Proceedings of the 2007 IEEE Symposium on Computational Intelligence in Scheduling (CI-Sched 2007)

A Genetic Algorithm for Scheduling Parallel Non-identical Batch Processing Machines

Shubin Xu and James C. Bean

Abstract— **In this paper, we study the scheduling problem of minimizing makespan on parallel non-identical batch processing machines. We formulate the scheduling problem into an integer programming model. Due to the difficulty of the problem, it is hard to solve the problem with standard mathematical programming software. We propose a genetic algorithm based on random keys encoding to address this problem. Computational results show that this genetic algorithm consistently finds a solution in a reasonable amount of computation time.**

I. Introduction

BATCH processing machines can process a number of jobs at the same time. The rationale for batching is simple: it may be economically efficient to process jobs in batches than to process them individually. Applications of batching in industry include heat treatment operations in metalworking, diffusion or burn-in operations in semiconductor fabrication. In this paper, we focus on the scheduling problem of burn-in test operations in semiconductor fabrication. There are four major steps in large scale integrated circuits (IC) manufacturing [1]: wafer fabrication, wafer probe, assembly or packaging, and final testing. Burn-in test operations take place in the final testing step. The processing times of burn-in operations are generally extremely long compared to those of other testing operations (e.g., 120 hours vs. 4–5 hours) [2]. Therefore, the burn-in operation is frequently the bottleneck process in the final testing step and because it occurs at the end of the manufacturing process, it plays a vital role in the commitment of on-time delivery of finished products. Effective scheduling of these operations is of great importance to the overall performance of a company.

In the burn-in operation, different kinds of IC chips are loaded onto boards and then placed into an oven for burn-in test. The oven is maintained at a constant high temperature for a period of time. The burn-in time and temperature for each IC chip are determined by the product test specification and thus are known a priori. The idea for burn-in test is to expose the IC chips to thermal stress so that any chip out of test specification can be sorted out. Due to the limited capacity of the burn-in oven, the IC chips must be sub-grouped into batches. To ensure the quality of the product, the processing time of a batch is determined by the longest processing time of all the jobs contained in the batch. A chip can be kept in the oven longer than its pre-specified burn-in time, but not taken out from the oven before the pre-specified burn-in time has elapsed. Once the processing of a batch initiates, it cannot be preempted and no job can be removed from or introduced to the oven until the processing of the batch is finished. The readers are referred to [2], [3] for more details about burn-in oven test.

In this research, we consider parallel burn-in ovens with non-identical capacities. All jobs are assumed to be ready at time zero. We model burn-in oven test operations as parallel batch processing machines with non-identical capacities. The performance measure to be optimized is makespan, $C_{\max}$, defined as the time to finish all jobs. We seek to minimize makespan so that the finished products can be shipped to customer as soon as possible. Following the three-field $\alpha \,|\, \beta \,|\, \gamma$ notation of Graham *et al.* [4], we denote this scheduling problem as $Rm \,|\, batch \,|\, C_{\max}$. The problem of minimizing makespan on a single batch processing machine with non-identical job sizes is proven to be NP-hard [5], so the scheduling problem of parallel non-identical batch processing machines with non-identical job sizes under study is also NP-hard. It is inefficient, or impossible, to solve this kind of problem with standard mathematical tool. Therefore, we suggest a genetic algorithm (GA) to solve this problem in a reasonable amount of computation time.

The remainder of this paper is organized as follows. In Section II, we survey previous related work on scheduling batch processing machines. In Section III, we present a mathematical formulation for the batching problem and try to solve the problem with standard mathematical programming software. Due to the difficulty of the problem, we propose a genetic algorithm to solve the problem in Section IV. Computational results are given in Section V. Finally, conclusions and future research directions are discussed in Section VI.

II. Previous Related Work

Many researchers have dedicated their efforts to the problem of scheduling batch processing machines. Early work on batch scheduling can be traced back to Ikura and Gimple [6] who study a single batch processing machine with identical job processing times and identical job sizes. They propose an $O(n^2)$ algorithm to minimize makespan. Since then, much research has been done in this area. For an excellent review of scheduling with batching, see [7]. The literature reviewed

Manuscript received October 31, 2006.

Shubin Xu is with the School of Mechanical Engineering, Shanghai Jiao Tong University, Shanghai 200030 China. He is now visiting to Charles H. Lundquist College of Business, University of Oregon, Eugene, OR 97403 USA (e-mail: sxu@uoregon.edu).

James C. Bean is with the Charles H. Lundquist College of Business, University of Oregon, Eugene, OR 97403 USA (corresponding author, phone: 541-346-3300; fax: 541-346-3331; e-mail: jcbean@uoregon.edu).

in that paper shows that batch scheduling problems are often addressed by heuristics or dynamic programming.

Lee *et al.* [2] first study the problem of scheduling semiconductor burn-in oven. They present efficient dynamic programming-based algorithms for minimizing a number of different performance measures such as maximum tardiness, the number of tardy jobs, on a single batch processing machine.

Chandru *et al.* [8] study the problem of minimizing total completion time on single and parallel identical batch processing machines. They propose an exact branch-and-bound algorithm for the single machine scheduling problem and heuristics for the parallel machines scheduling problem. Chandru *et al.* [9] also propose a dynamic programming algorithm for minimizing total completion time on a single batch processing machine.

Uzsoy [5] studies a problem of scheduling a single batch processing machine with non-identical job sizes. He develops branch-and-bound algorithm as well as heuristics to minimize total completion time.

Uzsoy [3] studies another problem of scheduling batch processing machines with incompatible job families, where jobs from different families cannot be grouped into the same batch. He develops several algorithms to minimize makespan, maximum lateness, and total weighted completion time for single and parallel identical batch processing machines.

Mehta and Uzsoy [10] study the problem of minimizing total tardiness on a batch processing machine with incompatible job families. They provide a dynamic programming algorithm that has polynomial time complexity when the number of job families and the capacity of the batch processing machine are fixed. They also examine a number of heuristics that can provide near optimal solutions in a reasonable amount of computation time.

Lee and Uzsoy [11] consider the scheduling problem of minimizing makespan on a single batch processing machine with dynamic job arrivals. They provide polynomial and pseudo-polynomial time algorithms for several special cases. They also develop efficient heuristics and evaluate their performance through computational experiments.

Sung and Choung [12] study the scheduling problem to minimize makespan for a single burn-in oven. They analyze a static problem in which all jobs are ready at time zero, and also investigate a dynamic problem with different job ready times. They propose a branch-and-bound algorithm and a number of heuristics to solve the scheduling problem.

Chang *et al.* [13] propose a simulated annealing approach to minimize makespan for parallel identical batch processing machines.

All the papers discussed above study either a single batch processing machine or parallel identical processing machines.

III. Problem Definition

The batch scheduling problem under study involves assigning jobs to batches and determining the batch sequence on the machines so as to minimize the makespan.

We make the following assumptions about our scheduling problem:

1. There are n jobs to be processed by m parallel non-identical batch processing machines. All the data, such as job processing times, job sizes, and machine capacities are deterministic and known a priori.
2. All jobs are ready to be processed at time zero.
3. The machines are continuously available.
4. The setup times of the machines, compared with the processing times, are negligible.
5. The processing time of a batch is represented by the longest processing time of all the jobs contained in the batch. The size of the batch cannot exceed the capacity of the machine.
6. Once a batch is processed by a machine, it cannot be interrupted, i.e., no preemption is allowed. No jobs can be introduced or removed from a batch while the batch is being processed.
7. All the jobs are considered equal in importance.
8. The performance measure for the scheduling system is makespan. Our objective is to minimize the makespan.

The scheduling problem can be formulated into an integer programming model. Here is a list of notation for the mathematical model.

Problem parameters:

j	index of job, $j \in \{1, 2, \ldots\}$.
b	index of batch, $b \in \{1, 2, \ldots\}$.
m	index of machine, $m \in \{1, 2, \ldots\}$.
J	set of all jobs.
B	set of all batches.
M	set of all machines.
p_j	processing time of job j.
s_j	size of job j.
K_m	capacity of machine m.

Problem decision variables:

X_{jbm}	binary, 1 if job j is assigned to batch b and processed by machine m, 0 otherwise.
Y_{bm}	binary, 1 if batch b is open on machine m, 0 otherwise.
P_{bm}	batch processing time of batch b processed by machine m, $\in \Re^+$.
$C_{\max}$	makespan, $\in \Re^+$.

The integer programming model can be formulated as follows:

$$\text{Minimize} \quad C_{\max} \tag{1}$$

subject to:

$$\sum_{b \in B} \sum_{m \in M} X_{jbm} = 1 \qquad \forall\, j \in J \tag{2}$$

$$\sum_{m \in M} Y_{bm} \leqslant 1 \qquad \forall\, b \in B \tag{3}$$

$$\sum_{j \in J} s_j X_{jbm} \leqslant K_m Y_{bm} \qquad \forall\, b \in B, m \in M \tag{4}$$

$$P_{bm} \geqslant p_j X_{jbm} \qquad \forall\, j \in J, b \in B, m \in M \tag{5}$$

$$C_{\max} \geqslant \sum_{b \in B} P_{bm} \qquad \forall\, m \in M \tag{6}$$

$$X_{jbm} \in \{0,1\} \qquad \forall\, j \in J, b \in B, m \in M \tag{7}$$

$$Y_{bm} \in \{0,1\} \qquad \forall\, b \in B, m \in M \tag{8}$$

The objective (1) minimizes the makespan. Constraint (2) ensures that job j is assigned to exactly one batch and processed on one machine. Constraint (3) ensures that batch b can only be opened on one machine at most. Constraint (4) specifies that for each machine m, the capacity of the machine cannot be exceeded when jobs are assigned to a batch. It also enforces that a job can only be assigned to batch machine combinations that are open. Constraint (5) determines the processing time of a batch. The processing time of a batch is represented by the longest processing time of all the jobs contained in the batch. Constraint (6) determines the makespan. The makespan is equivalent to the completion time of the last batch to leave the system. Constraints (7) and (8) are binary constraints.

We try to solve this integer program with mathematical programming software ILOG CPLEX 10.1. Preliminary experiments show with some small problem sets (10 jobs, two parallel non-identical batch processing machines), CPLEX can find an optimal solution within half an hour. If we increase the number of jobs to 50, CPLEX cannot find an optimal solution within two hours, which is of little practical value. Due to the limit of the mathematical programming software, we propose a GA to address this problem.

IV. A Genetic Algorithm Approach

A. Genetic Algorithm

The origins of genetic algorithms (GAs) can be traced back to the 1950s when a number of biologists used computers to perform simulations of genetic systems [14]. However, the work done by John Holland and his students and colleagues in the 1960s and 1970s at the University of Michigan led to GAs as they are known to us today [15]. GAs are adaptive search algorithms which apply the concepts from biological systems to a mathematical context. The underlying strategy is to start with randomly generated solutions and evolve these solutions according to the principle of natural selection, i.e., *survival-of-the-fittest*. Over many generations, the solutions in the population *evolve* until the best of the population is near optimal. For additional information about GAs, we refer to standard text [14].

GAs have been applied to scheduling problems by a number of researchers during the past three decades. There have been a lot of research on job shop and other types of scheduling problems using GAs as the optimization methods. We refer to [16]–[21] for an overview of applications of scheduling with GAs in business and industry. However, Applications of GAs in batch scheduling do not seem to have been studied extensively. Wang and Uzsoy [22] combine dynamic programming algorithm with a genetic algorithm to minimize maximum lateness on a batch processing machine in the presence of dynamic job arrivals. Koh *et al.* [23] study the problems of scheduling parallel identical batch processing machines with arbitrary job sizes and incompatible job families. They consider three kinds of problems whose performance measures are makespan, total completion time, and total weighted completion time, respectively. They devise a number of heuristics and use genetic algorithms to solve the problems. Koh *et al.* [24] study the same problems of scheduling a single batch processing machines using GAs. Damodaran *et al.* [25] propose a GA to minimize makespan on a single batch processing machine with non-identical job sizes. Balasubramanian *et al.* [26] and Mönch *et al.* [27] use GAs to minimize total weighted tardiness on parallel identical batch processing machines. We are unaware of any published literature studying the problem of scheduling parallel non-identical batch processing machines using GAs.

To apply GA to a scheduling problem, a suitable encoding or representation for the problem must be devised. The chromosomal encoding of the scheduling problem may take many forms and have a direct impact on the performance of the GA. A common problem in applying GAs is that some genetic operations may create feasibility problems, e.g., crossing over two feasible solutions does not result in a feasible solution as an offspring. Given a scheduling problem, often the hardest part in applying a GA is to encode the solutions as strings so that crossovers of feasible solutions result in feasible ones.

A Random Keys Genetic Algorithm (RKGA), introduced by Bean [28], differs from traditional GAs most notably in the representation of the solution. The random keys representation encodes a solution with *random* numbers. These numbers are served as sort *keys* to decode the solution. Chromosomal encodings are constructed to represent solutions. These encodings are evaluated in the fitness evaluation function in a way that avoids the feasibility problem. RKGA has been successfully applied to various scheduling problems, vehicle routing problems, resource allocation problems, and other optimization problems [22], [28]–[34].

We use RKGA as the principal method for our scheduling problem. The RKGA operates in two spaces, the chromosome space $\mathcal{C}$ and the schedule building space $\mathcal{S}$. Random numbers are typically sampled from $[0,1]^n$. These random numbers in the chromosome space are used as tags to represent solutions. The RKGA searches the chromosome space as a surrogate for the schedule building space. Points in the chromosome space are mapped to the schedule building space based on the sorted random key values.

As an example, consider a five-job, single machine scheduling problem. Begin by generating a uniform $(0,1)$ random number for each job. This would result in a chromosome with five genes

$$(0.28, 0.15, 0.64, 0.92, 0.73).$$

Sorting the random keys in ascending order would result in the job sequence

$$2 \rightarrow 1 \rightarrow 3 \rightarrow 5 \rightarrow 4.$$

If our objective is to minimize makespan, this sequence can be evaluated by fitness evaluation function for calculating the makespan. During the running of the RKGA program, jobs that should be early in the sequence evolve small keys and jobs that should be later develop large keys.

The random keys encoding has the advantage over a literal encoding, that all crossovers produce feasible offspring. Crossovers are executed on the chromosomes, the random keys, not on the job sequences. Therefore the offspring always contain random keys that can be sorted into an ordered set. Since any ordered set of random keys can be interpreted as a job sequence, all offspring are feasible solutions. The random keys simply serve as tags that the crossover operator uses to rearrange jobs.

There are many variations of genetic operators that may be used for a GA [14], [35]. The algorithm described in this paper uses elitist reproduction, Bernoulli crossover, immigration, and post-tournament selection, as the operators described in [28], [30], to move from one generation to the next.

Elitist reproduction is accomplished by copying the best individuals from one generation to the next. In [14], this is called *elitist* strategy. This method has the advantage over traditional probabilistic reproduction in that the best solution is monotonically improving. However, the potential downside is that the population may prematurely converge to local optima. This is overcome by introducing high mutation rates.

Bernoulli crossover (called parameterized uniform crossover in [36]) is used in place of the traditional one-point or two-point crossover. Two chromosomes are selected randomly from the current population as parents. Let $P_1 = (p_{11}, p_{12}, \ldots, p_{1n})$ and $P_2 = (p_{21}, p_{22}, \ldots, p_{2n})$ be the two parent chromosomes with n random key alleles. Let $\Delta = (\delta_1, \delta_2, \ldots, \delta_n)$ be n independent uniform $(0, 1)$ random variables and P_c be the probability of crossover for each gene. Let $O_1 = (o_{11}, o_{12}, \ldots, o_{1n})$ and $O_2 = (o_{21}, o_{22}, \ldots, o_{2n})$ be the two offspring that will be produced from the crossover of the two parent chromosomes. Determine each allele in O_1 and O_2 as follows:

$$\begin{cases} o_{1i} = p_{1i} \quad \text{and} \quad o_{2i} = p_{2i} & \text{if } \delta_i \leqslant P_c, \\ o_{1i} = p_{2i} \quad \text{and} \quad o_{2i} = p_{1i} & \text{if } \delta_i > P_c. \end{cases}$$

P_c is usually selected not equal to 0.5 to bias selection toward one parent. Experiments have shown that $P_c = 0.7$ works well for our scheduling problem.

Implementing a mutation operator for a real coded GA is difficult [30]. As a result, rather than the traditional gene-by-gene mutation with very small probability (on the order of $1/1000$), we use *immigration* as described in [28]. That is, at each generation, one or more new members of the population are randomly generated, as *immigrants*, from the same distribution as the original generation. Immigration plays a secondary role in RKGA. However, it is indispensable in that it diversifies the search space and protects from loss of genetic material which can be caused by reproduction and crossover.

A *post-tournament selection* [30] is used with Bernoulli crossover to fill the next generation. The two offspring chromosomes O_1 and O_2 , generated by crossover of two parent chromosomes P_1 and P_2, are evaluated, and only the one with better fitness value is allowed to enter the next generation. The other offspring is discarded.

B. Schedule Generation Procedure

In order to apply RKGA to the batch scheduling problem, we now extend the RKGA encoding technique to the multiple machines setting, using the method described in [28]. Consider the n job, m non-identical parallel machine scheduling problem to minimize makespan. Each job is encoded as a gene. Thus a chromosome for n jobs contains n genes. To obtain the gene for each job, generate an integer randomly in $\{1, \ldots, m\}$ and add a uniform $(0, 1)$ random variable. This real number serves as the random key. In the mapping, the integer part of any random key is interpreted as the machine assignment for that job and the fractional parts of all the random keys are sorted to provide the job sequence on each machine. For each machine, form batches from the job sequence on that machine such that the total size of all the jobs in the batch is less than or equal to the capacity of the machine. The processing time of a batch is determined by the longest processing time of all the jobs contained in the batch, as described in Section III. Assuming that batches are processed at their earliest possible time, a schedule can then be constructed and evaluated for makespan.

To generalize, given a chromosome with n genes (random keys), this chromosome is mapped to the schedule building space by decoding. A schedule can be constructed and the fitness (makespan in our problem) for this schedule can be evaluated in the fitness evaluation function. The procedure for schedule generation and fitness evaluation is formalized as follows.

Procedure SCHED_GEN

1. Sort the n genes (random keys) that correspond to the n jobs in ascending order.
2. Determine the machine assignment for each job and the job sequence on each machine.
3. For each machine, form batches from the job sequence on the machine such that the capacity of the machine is not exceeded.
4. Determine the batch processing times.
5. Determine the start and complete times for each batch.
6. Calculate the makespan.

C. The Pseudo-code for the RKGA

After discussed the random keys encoding, the operators used in the RKGA, and the schedule generation procedure, a pseudo-code for the RKGA is listed in Fig. 1.

Computational experiments of the proposed algorithm have been very successful. The results are reported in the next section.

V. Computational Results

In order to evaluate the performance of the proposed RKGA for the batch scheduling problem, we conducted a number of computational experiments using randomly generated problem instances. The parameters we need to specify are the number of jobs, the number of machines, and the distribution of job processing times, job sizes, and machine capacities.

Choose parameters for the program:
maximum number of generations to be run ($N_{\max}$), population size (N_{pop}), number of chromosomes copied to next generation (N_{c}), number of immigrants (N_{m}).
Determine a stopping criterion for the program:
an objective value bound or $N_{\max}$.
for each chromosome of the population **do**
generate random keys;
evaluate fitness based on Procedure SCHED_GEN;
end for
$stop \leftarrow 0$;
$count \leftarrow 0$;
while $stop = 0$ **do**
$count \leftarrow count + 1$;
rank old population by fitness;
copy N_{c} best chromosomes to the new generation;
for remainder of the population **do**
randomly choose two parents from old population;
perform Bernoulli crossover;
evaluate fitness of the two offspring;
include the better offspring in the new generation;
discard the other offspring;
end for
replace N_{m} worst chromosomes with immigrants;
if the stopping criterion is met **then**
$stop \leftarrow 1$;
end if
end while

Fig. 1. Pseudo-code for the RKGA

Three levels of the number of jobs were selected to indicate the different sizes of the scheduling problem: small (15 jobs), medium (50 jobs), and large (100 jobs). After specifying the number of jobs for a problem instance, the processing time p_j of each job j, was randomly generated from $DU[1, 10]$, where $DU[a, b]$ denotes a discrete uniform distribution within $[a, b]$. Two levels of job sizes were selected to test the effect of small jobs ($DU[1, 4]$), and small to large jobs ($DU[2, 8]$). Two levels of the number of machines were considered, i.e., two and four machines. For each machine m, the capacity, K_m, was randomly generated from $DU[8, 12]$. Totally, there are 12 unique design factor combinations. Table I summarizes these factors and their levels and values used in the experiments.

TABLE I
Factors and Levels Used in the Experiments

Factors	Levels	Values
Number of jobs, $\|J\|$	3	15, 50, 100
Processing time, p_j	1	$DU[1, 10]$
Job size, s_j	2	$DU[1, 4]$, $DU[2, 8]$
Number of machines, $\|M\|$	2	2, 4
Machine capacity, K_m	1	$DU[8, 12]$

We grouped our experiments into six sets of randomly generated problems based on the machine setting (two or four machines) and the number of jobs in the problem. For each machine setting, there are three sets of randomly generated problems of different sizes: small (15 jobs), medium (50 jobs), and large (100 jobs). Each set contains 10 problem instances. Totally, there are 60 different problem instances.

The RKGA was coded in C++ and run on a desktop computer with a Pentium III 997 MHz CPU and 512 MB RAM. For each problem instance, 10 replications of RKGA were executed with different random seeds. Population size of 1000 was used for all the small, medium, and large problems. In all the tests, RKGA was allowed to run up to 500 generations. All the problem instances were also solved in ILOG CPLEX 10.1 on the same computer.

Preliminary experiments indicated that for some problem instances, after two hours of run, CPLEX did not end up with optimal solutions or with very little improvements in the solutions. Therefore, it was decided to run CPLEX up to one hour, i.e., 3600.00 seconds, for all the problem instances. In this case, we may call the method we are using as CPLEX *heuristic*, or simply CPLEX-H.

Table II and III present the computational results from the RKGA and CPLEX-H on 60 different problem instances. Table II shows the computational results for two parallel machines while Table III for four parallel machines. In the tables, column 1 indicates the code for the problem instances. Initials `s`, `m`, and `l` stand for small, medium, and large problems, respectively. In Column 2, a triplet (J_μ, s_ν, M_ω), where $\mu \in \{1, 2, 3\}$, $\nu \in \{1, 2\}$, and $\omega \in \{1, 2\}$, is used to denote the size of the problem. For example, (J_3, s_2, M_1) specifies that the number of jobs is generated at level 3 (100 jobs), the job sizes at level 2 ($DU[2, 8]$), and the number of machines at level 1 (two machines). Columns 3–8 report the results from RKGA in terms of the $C_{\max}$ and the run time (in seconds) for the test. For each problem instance, the RKGA was run 10 times, using a different random seed for each run. The minimum, median, and maximum values over the 10 different random seeds are reported. The minimum and maximum values provide best and worst case of the performance measures (here, $C_{\max}$ and run time). The median is used rather than the mean in order to reduce the influence of extreme minimum and maximum values. Columns 9 and 10 report the results at which generation the best solution ($C_{\max}$) was first found and the seconds required to get this solution. The results are averaged over 10 different random

TABLE II
COMPUTATIONAL RESULTS FOR RANDOMLY GENERATED PROBLEMS WITH TWO MACHINES

Problem	Size	RKGA†								CPLEX-H‡		Improv.
		$C_{\max}$			Seconds			Best Soln.		$C_{\max}$	Seconds	RKGA vs. CPLEX-H (%)
		Min.	Med.	Max.	Min.	Med.	Max.	Gen.	Sec.			
(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)
sa1	(J_1, s_1, M_1)	20	20	20	7.80	7.88	8.19	10	0.19	20*	3600.00	0.00
sa2	(J_1, s_1, M_1)	12	12	12	7.72	7.77	7.88	11	0.19	12	24.42	0.00
sa3	(J_1, s_1, M_1)	13	13	13	7.70	7.78	7.98	4	0.08	13*	3600.00	0.00
sa4	(J_1, s_1, M_1)	16	16	17	7.99	8.08	8.19	27	0.47	16*	3600.00	0.00
sa5	(J_1, s_1, M_1)	14	14	14	7.66	7.82	8.13	11	0.20	14	249.24	0.00
sa6	(J_1, s_2, M_1)	17	17	17	8.14	8.27	8.44	18	0.32	17*	3600.00	0.00
sa7	(J_1, s_2, M_1)	26	26	26	8.54	8.61	9.04	14	0.26	26*	3600.00	0.00
sa8	(J_1, s_2, M_1)	26	26	26	8.64	8.77	9.06	8	0.16	26*	3600.00	0.00
sa9	(J_1, s_2, M_1)	23	23	23	8.12	8.27	8.56	15	0.27	23*	3600.00	0.00
sa10	(J_1, s_2, M_1)	29	29	29	8.57	8.65	8.82	5	0.10	29*	3600.00	0.00
ma1	(J_2, s_1, M_1)	31	32	33	29.95	30.10	30.39	120	7.35	34*	3600.00	8.82
ma2	(J_2, s_1, M_1)	33	34	35	30.22	30.49	31.25	118	7.37	36*	3600.00	8.33
ma3	(J_2, s_1, M_1)	36	38	39	31.16	31.43	31.96	164	10.51	42*	3600.00	14.29
ma4	(J_2, s_1, M_1)	36	38	39	30.51	30.74	31.56	135	8.49	42*	3600.00	14.29
ma5	(J_2, s_1, M_1)	37	38	38	30.28	30.61	31.29	132	8.19	41*	3600.00	9.76
ma6	(J_2, s_2, M_1)	88	88	90	32.11	32.52	33.28	185	12.34	97*	3600.00	9.28
ma7	(J_2, s_2, M_1)	72	74	76	32.61	33.09	34.68	144	9.72	83*	3600.00	13.25
ma8	(J_2, s_2, M_1)	70	70	71	31.90	32.33	32.57	188	12.39	84*	3600.00	16.67
ma9	(J_2, s_2, M_1)	61	62	63	32.28	32.41	34.40	154	10.25	77*	3600.00	20.78
ma10	(J_2, s_2, M_1)	75	76	77	32.18	32.40	32.88	187	12.39	88*	3600.00	14.77
la1	(J_3, s_1, M_1)	76	78	79	66.55	67.56	68.50	268	36.32	96*	3600.00	20.83
la2	(J_3, s_1, M_1)	76	77	77	66.85	67.33	68.53	262	35.73	97*	3600.00	21.65
la3	(J_3, s_1, M_1)	66	68	69	65.82	66.56	68.19	317	42.50	87*	3600.00	24.14
la4	(J_3, s_1, M_1)	76	77	78	66.67	66.97	68.81	285	38.65	96*	3600.00	20.83
la5	(J_3, s_1, M_1)	86	87	89	67.26	67.46	69.67	294	40.05	111*	3600.00	22.52
la6	(J_3, s_2, M_1)	125	126	129	70.24	70.72	72.33	271	38.75	160*	3600.00	21.88
la7	(J_3, s_2, M_1)	121	122	124	69.06	69.34	71.75	322	45.27	150*	3600.00	19.33
la8	(J_3, s_2, M_1)	154	154	157	70.77	71.18	73.83	235	34.03	194*	3600.00	20.62
la9	(J_3, s_2, M_1)	142	144	146	69.96	70.37	71.32	265	37.61	174*	3600.00	18.39
la10	(J_3, s_2, M_1)	143	145	146	70.51	70.80	73.56	260	37.49	172*	3600.00	16.86

† Computational effort over 10 different random seeds for each problem instance.
‡ Since CPLEX is an optimal algorithm, we may call it CPLEX *heuristic* or simply CPLEX-H when we shut off CPLEX at 3600.00 seconds.
* Results reported after 3600.00 seconds. CPLEX did not report it had found the optimal solution.

seeds. Columns 11 and 12 report the $C_{\max}$ and run times from CPLEX-H. Column 13 presents the improvements realized through RKGA over CPLEX-H in terms of the $C_{\max}$. The percentage improvements are calculated by $[C_{\max}(\text{CPLEX-H}) - C_{\max}(\text{RKGA})]/C_{\max}(\text{CPLEX-H})$.

As seen in Table II and III, RKGA requires less computation times compared with CPLEX-H for all the problem instances. In the mean time, RKGA obtains the same (for small problems) or better (for medium and large problems) $C_{\max}$ compared with CPLEX-H. For some of the small problems, e.g., sa2, sa5, sb1, sb2, sb4, sb8, and sb9, CPLEX-H can optimally solve them within 3600.00 seconds. For these problems, the same optimal solutions can also be achieved by RKGA, but with less computation times. It demonstrates that for these small problems RKGA achieves an improvement over CPLEX-H in that RKGA solves the problems with same or better solutions in shorter computation times. The improvement in computation time is more pronounced for medium and large problems. For these medium and large problems, RKGA can find better solutions in shorter computation times. Also note that all large problems, medium problems, and even some of the small problems could not be optimally solved by CPLEX-H, typically due to the extra long run times and lack of the memory storage of the computer if we wanted to solve the problems optimally. Another important thing is that within the same problem set (small, medium, or large), the computation

TABLE III
COMPUTATIONAL RESULTS FOR RANDOMLY GENERATED PROBLEMS WITH FOUR MACHINES

Problem	Size	RKGA†								CPLEX-H‡		Improv.
		$C_{\max}$			Seconds			Best Soln.		$C_{\max}$	Seconds	RKGA vs. CPLEX-H (%)
		Min.	Med.	Max.	Min.	Med.	Max.	Gen.	Sec.			
(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)
sb1	(J_1, s_1, M_2)	10	10	10	8.80	8.90	9.20	1	0.03	10	487.90	0.00
sb2	(J_1, s_1, M_2)	10	10	10	8.79	8.90	9.32	1	0.03	10	164.90	0.00
sb3	(J_1, s_1, M_2)	11	11	11	8.85	8.92	9.10	32	0.61	11*	3600.00	0.00
sb4	(J_1, s_1, M_2)	10	10	10	8.63	8.80	8.94	7	0.15	10	46.41	0.00
sb5	(J_1, s_1, M_2)	11	11	12	8.90	9.05	9.52	35	0.68	11*	3600.00	0.00
sb6	(J_1, s_2, M_2)	15	15	15	9.42	9.51	9.72	12	0.24	15*	3600.00	0.00
sb7	(J_1, s_2, M_2)	13	14	14	9.33	9.48	10.39	53	1.11	13*	3600.00	0.00
sb8	(J_1, s_2, M_2)	10	10	11	9.05	9.33	9.57	80	1.54	10	538.37	0.00
sb9	(J_1, s_2, M_2)	10	10	11	9.16	9.33	9.66	24	0.48	10	50.69	0.00
sb10	(J_1, s_2, M_2)	15	15	16	9.38	9.57	9.86	34	0.69	15*	3600.00	0.00
mb1	(J_2, s_1, M_2)	18	19	19	33.75	33.97	35.44	128	8.95	20*	3600.00	10.00
mb2	(J_2, s_1, M_2)	21	21	22	34.01	34.33	34.80	136	9.58	23*	3600.00	8.70
mb3	(J_2, s_1, M_2)	17	18	19	33.31	33.61	34.46	131	8.97	21*	3600.00	19.05
mb4	(J_2, s_1, M_2)	21	22	23	33.63	34.20	34.56	168	11.68	25*	3600.00	16.00
mb5	(J_2, s_1, M_2)	17	17	18	33.70	33.88	35.53	119	8.26	19*	3600.00	10.53
mb6	(J_2, s_2, M_2)	37	38	39	36.41	37.17	38.44	172	12.98	44*	3600.00	15.91
mb7	(J_2, s_2, M_2)	38	39	40	36.42	36.57	37.77	163	12.22	41*	3600.00	7.32
mb8	(J_2, s_2, M_2)	39	41	42	36.24	36.66	37.99	181	13.63	46*	3600.00	15.22
mb9	(J_2, s_2, M_2)	30	31	32	35.95	36.18	38.19	176	13.11	37*	3600.00	18.92
mb10	(J_2, s_2, M_2)	31	31	32	35.83	36.07	36.49	183	13.44	35*	3600.00	11.43
lb1	(J_3, s_1, M_2)	36	37	39	72.84	73.20	75.78	284	42.07	47*	3600.00	23.40
lb2	(J_3, s_1, M_2)	39	40	42	73.37	73.88	75.44	288	42.96	56*	3600.00	30.36
lb3	(J_3, s_1, M_2)	35	36	38	72.67	73.26	75.45	250	37.07	46*	3600.00	23.91
lb4	(J_3, s_1, M_2)	37	38	39	73.21	73.99	77.01	291	43.51	57*	3600.00	35.09
lb5	(J_3, s_1, M_2)	32	34	34	72.30	73.21	74.46	237	34.93	40*	3600.00	20.00
lb6	(J_3, s_2, M_2)	68	69	71	77.52	78.31	79.65	290	45.71	92*	3600.00	26.09
lb7	(J_3, s_2, M_2)	87	88	89	77.99	78.64	80.73	325	51.63	120*	3600.00	27.50
lb8	(J_3, s_2, M_2)	72	73	74	77.15	77.86	79.58	316	49.57	91*	3600.00	20.88
lb9	(J_3, s_2, M_2)	84	87	88	77.84	78.78	79.56	312	49.36	124*	3600.00	32.26
lb10	(J_3, s_2, M_2)	69	71	72	76.96	77.65	79.41	334	52.16	88*	3600.00	21.59

† Computational effort over 10 different random seeds for each problem instance.
‡ Since CPLEX is an optimal algorithm, we may call it CPLEX *heuristic* or simply CPLEX-H when we shut off CPLEX at 3600.00 seconds.
* Results reported after 3600.00 seconds. CPLEX did not report it had found the optimal solution.

times of different problem instances for RKGA are consistent, whereas the computation times for CPLEX-H are inconsistent and the ranges of the computation times are unpredictable. For example, look at problems `sa1-10` in Table II, all the computation times for RKGA are around 8–9 seconds. For CPLEX-H, the computation time for `sa2` is 24.42 seconds, `sa5` 249.24 seconds, and others more than 3600.00 seconds.

The computational results also demonstrate two important merits of RKGA. First, computation time increases in a reasonable manner as problem size increases. Second, RKGA produces consistent results across different random seeds. The RKGA appears to be very robust for the batch scheduling problem.

VI. CONCLUSIONS AND FUTURE WORK

We propose a genetic algorithm that uses the random keys encoding to solve the scheduling problem of minimizing makespan on parallel non-identical batch processing machines. Computational experiments indicate that the RKGA found solutions to all the randomly generated problems under the predefined conditions. The solutions and run times from RKGA were compared with those from CPLEX-H. It shows that RKGA outperformed CPLEX-H in terms of solutions and computation times, especially for larger problems. It was also observed that RKGA was very robust. The RKGA produced consistent solutions across different random seeds in reasonable computation times. We shall point out, however, for

some of the test problems (especially for medium and large problems), we do not know the optimal solution and hence cannot verify the quality of the RKGA solution.

A number of directions for future research exist. A tight value bound will not only reduce the CPU time for RKGA (e.g., RKGA may be terminated when a solution within five percent of the value bound is discovered), but also serve as a benchmark for evaluating the proposed algorithms. Though a genetic algorithm is explored in this study, the possibility of using other heuristics may be examined. Finally, extensions of the RKGA methodology to other problem complexities will be explored. Here we list three possible extensions. The first extension relaxes the assumption that all jobs are ready at time zero. This relaxation says the jobs have non-zero ready times, which is more reasonable in the real world. The second extension is relaxing the assumption that the setup times of the machines are negligible. We will consider sequence-dependent setup times. The third extension is to include other performance measures such as total tardiness. The results will be reported elsewhere.

ACKNOWLEDGMENT

The authors would like to thank the two anonymous referees whose comments improved the quality of this manuscript. Shubin Xu also wishes to thank Charles H. Lundquist College of Business at the University of Oregon for providing excellent research resources during his visit.

REFERENCES

[1] R. Uzsoy, C. Y. Lee, and L. A. Martin-Vega, "A review of production planning and scheduling models in the semiconductor industry Part I: System characteristics, performance evaluation and production planning," *IIE Trans.*, vol. 24, pp. 47–60, 1992.

[2] C. Y. Lee, R. Uzsoy, and L. A. Martin-Vega, "Efficient algorithms for scheduling semiconductor burn-in operations," *Oper. Res.*, vol. 40, pp. 764–775, 1992.

[3] R. Uzsoy, "Scheduling batch processing machines with incompatible job families," *Int. J. Prod. Res.*, vol. 33, pp. 2685–2708, 1995.

[4] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan, "Optimization and approximation in deterministic sequencing and scheduling: A survey," *Ann. Discrete Math.*, vol. 5, pp. 287–326, 1979.

[5] R. Uzsoy, "Scheduling a single batch processing machine with nonidentical job sizes," *Int. J. Prod. Res.*, vol. 32, pp. 1615–1635, 1994.

[6] Y. Ikura and M. Gimple, "Efficient scheduling algorithms for a single batch processing machine," *Oper. Res. Lett.*, vol. 5, pp. 61–65, 1986.

[7] C. N. Potts and M. Y. Kovalyov, "Scheduling with batching: a review," *Eur. J. Oper. Res.*, vol. 120, pp. 228–249, 2000.

[8] V. Chandru, C. Y. Lee, and R. Uzsoy, "Minimizing total completion time on batch processing machines," *Int. J. Prod. Res.*, vol. 31, pp. 2097–2121, 1993.

[9] ——, "Minimizing total completion time on a batch processing machine with job families," *Oper. Res. Lett.*, vol. 13, pp. 61–65, 1993.

[10] S. V. Mehta and R. Uzsoy, "Minimizing total tardiness on a batch processing machine with incompatible job families," *IIE Trans.*, vol. 30, pp. 165–178, 1998.

[11] C. Y. Lee and R. Uzsoy, "Minimizing makespan on a single batch processing machine with dynamic job arrivals," *Int. J. Prod. Res.*, vol. 37, pp. 219–236, 1999.

[12] C. S. Sung and Y. I. Choung, "Minimizing makespan on a single burn-in oven in semiconductor manufacturing," *Eur. J. Oper. Res.*, vol. 120, pp. 559–574, 2000.

[13] P. Y. Chang, P. Damodaran, and S. Melouk, "Minimizing makespan on parallel batch processing machines," *Int. J. Prod. Res.*, vol. 42, pp. 4211–4220, 2004.

[14] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*. Reading, MA: Addison-Wesley, 1989.

[15] D. Dasgupta and Z. Michalewicz, "Evolutionary algorithms — an overview," in *Evolutionary Algorithms in Engineering Applications*, D. Dasgupta and Z. Michalewicz, Eds. Berlin, Germany: Springer-Verlag, 1997.

[16] R. Cheng, M. Gen, and Y. Tsujimura, "A tutorial survey of job-shop scheduling problems using genetic algorithms, part I: Representation," *Comput. Ind. Eng.*, vol. 30, pp. 983–997, 1996.

[17] ——, "A tutorial survey of job-shop scheduling problems using genetic algorithms, part II: Hybrid genetic search strategies," *Comput. Ind. Eng.*, vol. 36, pp. 343–364, 1999.

[18] C. R. Reeves, "Genetic algorithms for the operations researcher," *INFORMS J. Comput.*, vol. 9, pp. 231–250, 1997.

[19] A. S. Jain and S. Meeran, "Deterministic job-shop scheduling: Past, present and future," *Eur. J. Oper. Res.*, vol. 113, pp. 390–434, 1999.

[20] C. Dimopoulos and A. M. S. Zalzala, "Recent developments in evolutionary computation for manufacturing optimization: Problems, solutions, and comparisons," *IEEE Trans. Evol. Comput.*, vol. 4, pp. 93–113, July 2000.

[21] H. Aytug, M. Khouja, and F. E. Vergara, "Use of genetic algorithms to solve production and operations management problems: a review," *Int. J. Prod. Res.*, vol. 41, pp. 3955–4009, 2003.

[22] C. S. Wang and R. Uzsoy, "A genetic alogrithm to minimize maximize lateness on a batch processing machine," *Comput. Oper. Res.*, vol. 29, pp. 1621–1640, 2002.

[23] S. G. Koh, P. H. Koo, J. W. Ha, and W. S. Lee, "Scheduling parallel batch processing machines with arbitrary job sizes and incompatible job families," *Int. J. Prod. Res.*, vol. 42, pp. 4091–4107, 2004.

[24] S. G. Koh, P. H. Koo, D. C. Kim, and W. S. Hur, "Scheduling a single batch processing machine with arbitrary job sizes and incompatible job families," *Int. J. Prod. Econ.*, vol. 98, pp. 81–96, 2005.

[25] P. Damodaran, P. K. Manjeshwar, and K. Srihari, "Minimizing makespan on a batch-processing machine with non-identical job sizes using genetic algorithms," *Int. J. Prod. Econ.*, vol. 103, pp. 882–891, 2006.

[26] H. Balasubramanian, L. Mönch, J. Fowler, and M. Pfund, "Genetic algorithm based scheduling of parallel batch machines with incompatible job families to minimize total weighted tardiness," *Int. J. Prod. Res.*, vol. 42, pp. 1621–1638, 2004.

[27] L. Mönch, H. Balasubramanian, J. W. Fowler, and M. E. Pfund, "Heuristic scheduling of jobs on parallel batch machines with incompatible job families and unequal ready times," *Comput. Oper. Res.*, vol. 32, pp. 2731–2750, 2005.

[28] J. C. Bean, "Genetic algorithms and random keys for sequencing and optimization," *ORSA J. Comput.*, vol. 6, pp. 154–160, 1994.

[29] B. A. Norman and J. C. Bean, "Random keys genetic algorithm for job shop scheduling," *Eng. Des. Autom.*, vol. 3, pp. 145–156, 1997.

[30] ——, "A genetic algorithm methodology for complex scheduling problems," *Nav. Res. Logist.*, vol. 46, pp. 199–211, 1999.

[31] ——, "Scheduling operations on parallel machine tools," *IIE Trans.*, vol. 32, pp. 449–459, 2000.

[32] M. E. Kurz and R. G. Askin, "Scheduling flexible flow lines with sequence-dependent setup times," *Eur. J. Oper. Res.*, vol. 159, pp. 66–82, 2004.

[33] J. F. Gonçalves, J. J. de Magalhães Mendes, and M. G. C. Resende, "A hybrid genetic algorithm for the job shop scheduling problem," *Eur. J. Oper. Res.*, vol. 167, pp. 77–95, 2005.

[34] L. Snyder and M. Daskin, "A random-key genetic algorithm for the generalized traveling salesman problem," *Eur. J. Oper. Res.*, vol. 174, pp. 38–53, 2006.

[35] Z. Michalewicz, *Genetic Algorithms + Data Structures = Evolution Programs*, 3rd ed. New York, NY: Springer-Verlag, 1996.

[36] W. M. Spears and K. A. De Jong, "On the virtues of parameterized uniform crossover," in *Proc. Fourth Int. Conf. Genetic Algorithms*, San Diego, CA, July 13–16, 1991, pp. 230–236.