When solutions were obtained for both SA and ENUM the results were identical. This, coupled with the run time findings above, verifies that the simulated annealing algorithm was properly designed and performs good optimization.

The performance of the DA and DA_STAT algorithms was interesting. First, both algorithms always returned solutions that were at least as good (defined by the objective function) as those found with SW. This implies some degree of optimization, validating the underlying design advisor heuristics. In some cases, however, these algorithms actually returned *better* solutions than ENUM and SA. This was surprising. Since ENUM performs an exhaustive search over all processor specifications, it seemed unlikely that a better solution could be obtained by an algorithm using the same task allocation (packing) technique. The reason that this happened, however, is because DA and DA_STAT use an *incremental* design technique. This means that processor specifications are refined *while* packing is done. Because packing decisions are influenced by the current set of processor specifications, refining the hardware on the fly can change task assignment decisions. Thus, a solution can evolve incrementally that, once completed, is feasible but not packable. This type of solution can be found by the incremental approaches (DA and DA_STAT), but since it is not packable, it would be skipped over by a search algorithm that was mistakenly led to believe the solution was infeasible.

Comparing the DA and DA_STAT results, the performance of the processor creation heuristic can be determined. As the results indicate, in most instances DA finds a solution that is at least as good as DA_STAT's, with a slight increase in run time. In three cases, however, the results found with DA_STAT were marginally better. In all of these cases, however, the number of processors used by DA_STAT and DA were identical. As a whole, this indicates that the processor creation heuristic generally acts to optimize the objective function. But, since its decisions are only locally optimum, they are not always perfect.

A summary of the experimentation results is given in Figure 39.

Observations:

1. ENUM and SA have longer run-times than SW and DA.
2. ENUM run-times grow exponentially with problem size.
3. Cost of ENUM, SA and DA solutions are at least as good as SW solution.
4. ENUM and SA return identical solutions.
5. Cost of DA solutions are *comparable* to ENUM and SA solutions.

Conclusion;

1. DA and SA are both effective solution techniques.
2. A trade-off between run-time and solution quality exists for DA and SA.

Figure 39
Summary of Experimentation Results

2.3.9 Conclusion

Four algorithms were developed, based on the packing paradigm, to solve the coupled design problems: ENUM, SA, DA and SW. All of these algorithms use the heuristic packing algorithm, (Nb-RBX), for task allocation.

As the results indicate, both the SA and DA algorithms perform well. The DA algorithm has a significant run time advantage over SA. The SA algorithm, however, may return a marginally better solution in some, but not all cases. Thus, a trade-off must be considered by the designer when choosing from these two approaches.

2.4 Summary

This section has introduced a packing-based representation of the coupled design problems. Furthermore, a multi-dimensional extension to the bin packing problem was defined that is isomorphic to the task allocation problem. Next, heuristic methods of solving this packing problem were investigated. This resulted in the discovery of an effective algorithm, (Nb-RBX), that can be used to solve the task allocation problem.

Four approaches for solving the coupled design problems were proposed that utilize the (Nb-RBX) algorithm. Two of these, SA and DA, were found to be effective solution techniques. Furthermore, a trade-off between run time and solution quality was found to exists between the two approaches.

3 Representation Two: Graph Partitioning-Based

3.1 Problem Representation

An arbitrary assignment of tasks to processors dictates the minimum complexity processor specifications that are needed to *feasibly* support the assignment. This leads naturally to a *graph partitioning-based* representation of the problem defined in Section 1.

First, realize that there is not a one-to-one correspondence between n-way partitions of the DFG and n-processor assignments. Consider the DFG and 2-processor assignment shown in Figure 40. The assignment shown in the graph does not correspond to any 2-way partition of the DFG.

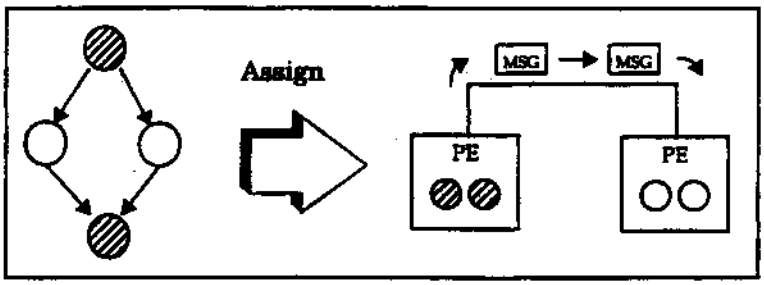


Figure 40
A DFG and a 2-processor assignment

Thus, for a *graph partitioning-based* representation, a modification is needed to obtain this one-to-one correspondence. This is achieved by inserting zero-weighted arcs between every pair of non-communicating tasks in the DFG. Figure 41 shows the example of Figure 40 after the zero-weighted arcs were added. Now, the 2-processor assignment corresponds to a specific 2-way partition of the DFG, as indicated in the figure.

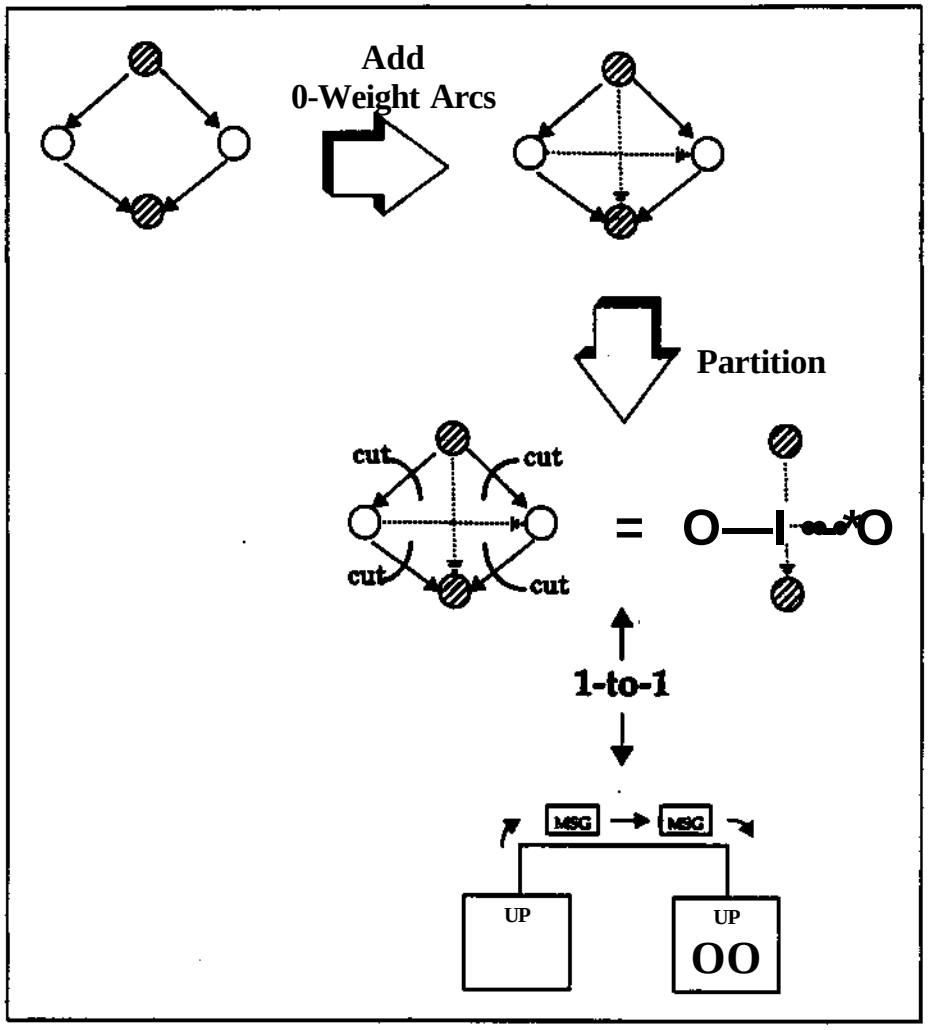


Figure 41
A modified DFG and 2-processor assignment

Any n-way partition of the modified DFG defines an n-processor task assignment. After partitioning, the cumulative requirements of the nodes in each disjoint sub-graph *specify* the minimum complexity, feasible processor needed to implement them. In fact/the processor specifications can be obtained by creating n new processors, assigning tasks to the processors based on the DFG partition, and then *shrink-wrapping* them. Similarly, after

partitioning, the cumulative weights of the cut-arcs determine the utilization level of the communication bus.

Any arbitrary partition of the modified DFG will be feasible or infeasible, based on the feasibility constraints defined for the problem. Furthermore, each feasible partition will have a cost associated with it that corresponds to the value returned by the objective function applied to the shrink-wrapped hardware.

Based on this problem representation, an algorithm for finding the optimum solution to the coupled design problems is shown in Figure 15. This algorithm is clearly exponential and computationally intractable. Thus, heuristic techniques will be investigated that are tractable and that return near-optimal solutions.

```
For All Graph Partitions {
   Assign Tasks to Processors
   Shrink-Wrap Processors
      If (Feasible) [
         If (Cost < Best) [
            Update Best
         ]
      ]
}
Return Best
```

Figure 42
OptimumAlgorithm

The rest of this section is organized as follows. Section 3.2 summarizes pending work in this area.

3.2 Pending Work

An Outline of pending work is shown in Figure 43.

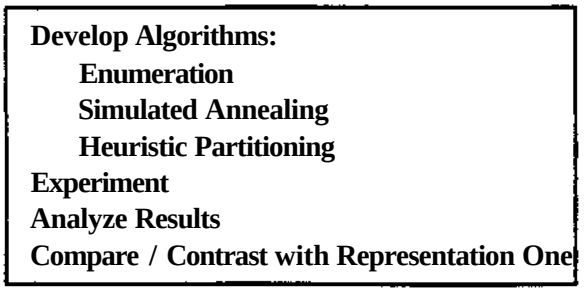


Figure 43
Outline of Pending Work

4 Summary and Future Directions

This paper has focused on two coupled design problems relevant to multi-computer systems: task allocation and processor specification. Two distinct problem representations were presented. One was based on a packing paradigm and the other on graph partitioning.

Automated solution strategies based on the packing representation were conceived, implemented and evaluated. Two algorithms, SA and DA, were found to be effective automated solution strategies. Furthermore, a trade-off between run-time and solution quality was found to exist for these two algorithms.

The graph-partitioning representation remains unexplored. An outline of pending work was presented. This work will be undertaken and subsequently reported by the authors.

4.1 Maximum Software Delay Paths

One area of future research involves a problem extension for supporting the specification of maximum software delay paths.

An important class of DFGs are those that have real-time *latency* requirements. This means that the time needed to execute a specified sequence of tasks along a path through the DFG is constrained by a maximum value. In effect, this amounts to specifying maximum path delays through the DFG. This represents an important class of problems since latency requirements often arise in control applications, which is a dominant sector of the embedded system market.

An extension to the existing approaches is needed to handle this type of problem. Specifically, the software model must be modified to accept path delay specifications. Next, the two problem representations, packing-based and graph partitioning-based, would have to be extended to handle a new class of constraints.

Specifically, a constraint would be needed to determine whether a sequence of DFG tasks completes before an arbitrary deadline. Note, however, that when the task sequence is implemented, bus traffic due to interprocessor communication may contribute to completion time. Furthermore, interprocessor communication is a function of task allocation decisions. Therefore, constraint satisfaction is a function of task allocation and the processor specifications of all processors involved with tasks along the path. This type of constraint does not fit the present paradigm, since feasibility constraints are presently defined in the context of a single task set assigned to a single processor. Thus, extensions to the problem representations would be needed, as well as extensions to the derived solution techniques.

4.2 Model Development

Another area of future work is in the area of model development. The set of design variables, feasibility constraints and the objective function introduced in Section 1 were adequate for developing the design strategies. However, if these strategies are to be used on real designs, then models will be needed that match the characteristics of the problem class being designed. Specifically, models would be needed for the following items:

1. **Processor Design Variables**
2. **Bus Transmission Characteristics**
3. **Bus Scheduling Characteristics**
4. **Processor Scheduling Characteristics**
5. **I/O Device Interface Characteristics**
6. **Objective Function**
7. **Feasibility Constraints**

5 References

[I] J. Beck, *"Characterization of an Automotive Pawertrain Control Application,"* Tech Report, Delco Electronics Corp., 1994.

[2] J. Beck and C. Lupini, *"Modeling Automotive Intercontroller Communication,"* Tech Report, Delco Electronics Corp., 1994.

[3] Birmingham, Gupta and Siewiorek, *Automating the Design of Computer Systems: The MICON Project,* Jones and Bartlett, 1992.

[4] W. Brantley, *"Automatically Decomposing Signal Processing Applications on Multiprocessors,"* Ph.D. Thesis, Carnegie Mellon University, 1978.

[5] *"Class C: General Motor's High Speed Serial Data Link Protocol and Protocol Handler Application Document,"* Tech Report, Delco Electronics Corp., 1993.

[6] J. Donnett, *"Simulated Annealing and Code Partitioning for Distributed Multimicroprocessors,"* Tech Report 87-194, Queen's University, 1987.

[7] M. Garey and D. Johnson, *Computers and Intractability: A Guide to the Theory cfNP Completeness,* W. H. Freeman, 1979.

[8] M. Huang, F. Romero and A. Sangiovanni-Vincentelli, *"An Efficient General Cooling Schedule for Simulated Annealing,"* Proceedings of the IC CAD Conference, 1986.

[9] S. Kirkpactrick, C. Gelatt and M. Vecchi, *"Optimization by Simulated Annealing,"* Science, Vol. 220, N. 4598, pp 671-680, May, 1983.

[10] T. Nielsen, *Combinatorial Bin Packing Problems,* University of Arizona, 1985.

[II] Rich and Knight, *Artificial Intelligence,* McGraw Hill, 1991.

[12] K. Tindell, *"Allocating Real-Time Tasks, (An NP-Hard Problem made Easy)"*, Tech Report YCS-149, University of York, 1990.

Appendix A: Input DFGs

name	nodes	NODES						
		Deriod (ms)	CPU	ROM	RAM	DIO	AIO	PIO
			(a,b of unit-PE capacity)					
RANI	100	(3(100,5)	GKO.7,0.7)	CK29.3,34.2)	CK2.9,0.9)	GK3.1,3.2)	**GK0,5.6)**	GK0,17.5)
RAN2	100	(3(80,5)	GKO.4,0.4)	GK3.9,3.9)	CK5.4,3.9)	GK3.1,3.2)	(3(0,5.6)	GK0,17.5)
RAN3	10	(3(1000,300)	(3(3.1,1.3)	GK12.5,1.7)	CK5.1,2.9)	(3(4.7,6.3)	GK12.5,2.5)	GKO.20.0)
RAN4	120	GK500,50)	G(0.5,0.6))	GK50.0,24.4)	GK8.8,19.5)	CK6.2,9.3)	(3(12.5,12.5)	**GK0,20.0))**
RAN5	50	(3(400,10)	CKO.2,0.1)	GK2.9,1.5)	GK3.4,1.5)	(3(3.1,0.6	GK0,1.3)	**GK0,5.0)**
RAN6	200	(3(70,3)	GKO.2,0.03)	GK48.8,9.7)	(3(1.9,0.5)	GK4.7,4.7)	GK12.5,2.5)	**GK0,15.0)**
RAN7	50	GK2O,1)	GKO.4,0.2)	GK6.8,3.9)	CK2.5,1.9)	GK9.4,6.3)	(3(0,2.5)	**GK0,20.0)**
RAN8	75	CK1500,50)	GK2.5,1.3)	GKl.2,0.5)	CK39.0,9.8)	(3(9.4,3.1)	GKO.8.8)	**GK0,5.0)**
RAN9	90	(K1700,80)	GKO.3,0.1)	GK1.2,0.5)	CK39.0,14.6)	GK3.1,3.1)	€K0,8.8)	**GK0,5.0)**
RAN10	25	(3(150,8)	GKO.01,0.01)	(3(19.5,3.9)	(3(5.7,9.8)	GK6.2,6.2)	GK9.4,6.2)	**GK0,5.0)**
RAN11	100	G(100,l)	(3(4.0,2.0)	(3(39.1,19.5)	(3(39.1,19.5)	GK46.9,21.9)	(3(45.0,26.3)	**GK50.0,15.0)**

name	arcs	ARCS			
		source	dest	Deriod (ms)	IMC (%BW)
RANI	200	U(l,100)	U(l,100)	G(100,5)	CKO.012,0.011)
RAN2	200	U(l,100)	U(l,100)	GK80,5)	GK0.(»5,0.076)
RAN3	10	U(l,10)	U(l,10)	CK1000,300)	CK40.9.0.82)
RAN4	0	-	-	G(500,50)	-
RAN5	50	**UU.50)**	U(l,50)	GK400,10)	CK2.5,1.2)
RAN6	25	U(l,200)	U(l,200)	(K70.3)	G(0.036,0.011)
RAN7	110	U(l,50)	U(l,50)	GK20,l)	G(0.011,0.055)
RAN8	100	U(l,75)	U(l,75)	G(1500,50)	(XO.82,0.087)
RAN9	50	U(l,90)	U(l,90)	G<1700,80)	GKO.82,0.087)
RAN10	50	U(l,25)	UU.25)	G(150,8)	G(0.16,0.66)
RAN11	10	U(l,100)	U(l,100)	GK1OO,l)	G<0.92,0.021)

KEY:

6(a,b) = Gaussian Probability Distribution: means a standard deviation = b

U(a,b) = Uniform Probability Distribution: Range = [a,b]

Random DFGs (Specified)

name	nodes	NODES: Avg. (CPU period)	Avg. ROM	Avg. RAM	Avg. DIO	Avg. AIO	Avg. PIO
		(4% of unit-PE capacity)					
RANI	100	8.70	31.44	2.93	3.09	1.13	3.50
RAN2	100	5.45	4.06	5.67	3.09	1.13	3.50
RAN3	10	7.05	12.27	6.10	4.06	12.5	0
RAN4	120	1.21	50.07	5.85	7.68	13.65	5.00
RAN5	50	0.51	2.67	1.57	0	0	0
RAN6	200	2.86	48.52	1.98	5.23	12.44	1.00
RAN7	50	18.78	6.19	5.45	8.56	0	6.00
RAN8	75	1.63	1.22	40.09	8.79	2.83	0
RAN9	90	0.15	2.43	39.29	3.13	2.92	0
RAN10	25	0.11	18.31	10.48	5.88	10.00	0
RAN11	100	41.25	39.40	39.87	44.75	45.37	48.50

name	arcs	ARCS: Avg. (IMCV(Deriod) (%BW)
RANI	200	0.14
RAN2	200	0.55
RAN3	10	48.45
RAN4	0	-
RAN5	50	5.92
RAN6	25	0.51
RAN7	110	0.91
RAN8	100	0.55
RAN9	50	0.48
RANIO	50	2.60
RAN11	10	9.29

Random DFGs (Actual)

		NODES					
name	nodes	Av. (CPUV(Seriod)	Avg. R ok.	Avg. RAM	m as		Avg. I PIO
		(% of unit-PE capacity)					
SUPERCH	154	0.165	3.95	5.53	0	0	0.33
TRACTION	160	0.161	3.84	9.33	0	0	0.63
IAC	78	0.244	4.34	5.85	0	0	0

		ARCS	
name	arcs	AVR. (IMCVtoeriod)	j 1
		(%BW)	
SUPERCH	316	0.348	
TRACTION	282	0.359	
IAC	95	0.416	

RedDFGs(Scaled)

		NODES					
name	nodes	1 Avg. (CFUWoeriod)	Av Rofa	Av RASI	Aft	to:	Avg. \ PIO i
		(% ofuhit-PE capacity)					1
SUPERCH	154	0.331	39.72	5.53	0	0	0.33
TRACTION	160	0.322	38.67	9.33	0	0	0.63
IAC	78	0.488	43.64	5.85	0	0	0

		ARCS	
name	arcs	Avi ! (IMCWtferiod)	!
		(%BW)	
SUPERCH	316	0.696	
TRACTION	282	0.719	
IAC	95	0.833	

RealDFGs(Unsealed)

name	nodes	NODES					
		Avg. (CPUVCoeriod)	Avg. ROM	ATK« O.ATHT	«IK« T^Tfl	tut	Av PI& i
		(% of unit_PE capacity)					
		V.v.V.UI.U.U.U." A.D Capacity/—					
CONl	8	9.56	28.02	36.26	12.11	18.75	25.0
CON2	12	18.03	21.30	20.65	10.67	15.62	25.0

name	arcs	ARCS Avg. (IMCVfoeriod) (%BW)
CONl	7 (tree)	10.40
CON2	12 (loop)	5.86

ContrivedDFGs

Appendix B : Upper and Lower Packing Bounds

Upper and Lower bounds on the number of bins needed to pack a DFG are calculated as follows:

Let the DFG be defined by a set of Nodes and Edges:

$DFG = \{N, E\}$

The number of nodes equals the cardinality of the node set.

$NODES = CARD(N)$

Consider a node: «, e N. From Figure 2:

$n_t = (x, p, x, 6, e, <|>, Y)$

Define the *demand vector* for a node as follows:

$$\vec{d}_i = \left(\left\lceil \frac{\beta}{a} \right\rceil, \chi, \delta, \varepsilon, \phi, \gamma\right)$$

Define *capacity vector* for the unit-PE; from Figure 4:

$\vec{cap} = (a, b, c, d, e, f)$

Now, define *cumulative demand vector* to be the vector sum of all demand vectors:

$$sum = \sum_{Vi} \vec{d}_i$$

Last, define *maximum demand vector* to be the vector whose elements are the max value for that dimension over all demand vectors.

$\vec{max} = (MAX_{Vi}(1), \ldots, MAX_{Vi}(6))$ where $d_i(k) = k^{th}$ *element of* i^{th} *demand vector*

Then:

$$LowerBound = \left\lceil \frac{SMm(t)}{\vec{cap}(i)} \right\rceil$$

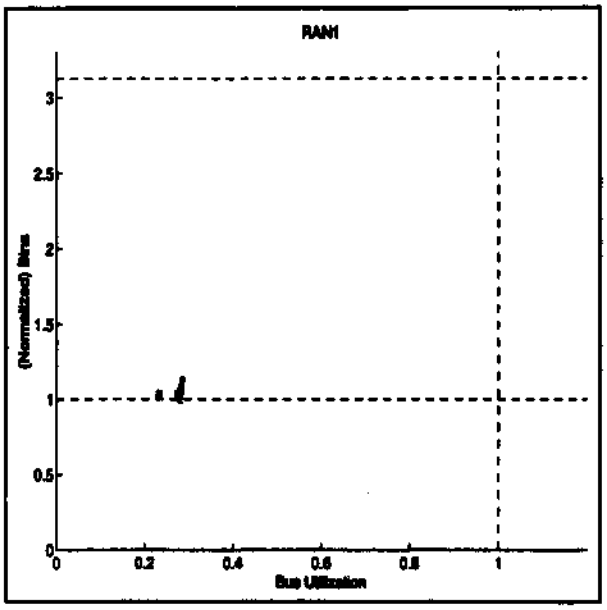
$$UpperBound = \left\lceil \frac{Nodes}{MIN_{Vij}\left(\frac{cap\,i}{max}\right)} \right\rceil$$

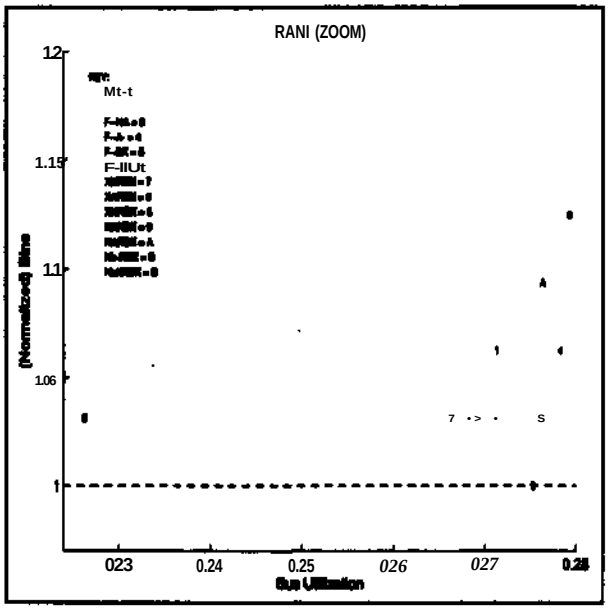
Appendix C: Heuristic Packing Algorithm Results

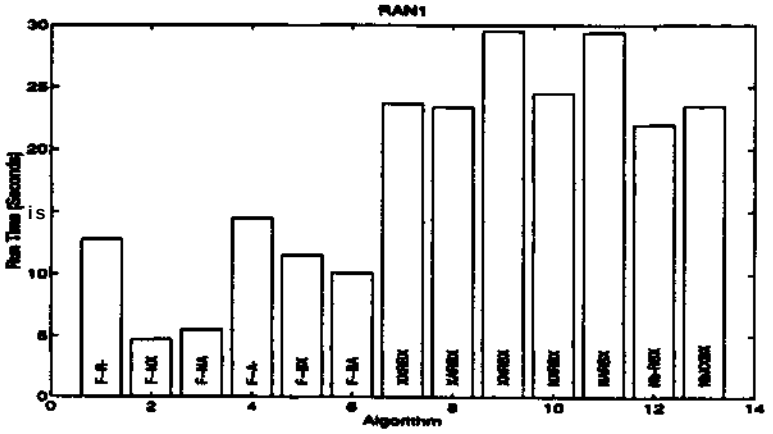
DFG Rani:

Alg	Bins	BusUtil.	Run lime
LB	32(1)	*m*	-
F-R-	34(1.06)	0.2715	12.8 (tec)
F-NX	32(1)	0.2752	4.7
F-NA	33 (1.03)	0.2687	5.5
F-A-	34(1.06)	0.2775	14.5
F-BX	33(1.03)	0.2763	11.5
F-BA	33(1.03)	0.2681	10.1

Alg	Bins	BusUtil.	RunTíme
XXRBX	33 (1.03)	0.2657	23.7
XARBX	33 (1.03)	0.2710	23.4
XNRBX	33 (1.08)	0.2691	29.5
NXRBX	36(1.13)	0.2792	24.5
NARBX	35 (1.10)	0.2760	29.4
Nb-RBX	33 (1.03)	0.2262	22.0
NhXXBX	33 (1.03)	0.2262	23.5
UB	100(3.12)	-	-



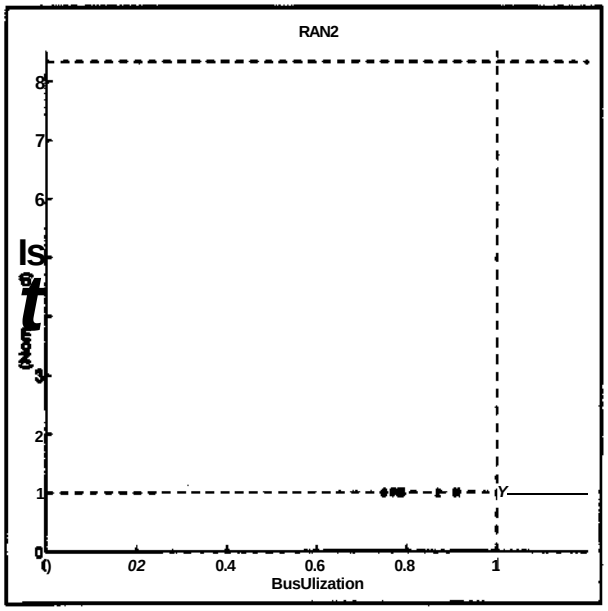


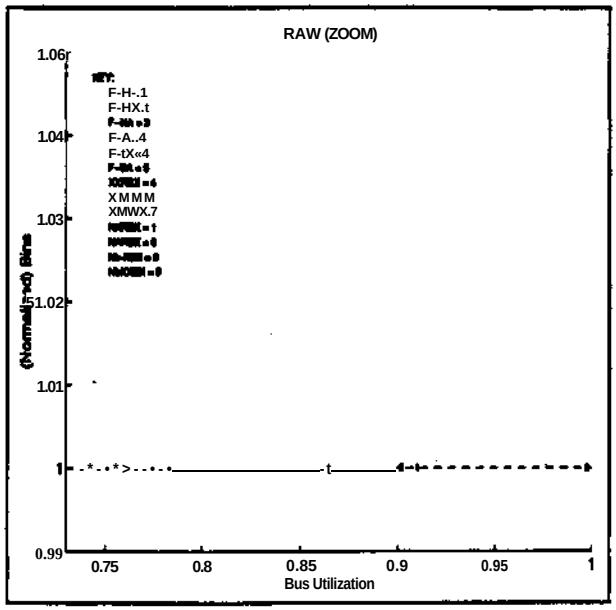


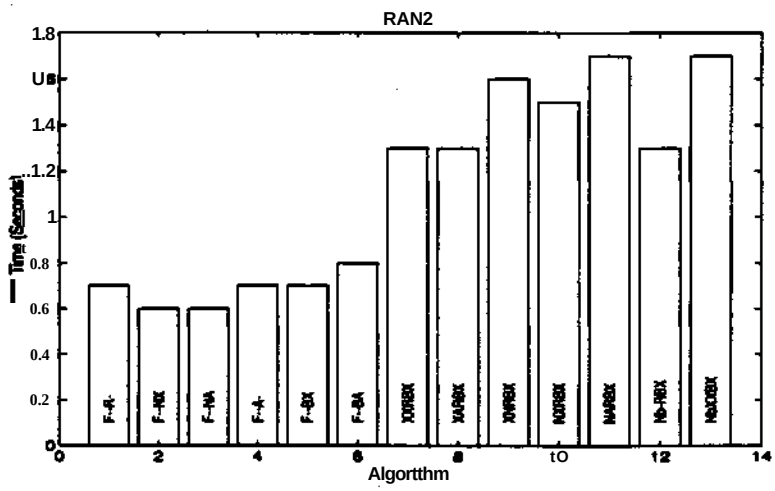
DFG Ran2:

Alg	Bins	Bus Util.	RunTime
LB	6(1)	-	—
F-R-	6(1)	0.9084	0.7 (sec)
F-NX	6(1)	0.8627	0.6
F-NA	6(1)	0.9003	0.6
F-A-	6(1)	0.7609	0.7
F-BX	6(1)	0.7836	0.7
F-BA	6(1)	0.7609	0.8

Alg	Bins	Bus Util.	RunTime
XXRBX	6(1)	0.7609	1.3
XARBX	6(1)	0.7757	1.3
XNRBX	6(1)	0.7639	1.6
NXRBX	6(1)	0.9072	1.5
NARBX	6(1)	0.9964	1.7
Nb-RBX	6(1)	0.7430	1.3
NbXXBX	6(1)	0.7430	1.7
UB	50(8.33)	-	-



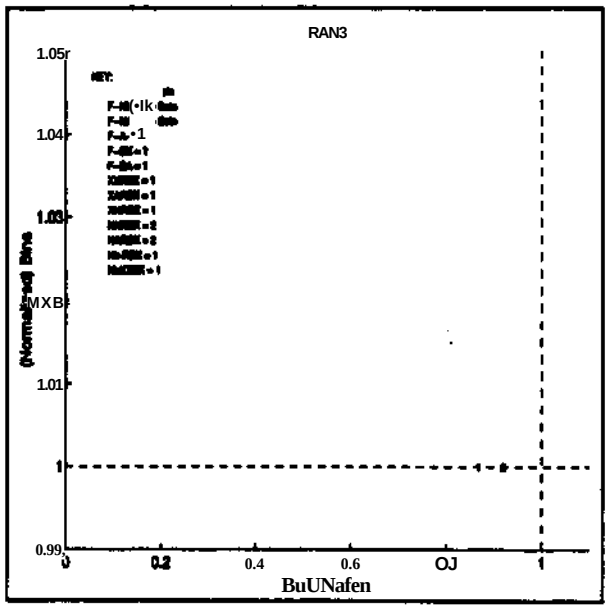




DFG Ran3:

Alg	Bins	BusUtil.	RunTime
LB	2(1)	-	-
F-R-	NoSoln	-	< 0.1 (see)
F-NX	NoSoln	-	<0.1
F-NA	NoSoln	-	<0.1
F-A-	2(1)	0.8625	< 0.1
F~BX	2(1)	0.8625	< 0.1
F-BA	2(1)	0.8625	<0.1

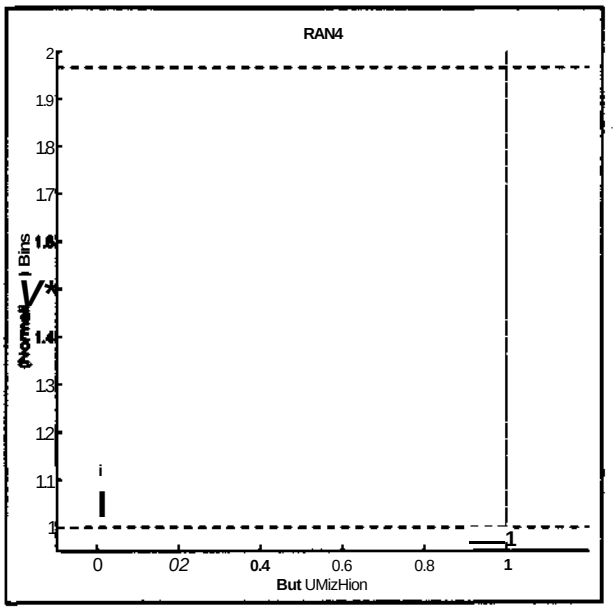
Alg	Bins	Bus Util.	Run Time
XXRBX	2(1)	0.8625	< 0.1
XARBX	2(1)	0.8626	<0.1
XNRBX	2(1)	0.8625	< 0.1
NXRBX	2(1)	0.9145	< 0.1
NARBX	2(1)	0.9145	<0.1
Nb-RBX	2(1)	0.8625	< 0.1
NhXXBX	2(1)	0.8625	< 0.1
UB	2(1)	-	-

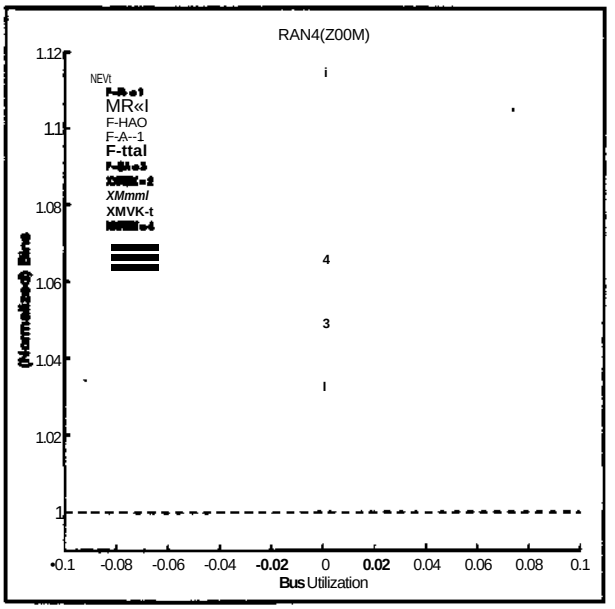


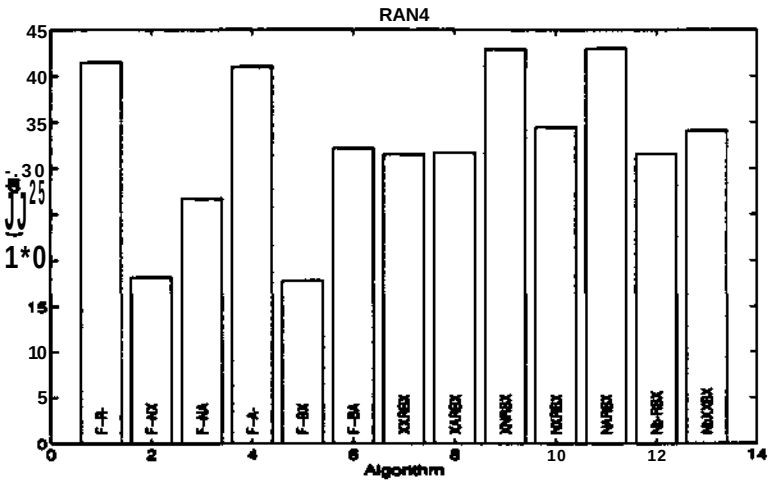
DFG Ran4:

Alg	Bins	Bus Util.	RunTime
LB	61(1)	-	-
F~R-	68(1.115)	0	41.5 (dec)
F-NX	63 (1.033)	0	18.2
F-NA	64(1.049)	0	26.7
F-A-	68(1.115)	0	41.0
F-BX	63 (1.033)	0	17.8
F-BA	64(1.049)	0	32.2

Alg	Bins	Bus Util.	RunTime
XXRBX	63 (1.033)	0	31.5
XARBX	63 (1.033)	0	31.7
XNRBX	63 (1.033)	0	42.9
NXRBX	65 (1.066)	0	34.5
NARBX	64(1.049)	0	43.0
Nb-RBX	63 (1.033)	0	31.6
NbXXBX	63 (1.033)	0	34.1
UB	120(1.967)	-	-

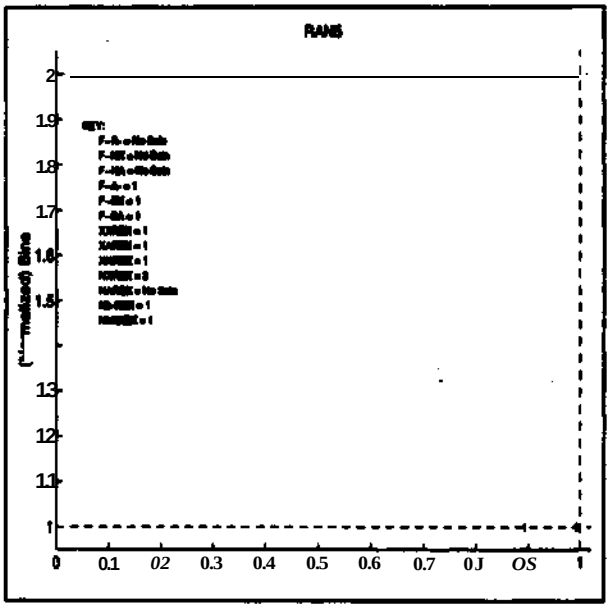


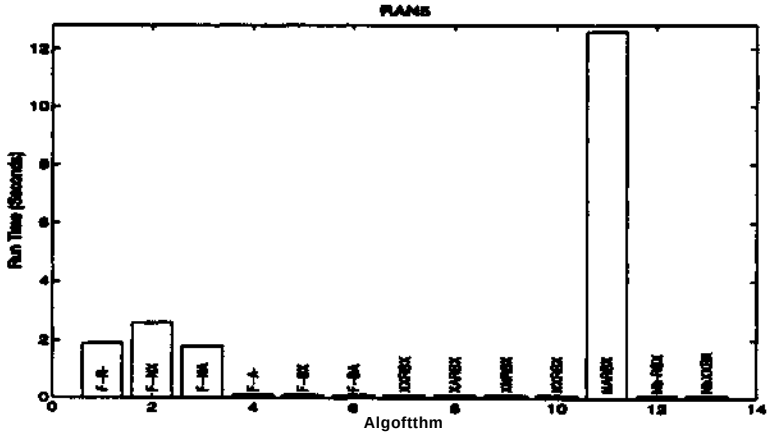




DFGRanS:

Alg	Bins	BuaUtil.	RunTime	Alg	Bins	BuflUtil.	RunTime
LB	2(1)	-	-	XXRBX	2(1)	0.8882	0.1
F-R-	NoSoln	-	1.9 (MC)	XARBX	2(1)	0.8882	0.1
F~NX	NoSoln	-	2.6	XNRBX	2(1)	0.8882	0.1
F-NA	NoSoln	-	1.8	NXRBX	2(1)	0.9898	0.1
F-A-	2(1)	0.8882	0.1	NARBX	NoSoln	.	12.6
F~BX	2(1)	0.8882	0.1	Nb-RBX	2(1)	0.8882	0.1
F-BA	2(1)	0.8882	0.1	NhXXBX	2(1)	0.8882	0.1
				UB	4(2)	-	-

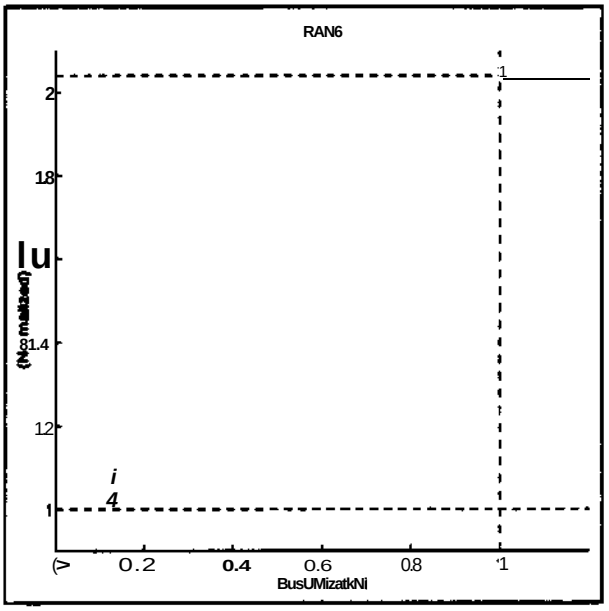


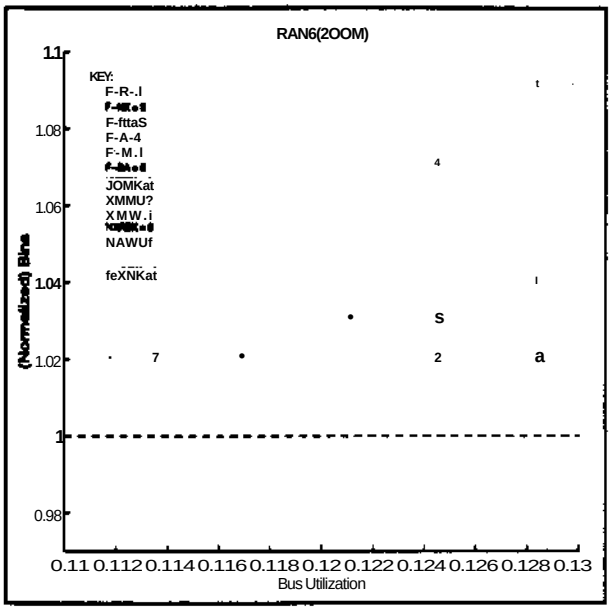


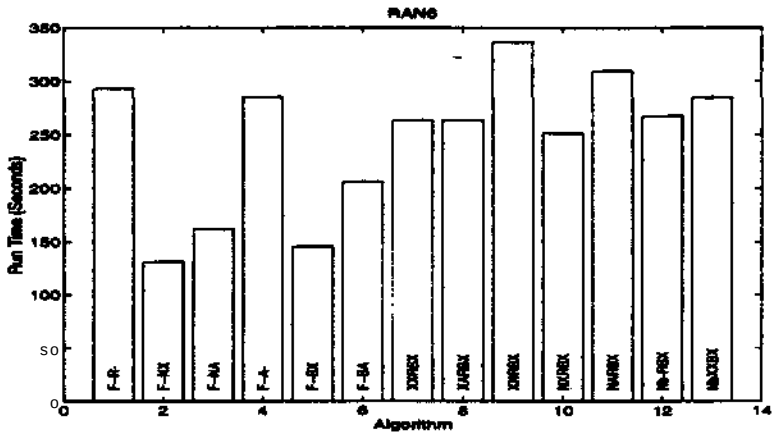
DFG Ran6:

Alg	Bins	Bus Util.	RunTime
LB	**98(1)**	**-**	*m*
F-4t-	**107(1.092)0.1283**		**293.1 (sec)**
F-NX	**100(1.020) 0.1244**		**131.6**
F-NA	**100(1.020)0.1283**		**162.1**
F-A-	**105(1.071)0.1244**		**285.2**
F-BX	**101(1.031)0.1244**		**145.9**
F-BA	**101 (1.031) 0.1244**		**205.9**

Alg	Bins	**Bus** Util.	RunTime
XXRBX	**100(1.020)**	**0.1168**	**263.5**
XARBX	**100(1.020)**	**0.1134**	**263.5**
XNRBX	**101 (1.031)**	**0.1244**	**336.8**
NXRBX	**102 (1.041)**	**0.1283**	**251.3**
NARBX	**102 (1.041)**	**0.1283**	**309.6**
Nb-RBX	**101 (1.031)**	**0.1210**	**267.3**
NbXXBX	**101 (1.031)**	**0.1210**	**284.6**
UB	**200(2.041)**	**-**	**-**



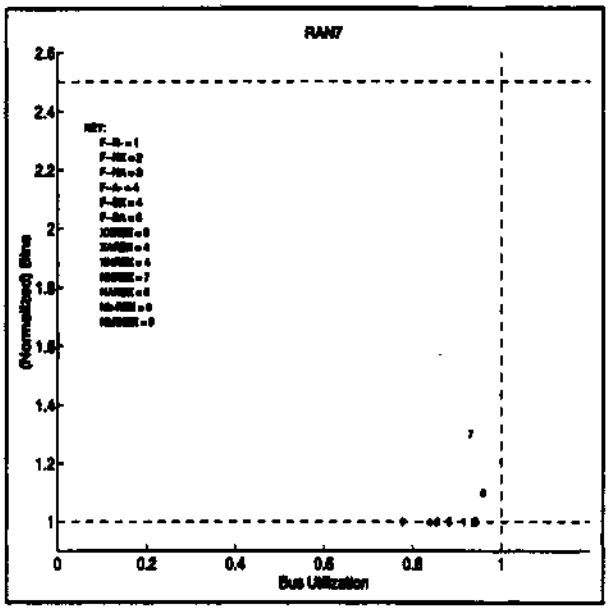


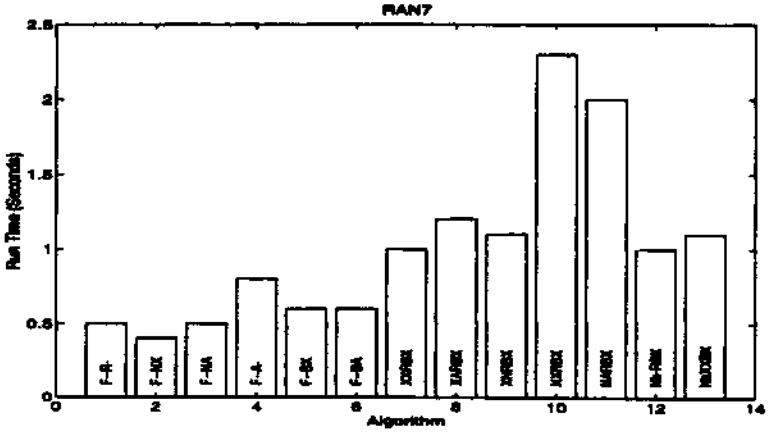


DFG Ran7:

Alg	Bins	BuaUtil.	RunTime
LB	10(1)	*m*	-
F-R-	10(1)	0.9101	0.5 (MC)
F-1«	10(1)	0.9380	0.4
F-MA	10(1)	0.9372	0.5
F-A-	10(1)	0.8380	0.8
F-BX	10(1)	0.8380	0.6
F-BA	10(1)	0.8769	0.6

Alg	Bins	BusUtil.	RUB lime
XXRBX	10(1)	0.8487	1.0
XARBX	10(1)	0.8320	1.2
XNRBX	10(1)	0.8320	1.1
NXRBX	13 (1.3)	0.9854	2.3
NARBX	11 (1.1)	0.9531	2.0
Nb-RBX	10(1)	0.7712	1.0
NhXXBX	10(1)	0.7712	1.1
UB	25(2.5)	-	-

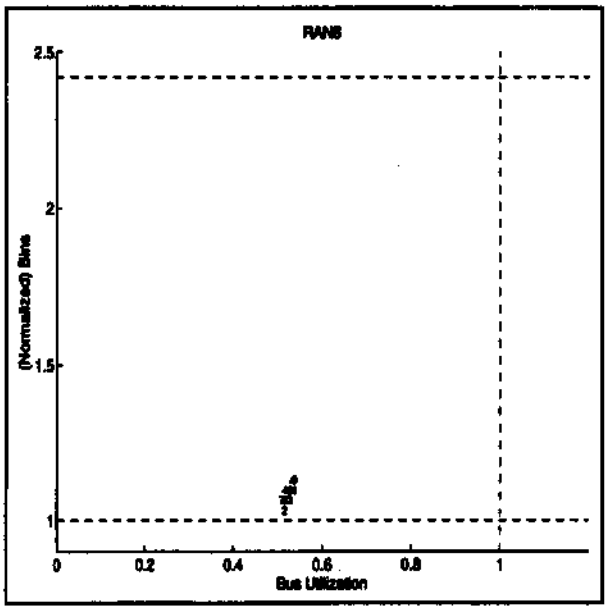


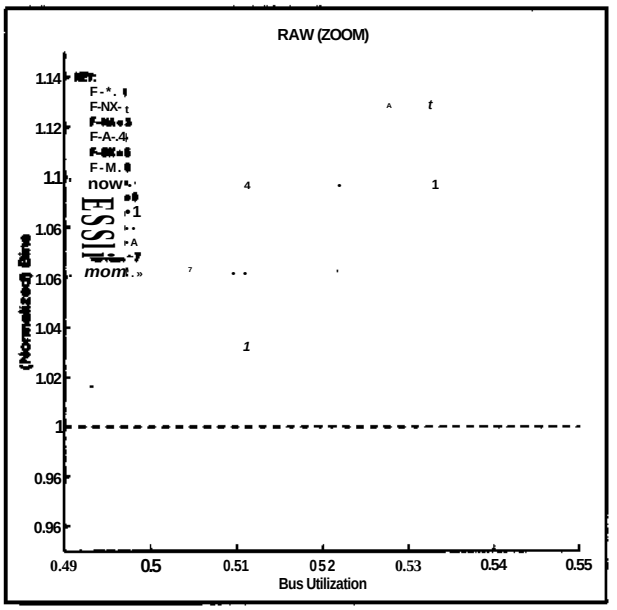


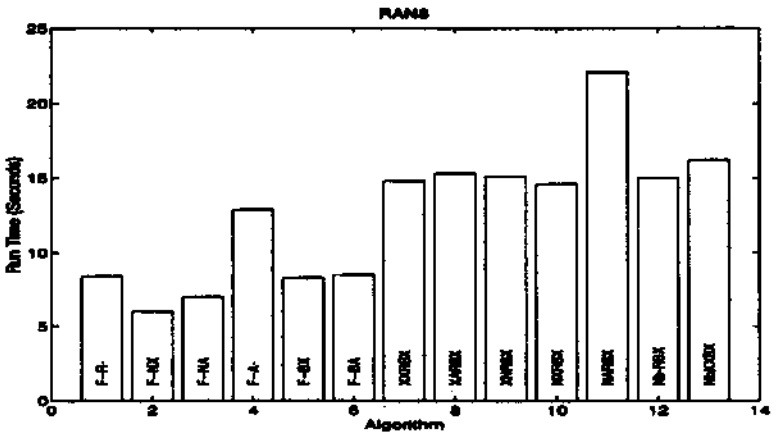
DFG Ran8:

Alg	Bins	Bus Util.	RunTime
LB	31 (1)	-	-
F-R*	34 (1.097)	0.5326	8.4 (sec)
F-NX	32 (1.032)	0.5109	6.0
F-NA	33 (1.065)	0.5213	7.0
F-A-	34(1.097)	0.5109	12.9
F-BX	33(1.065)	0.5109	8.3
F-BA	34(1.097)	0.5215	8.5

Alg	Bins	Bus Util.	Run Time
XXRBX	33 (1.065)	0.5042	14.8
XARBX	33 (1.065)	0.5092	15.3
XNRBX	33 (1.065)	0.5109	15.1
NXRBX	35 (1.129)	0.5323	14.6
NARBX	35 (1.129)	0.5274	22.1
Nb-RBX	33 (1.065)	0.5042	15.0
NbXXBX	33 (1.065)	0.5042	16.2
UB	75 (2.419)	-	-

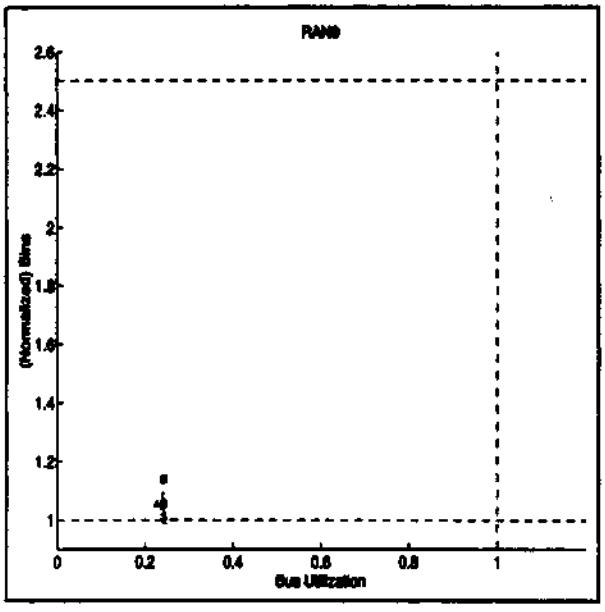


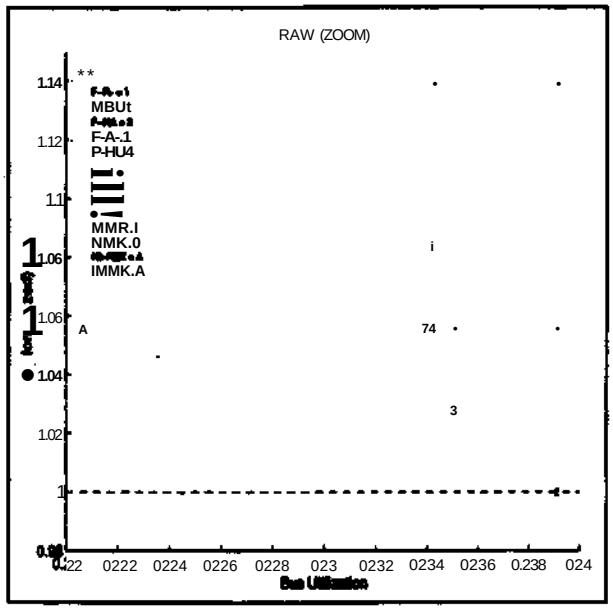


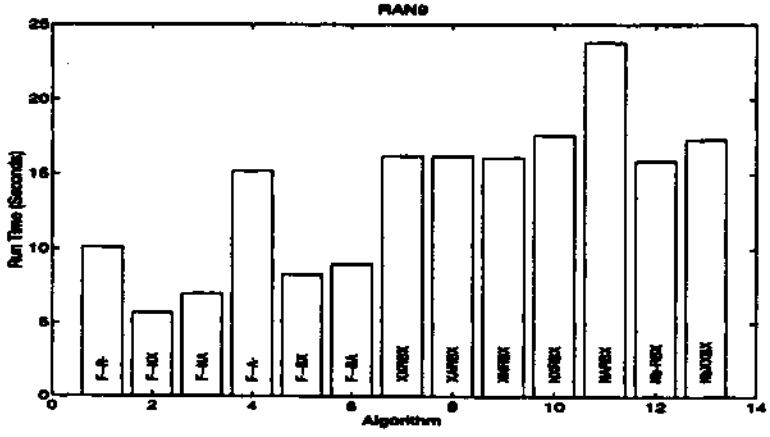


DFG Ran9:

Alg	Bins	BusUtil.	Run Time	Alg	Bins	BusUtil.	RunTime
LB	36(1)	*m*	—	XXRBX	38 (1.055)	0.2390	16.2
F-RP	39(1.083)	0.2341	10.1 (see)	XARBX	38(1.055)	0.2338	16.2
F-NX	36(1)	0.2390	5.7	XNRBX	38(1.055)	0.2341	16.1
F-NA	37(1.028)	0.2349	7.0	NXRBX	41 (1.139)	0.2390	17.6
F-A-	39(1.083)	0.2341	15.2	NARBX	41 (1.139)	0.2342	23.8
F-BX	38(1.056)	0.2341	8.3	Nb-RBX	38(1.055)	0.2205	15.9
F-BA	38(1.055)	0.2350	9.0	NNQCBX	38(1.055)	0.2205	17.3
				UB	90(2.500)	-	-

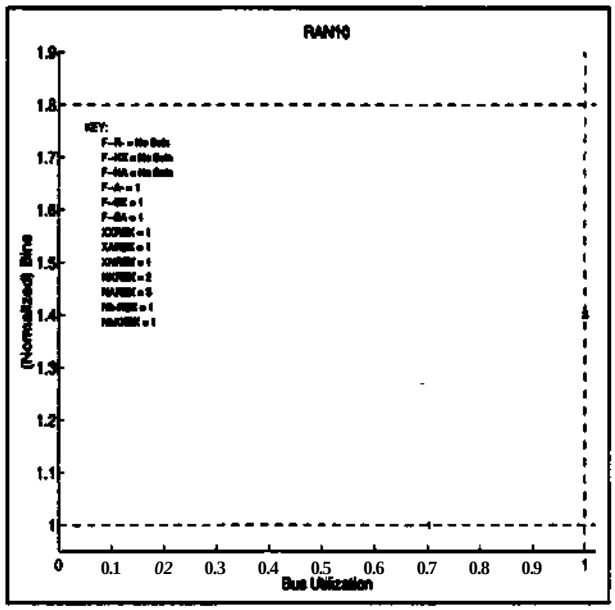


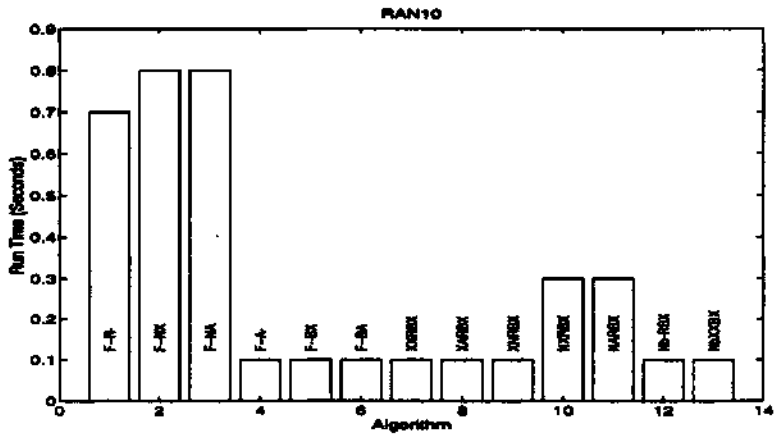




DFG RanlO:

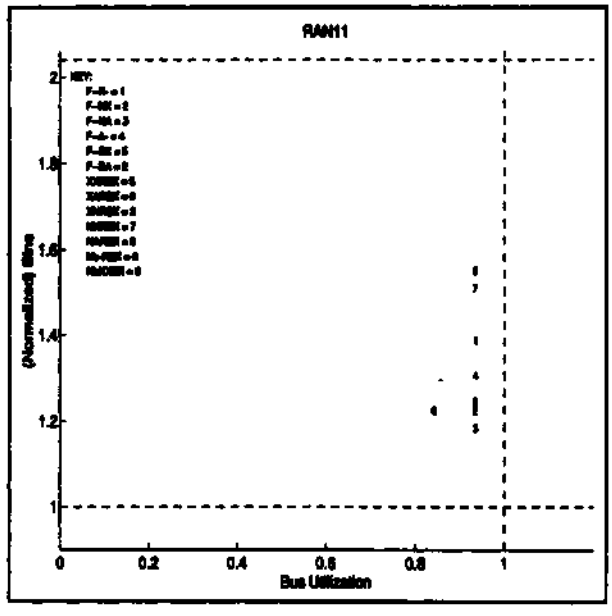
Alg	Bins	BusUtil.	RunTime	Alg	Bins	Bus Util.	RunTime
LB	5(1)	-	*m*	XXRBX	5(1)	0.6978	0.1
F-R-	NoSoln	-	0.7 (sec)	XARBX	5(1)	0.6978	0.1
F~NX	NoSoln	-	0.8	XNRBX	5(1)	0.6978	0.1
F-NA	NoSoln	-	0.8	NXRBX	7 (1.4)	0.9993	0.3
F-A-	5(1)	0.6978	0.1	NARBX	7 (1.4)	0.9958	0.3
F-BX	5(1)	0.6978	0.1	Nb-RBX	5(1)	0.6978	0.1
F-BA	5(1)	0.6978	0.1	NbXXBX	5(1)	0.6978	0.1
				UB	9(1.8)	-	-

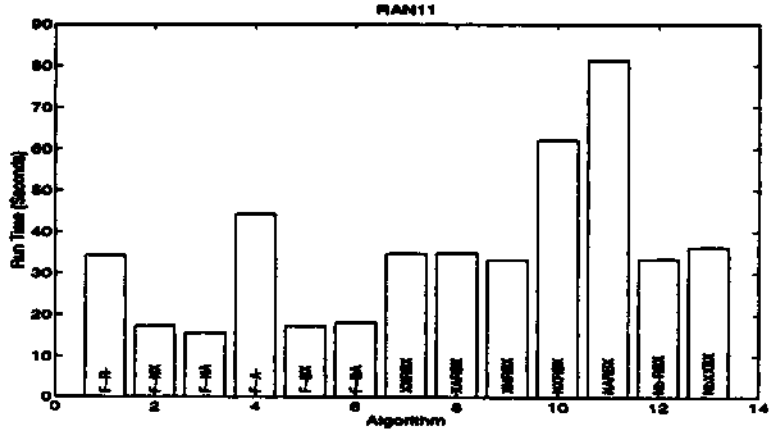




DFG Ranll:

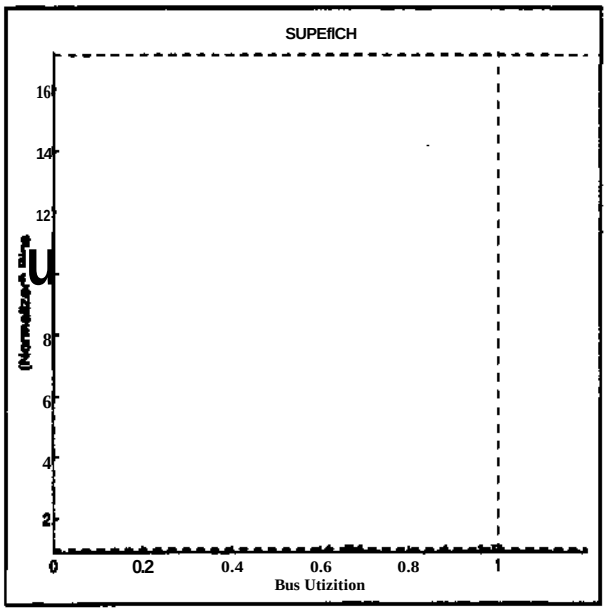
Alg	Bins	BuflUtil.	RunTime	Alg	Bins	BuaUtiL	RunTime
LB	49(1)	-	-	XXRBX	61(1.245)	0.9288	34.7
F-R-	68(1.387)	0.9288	34.3 (sec)	XARBX	60(1.224)	0.8367	34.9
F-NX	60(1.224)	0.9288	17.3	XNRBX	60(1.224)	0.9288	33.3
F-NA	58(1.184)	0.9288	15.6	NXRBX	74(1.510)	0.9288	62.1
F-A-	64(1.306)	0.9288	44.2	NARBX	76(1.551)	0.9288	81.4
F-BX	61(1.245)	0.9288	17.3	Nb-RBX	60(1.224)	0.8367	33.5
F-BA	60(1.224)	0.9288	18.2	NbXXBX	60(1.224)	0.8367	36.3
				UB	100(2.04)	-	-

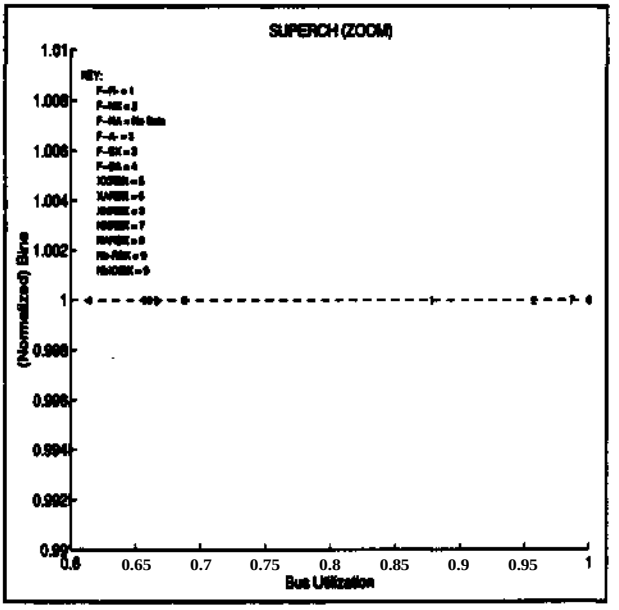


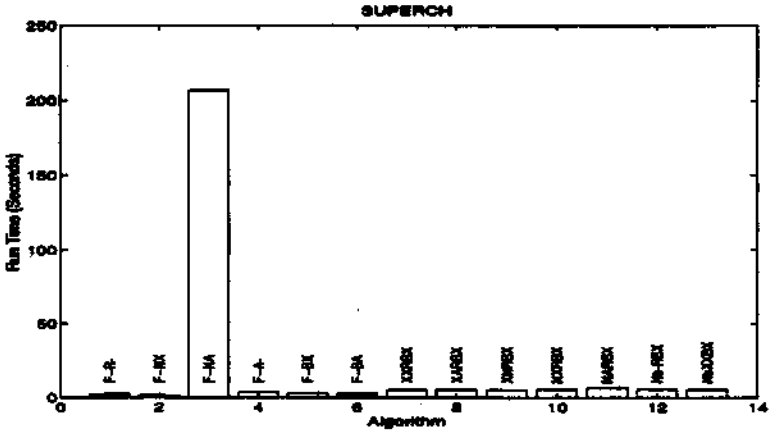


DFG Superch (Scaled):

Alg	Bins	Bus Util.	RunTime	Alg	Bins	Bus Util.	RunTime
LB	9(1)	-	-	XXRBX	9(1)	0.6856	5.3
F-R-	9(1)	0.8760	2.3 (see)	XARBX	9(1)	0.6588	5.3
F-NX	9(1)	0.9550	1.4	XNRBX	9 (1)	0.6641	5.2
F-NA	NoSoln	.	206.8	NXRBX	9(1)	0.9845	5.8
F-A-	9(1)	0.6641	3.7	NARBX	9(1)	0.9981	6.8
F-BX	9(1)	0.6641	2.9	Nb-RBX	9(1)	0.6550	5.7
F-BA	9(1)	0.6123	2.8	NbXXBX	9(1)	0.6550	5.5
				UB	154 (17.11)	-	-



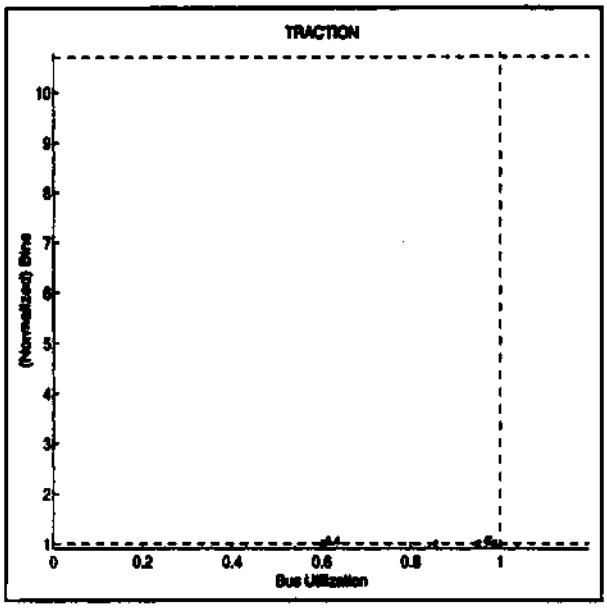


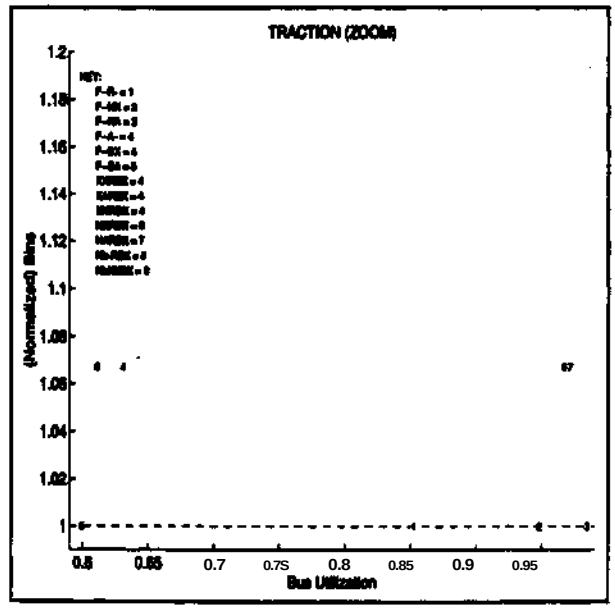


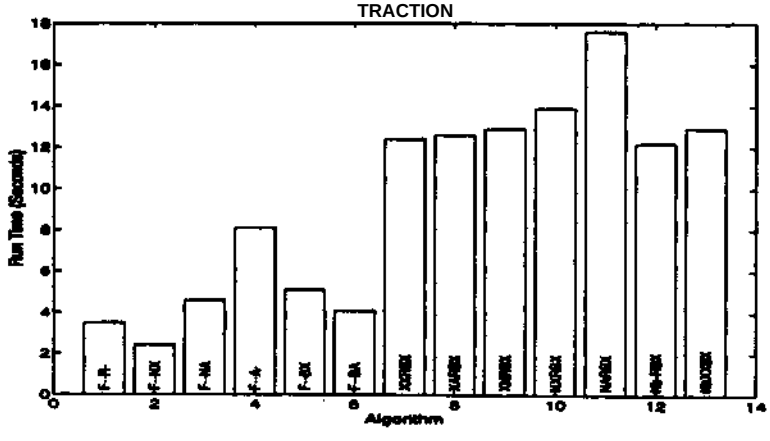
DFG Traction (Scaled):

Alg	Bins	BusUtil.	RunTime
LB	15(1)	*m*	-
F-4t-	15(1)	0.8503	3.5 (we)
F-NX	15(1)	0.9461	2.4
F-NA	16(1.067)	0.9821	4.6
F-A-	16(1.067)	0.6287	8.1
F-BX	16(1.067)	0.6287	5.1
F-BA	15(1)	0.5972	4.1

Alg	Bins	BusUtil.	Run lime
XXRBX	16(1.067)	0.6287	12.4
XARBX	16(1.067)	0.6290	12.0
XNRBX	16(1.067)	0.6287	12.9
NXRBX	16(1.067)	0.9648	13.9
NARBX	16(1.067)	0.9696	17.6
Nb-RBX	16(1.067)	0.6090	12.2
NhXXBX	16(1.067)	0.6090	12.9
UB	160(10.67)	-	-

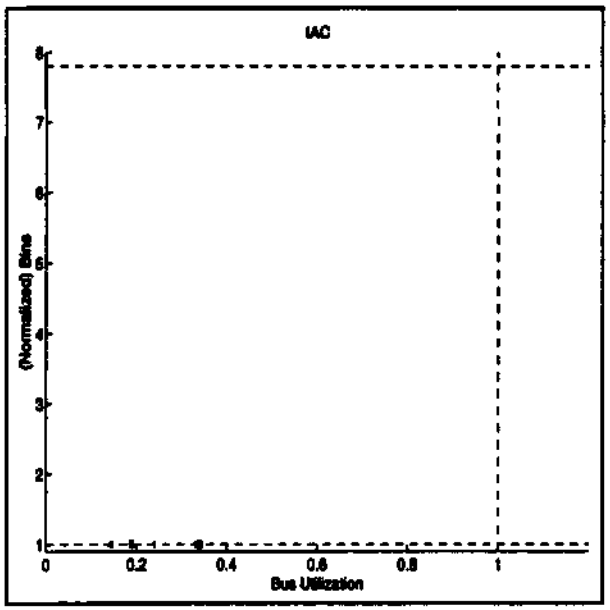


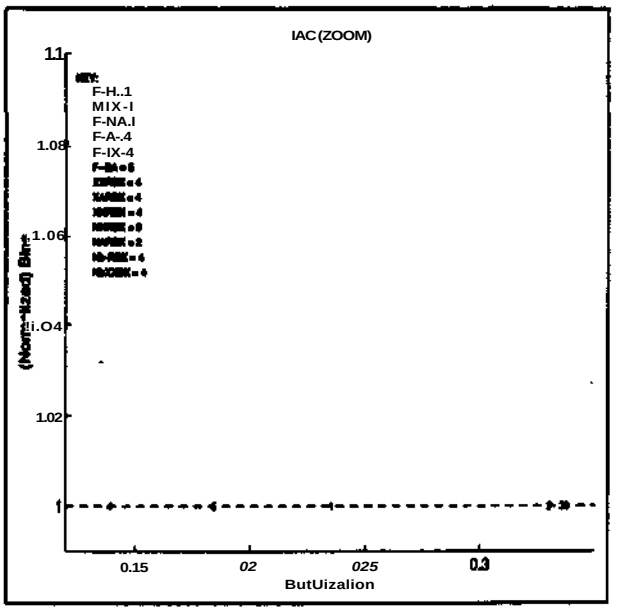


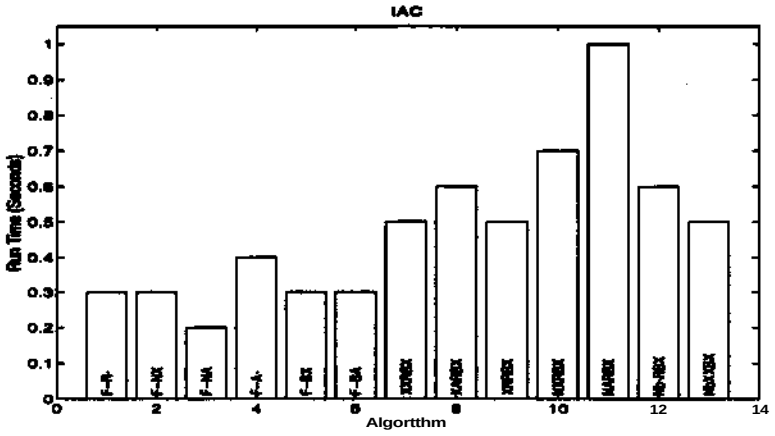


DFG IAC (Scaled):

Alg	Bins	Bus Util.	RunTime	Alg	Bins	Bus Util.	RunTime
LB	5(1)	-	-	XXRBX	5(1)	0.1378	0.5
F~R-	5(1)	0.2341	0.3 (sec)	XARBX	5(1)	0.1378	0.6
F-NX	5(1)	0.3291	0.3	XNRBX	5(1)	0.1378	0.5
F-NA	5(1)	0.3347	0.2	NXRBX	5(1)	0.3368	0.7
F-A-	5(1)	0.1378	0.4	NARBX	5(1)	0.3298	1.0
F-BX	5(1)	0.1378	0.3	Nb-RBX	5(1)	0.1377	0.6
F-BA	5(1)	0.1831	0.3	NbXXBX	5(1)	0.1377	0.5
				UB	39(7.8)	-	-



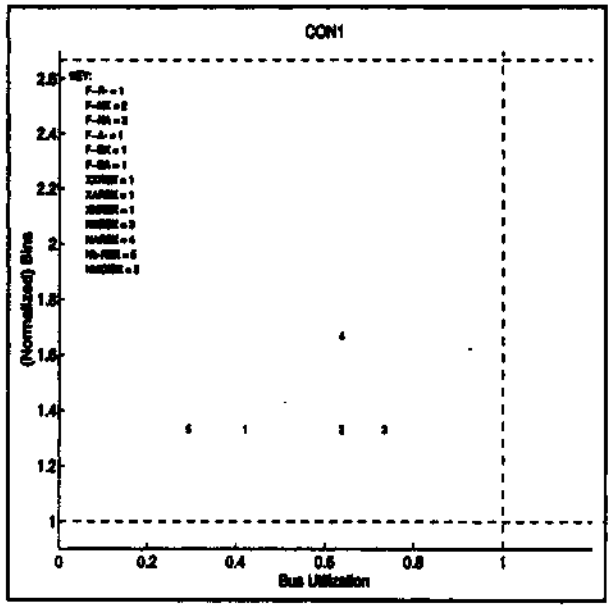




DFG Conl:

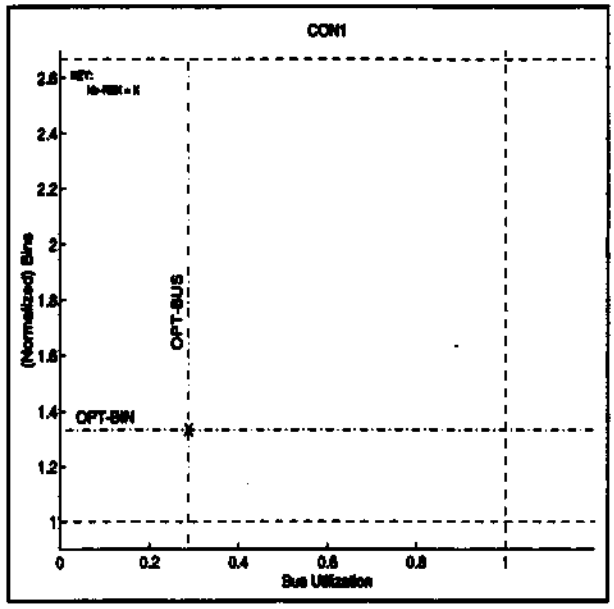
Alg	Bins	Bus Util.	Run Time
LB	3(1)	-	-
F~R-	4(1.333)	0.4153	< 0.1 (sec)
F-NX	4(1.333)	0.6318	<0.1
F-NA	4(1.333)	0.7280	< 0.1
F-A-	4(1.333)	0.4153	<0.1
F-BX	4(1.333)	0.4153	< 0.1
F-BA	4(1.333)	0.4153	< 0.1

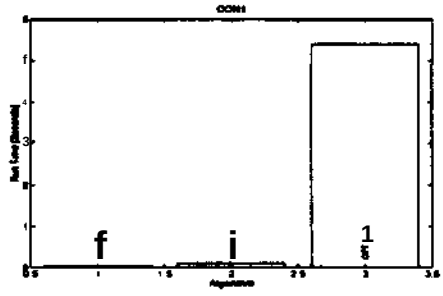
Alg	Bins	Bus Util.	Run Time
XXRBX	4 (1.333)	0.4153	< 0.1
XARBX	4(1.333)	0.4153	< 0.1
XNRBX	4(1.333)	0.4153	< 0.1
NXRBX	4(1.333)	0.7280	< 0.1
NARBX	5 (1.667)	0.6318	< 0.1
Nb-RBX	4 (1.333)	0.2863	< 0.1
NhXXBX	4(1.333)	0.2863	< 0.1
UB	8(2.667)	-	-



DFG Conl:

Alg	Bins	Bua Utih	Run Time
LB	3(1)	-	-
OPT-BINS	4(1.333)	0.3129	4.5 (sec)
OPT-BUS	5(1.667)	0.2863	0.1
Nb-RBX	4(1.333)	0.2863	<0.1
UB	8(2.667)	-	-

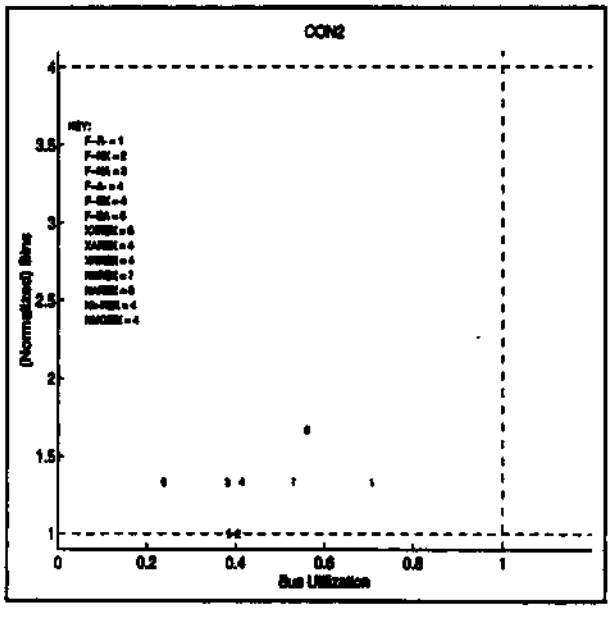




DFG Con2:

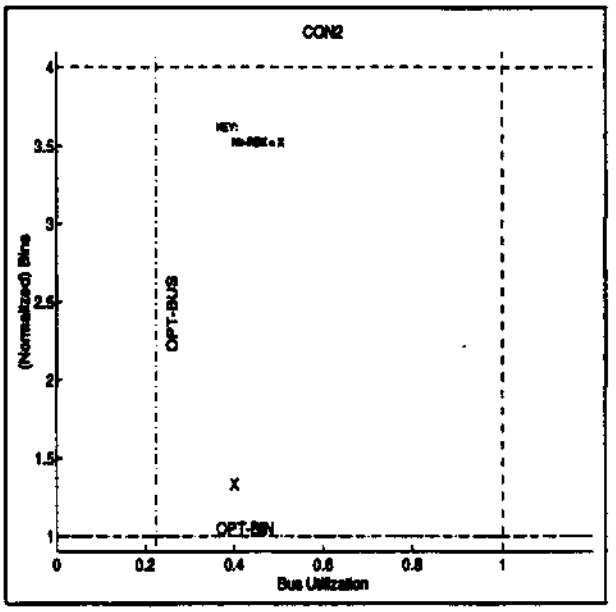
Alg	Bins	Bus Util.	Run Time
LB	3(1)	—	—
F-R-	4(1.333)	0.7010	< 0.1 (sec)
F-NX	3(1)	0.4000	<0.1
F-NA	4(1.333)	0.3754	<0.1
F-A-	4(1.333)	0.4079	< 0.1
F~BX	4(1.333)	0.4071	< 0.1
F-BA	3(1)	0.3797	< 0.1

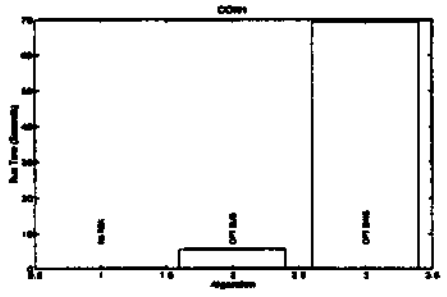
Alg	Bins	Bus Util.	RunTime
XXRBX	4(1.333)	0.2321	<0.1
XARBX	4(1.333)	0.4006	< 0.1
XNRBX	4(1.333)	0.4018	<0.1
NXRBX	4(1.333)	0.5246	< 0.1
NABBX	5(1.667)	0.5549	<0.1
Nb-RBX	4(1.333)	0.4007	<0.1
NhXXBX	4(1.333)	0.4007	< 0.1
UB	12(4.00)	-	-



DFGCon2:

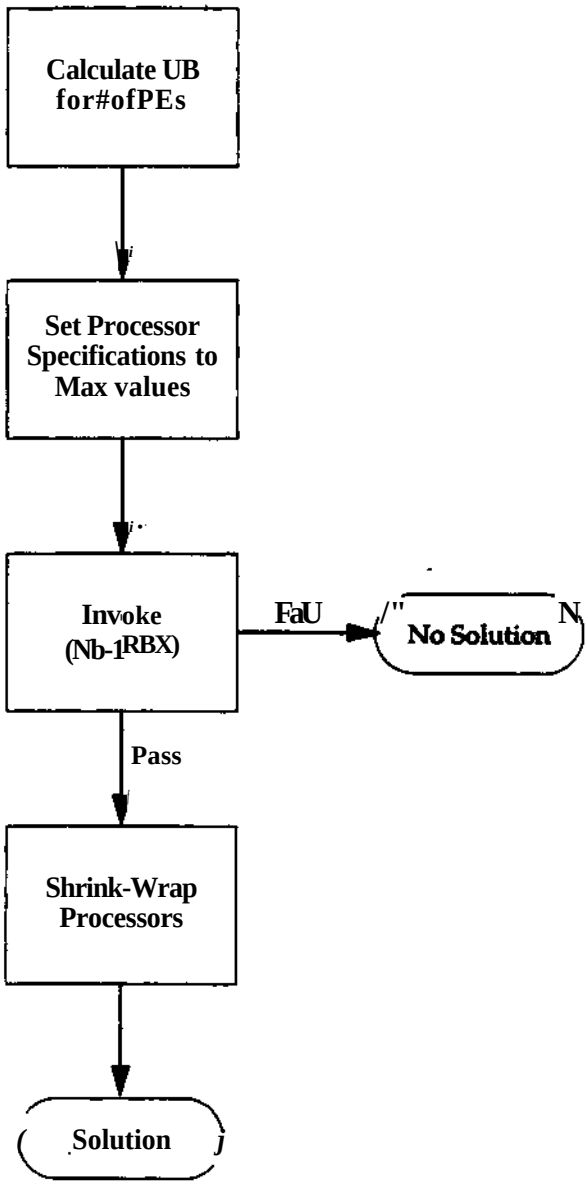
Alg	Bins	BusUtil.	RunTime
LB	3(1)	-	-
OPT-BINS	3 (1)	0.4000	69.4 (sec)
OPT-BUS	5(1.667)	0.2247	5.6
Nb-RBX	4(1.333)	0.4007	<0.1
UB	12(4)	-	-



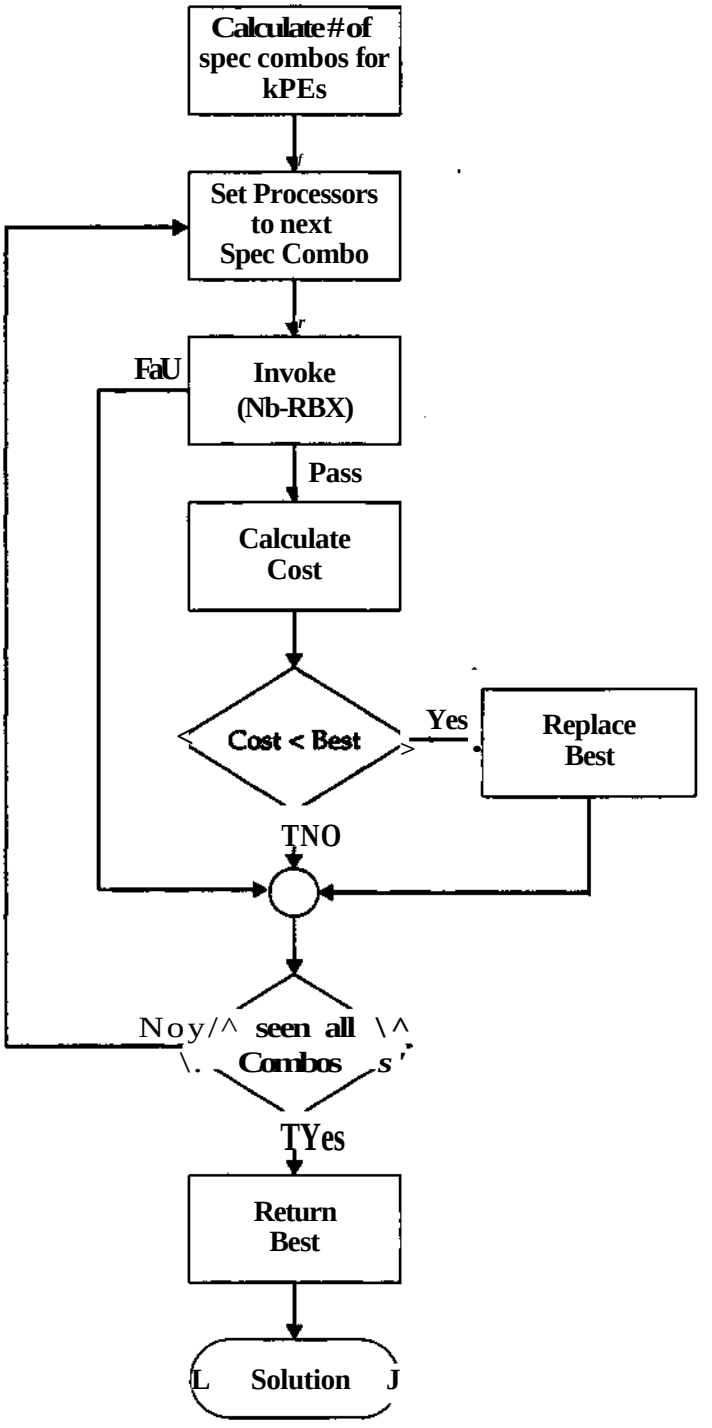


Appendix D: Flowcharts for Packing-Based Algorithms

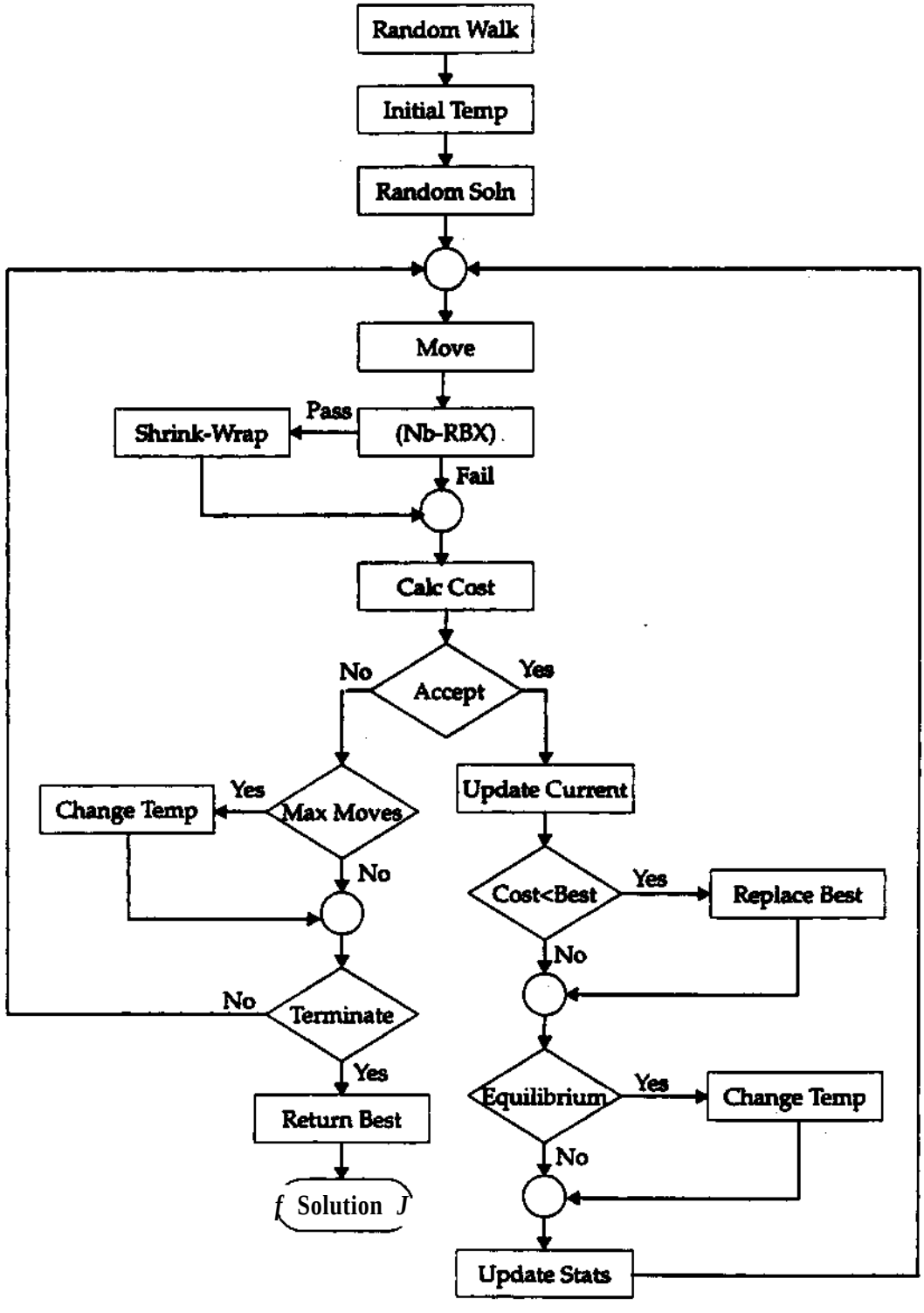
SW Algorithm



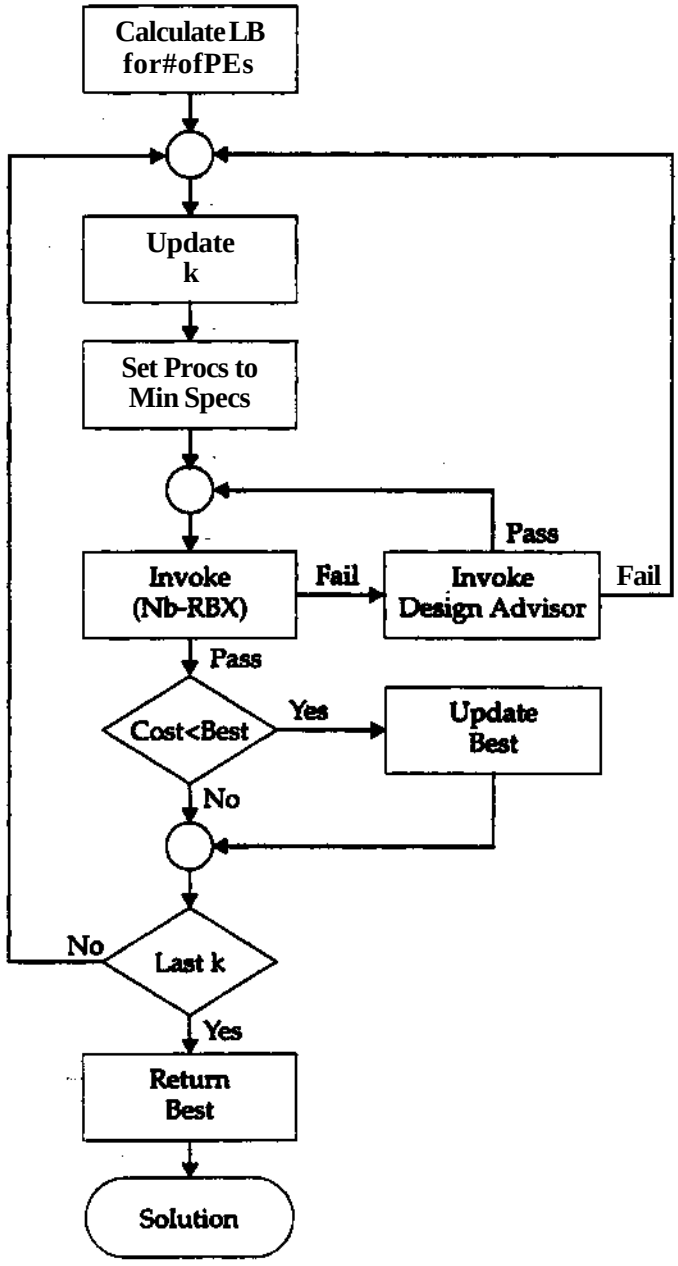
ENUM Algorithm



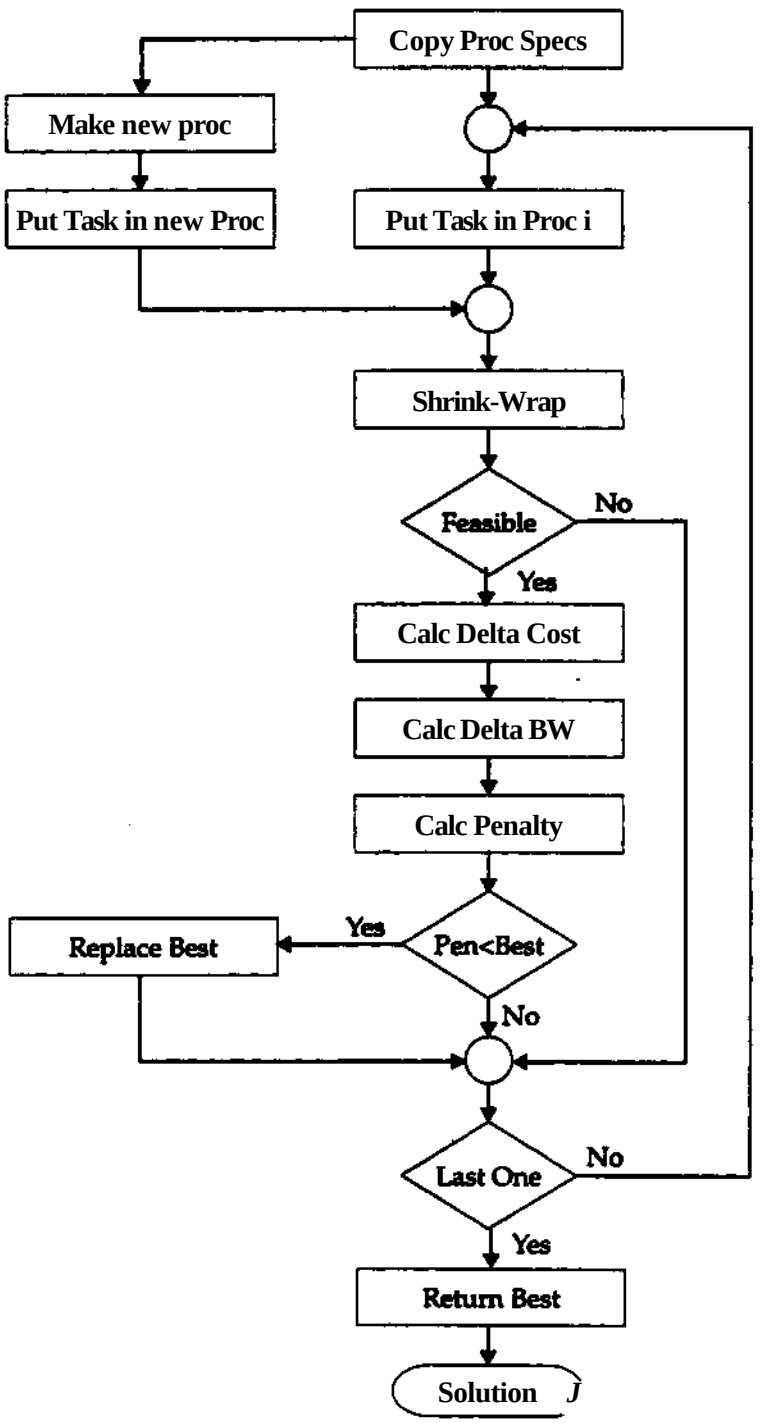
SA Algorithm



DA Algorithm



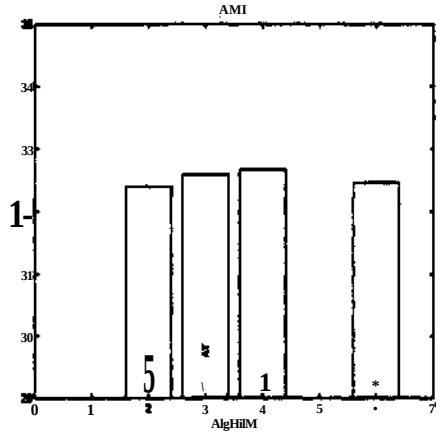
Design Advisor

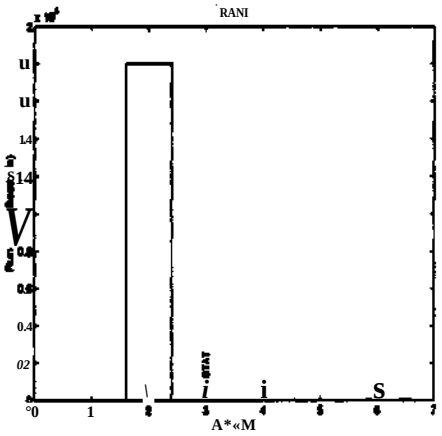


Appendix E: Results for Packing-Based Algorithms

DFG Rani:

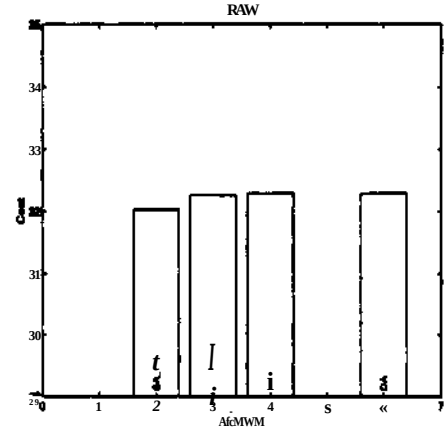
	ENUM	SAJSW	DAJSTAT	SW	DA
Processors	-	9	9	9	10
Cost	-	32.3971	32.5906	32.6705	32.4551
Run Time (Sec)	-	180,316.1L	11.2	4.7	37.3

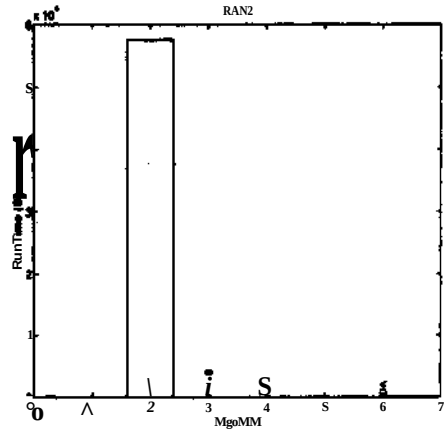




DFGRan2:

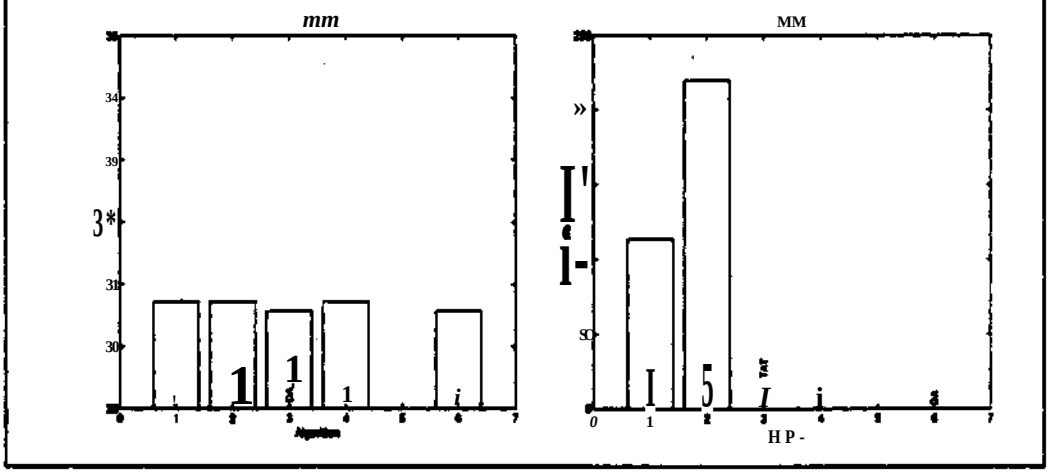
	ENUM	SAJSW	DA.STAT	SW	DA
Processors	-	6	6	6	6
Cost	-	32.0279	32.2634	32.2906	32.2906
Run Time (Sec)	-	57,602.4	4.3	4.6	12.9





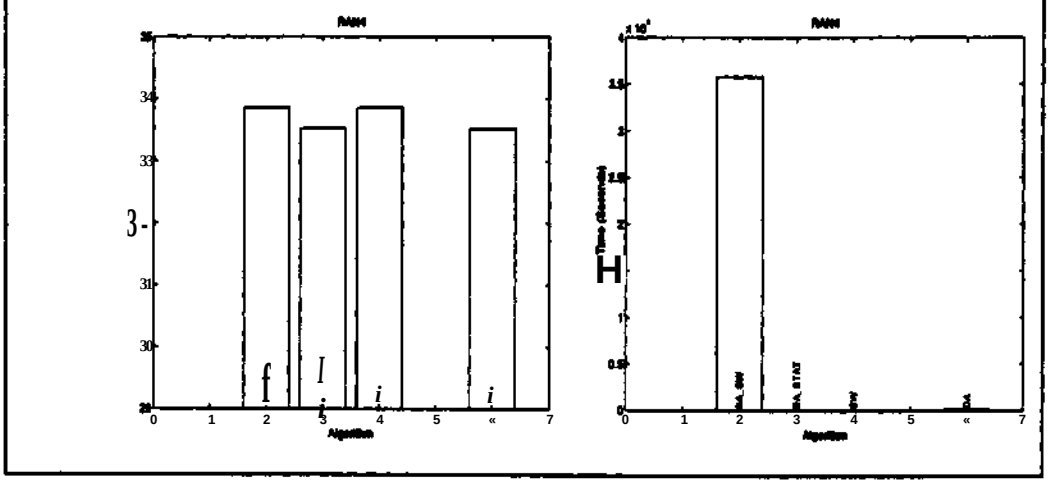
DFGRan3:

	BNUM	SAJSW	DAJSTAT	SW	DA
Processors	2	2	2	2	2
Cost	30.7247	30.7247	30.5733	30.7247	30.5733
Run Time (Sec)	113.7	219.8	<0.1	<0.1	0.1



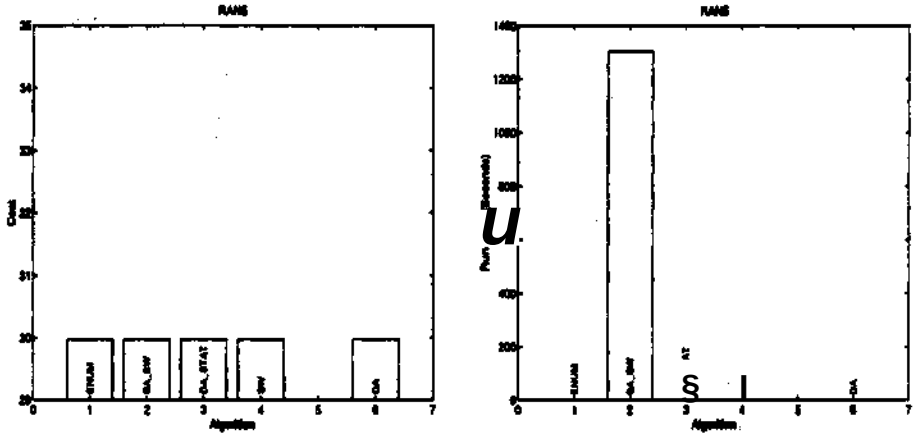
DFGRan4:

	ENUM	SAJSW	DAJSTAT	SW	DA
Processors	-	17	17	17	19
Coat	-	33.8509	33.5252	33.8509	33.5181
Run Time (Sec)	-	35,764.1	33.5	4.5	154.2



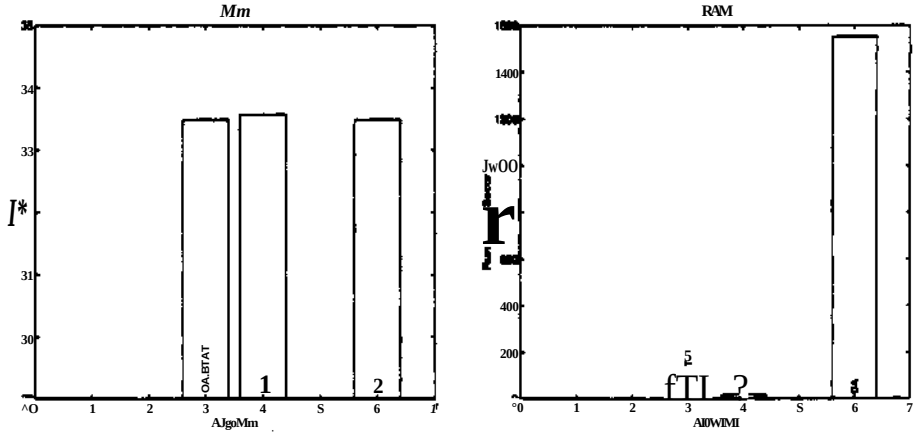
DFG Ran5:

	ENUM	SA_SW	DAJSTAT	SW	DA
Processors	1	1	1	1	1
Cost	29.9745	29.9745	29.9745	29.9745	29.9745
Run Time (Sec)	0.6	1,304.0	0.2	0.6	1.0



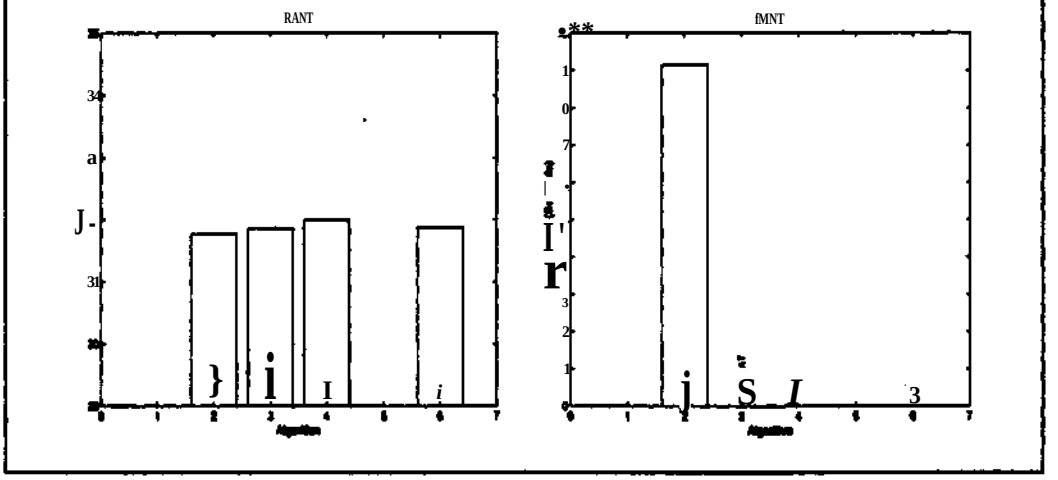
DFG Ran6:

	ENUM	SAJ3W	DAJSTAT	SW	DA
Processors	-	-	25	25	25
Cost	-	-	33.4856	33.5701	33.4856
Run Time (Sec)	-	-	130.2	17.8	1,553.5



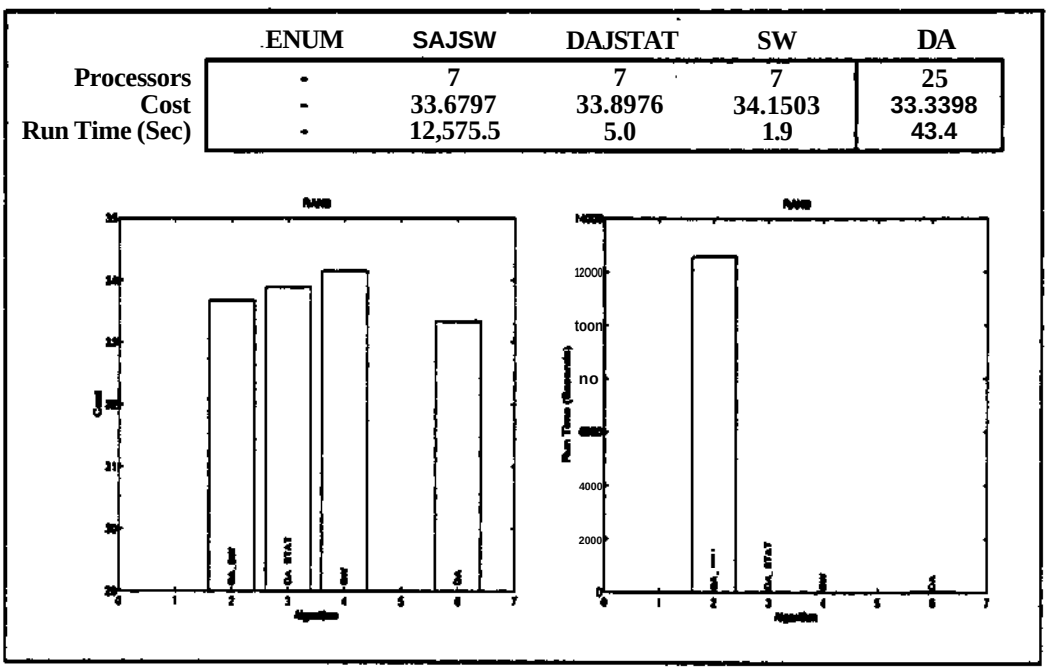
DFGRan7:

	ENUM	SAJSW	DAJSTAT	SW	DA
Processors	-	10	10	10	10
Cost	-	31.7677	31.8561	31.9902	31.8706
Run Time (Sec)	-	91,542.6	3.9	0.9	3.6



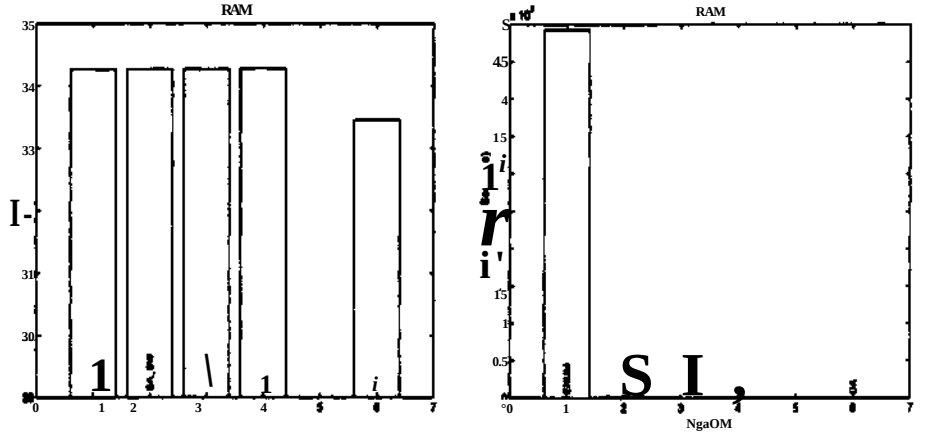
DFG Ran8:

	ENUM	SAJSW	DAJSTAT	SW	DA
Processors	-	7	7	7	25
Cost	-	33.6797	33.8976	34.1503	33.3398
Run Time (Sec)	-	12,575.5	5.0	1.9	43.4



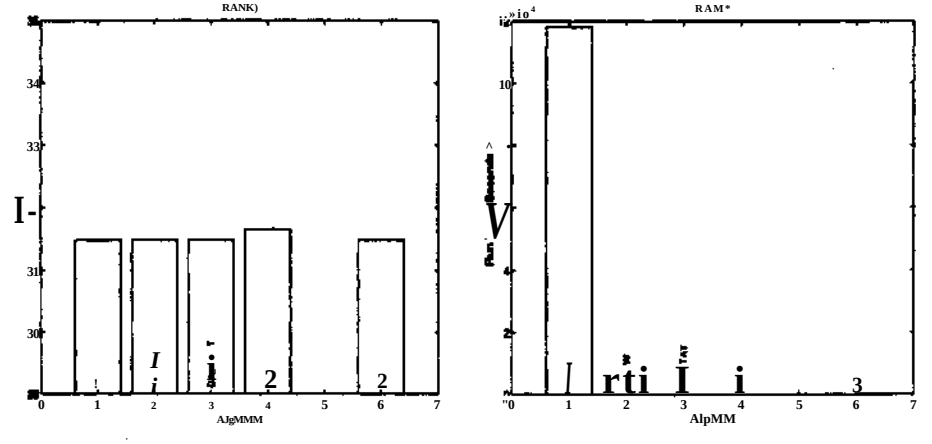
DFG Ran9:

	ENUM	SAJSW	DAJSTAT	SW	DA
Processors	-	**3**	3	**3**	**26**
Cost	-	**34.2626**	**34.2626**	**34.2773**	**33.4519**
Run Time (Sec)		**1,803.5**	**1.1**	**2.3**	**> 25.8**



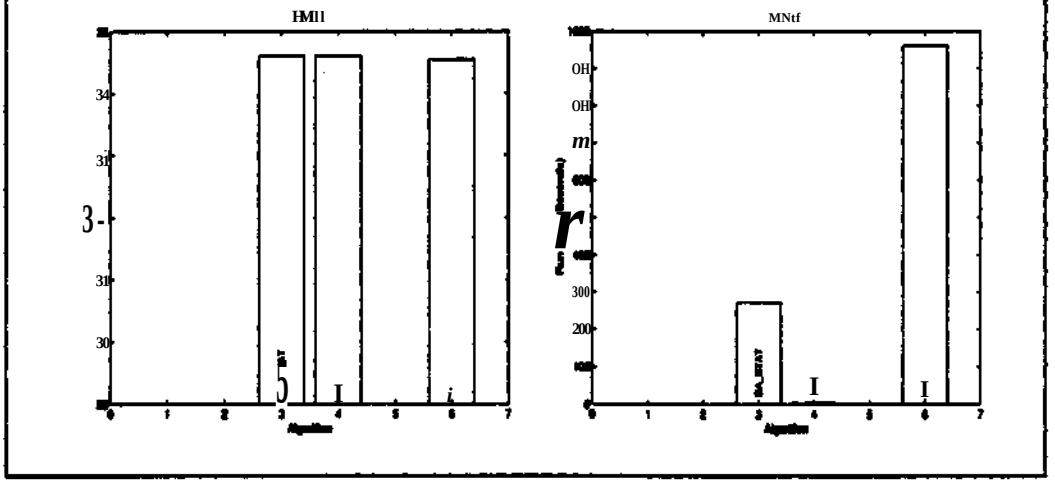
DFG RanlO:

	ENUM	SAJSW	DAJSTAT	**SW**	**DA**
Processors	**3**	3	3	**3**	**3**
Cost	**31.4815**	**31.4815**	**31.4815**	**31.6462**	**31.4815**
Run Time (Sec)	**118,252.3**	**6,616.4**	**0.3**	**0.1**	**1.6**



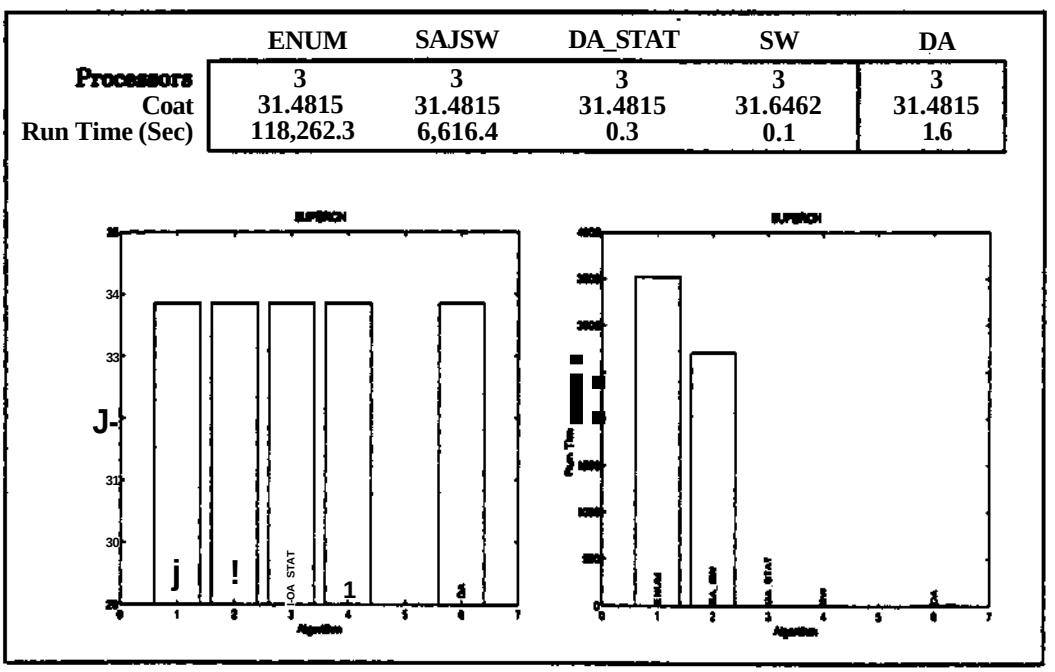
DFG Ranll:

	ENUM	SAJSW	DA.STAT	SW	PA
Processors		-	55	55	56
Cost		-	34.6124	34.6124	34.5447
Run Time (Sec)		-	271.7	2.8	960.8



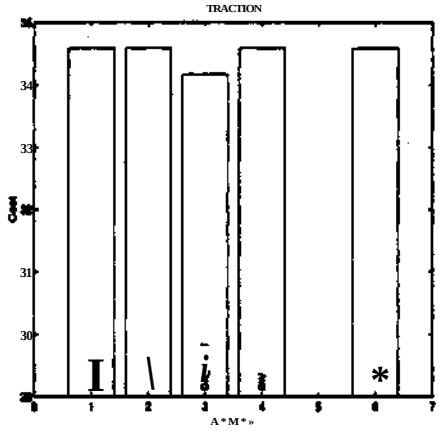
DFG Superch:

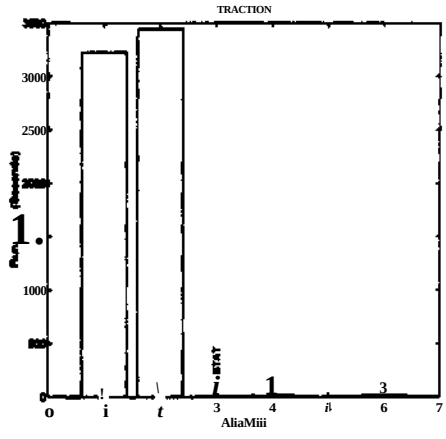
	ENUM	SAJSW	DA_STAT	SW	DA
Processors	3	3	3	3	3
Coat	31.4815	31.4815	31.4815	31.6462	31.4815
Run Time (Sec)	118,262.3	6,616.4	0.3	0.1	1.6



DFG Traction:

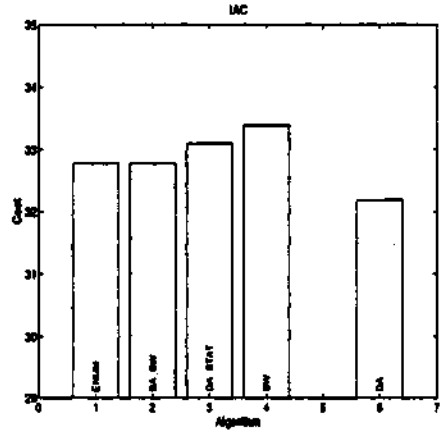
	ENUM	SAJSW	DAJSTAT	SW	DA
Processors	2	2	2	2	2
Cost	34.5848	34.58461	34.1649	34.5848	34.5848
Run Time (Sec)	3,222.2	3,447.2	2.3	12.8	17.9

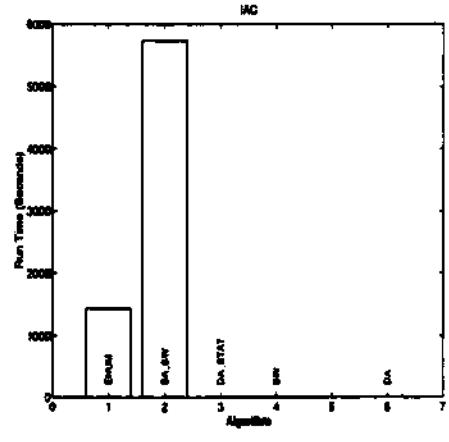




DFGIAC:

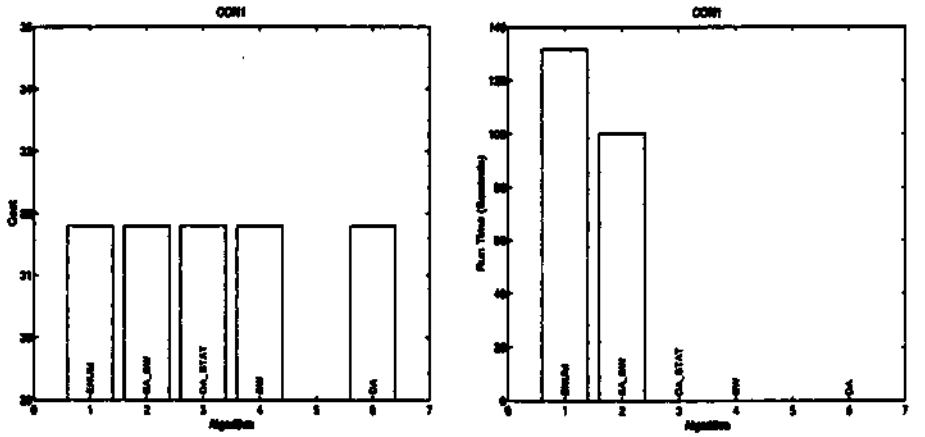
	ENUM	SAJ3W	DAJSTAT	SW	DA
Processors	2	2	2	2	8
Cost	32.7756	32.7756	33.0954	33.3883	32.1806
Run Time (Sec)	1,435.3	5,735.2	0.9	1.7	7.0





DFG Conl:

	ENUM	SAJSW	DAJSTAT	SW	DA
Processors	2	2	2	2	2
Cost	31.7934	31.7934	31.7934	31.7934	31.7934
Run Time (Sec)	131.6	100.3	< 0.1	< 0.1	0.1



DFGCon2:

	ENUM	SAJSW	DAJSTAT	SW	DA
Processors	3	3	3	3	3
Cost	31.6385	31.6385	31.6385	31.706	31.706
Run Time (Sec)	75.652.4	502.4	0.1	< 0.1	0.4