Scheduling Semiconductor Burn-In Operations to Minimize Total Flowtime

Dorit S. Hochbaum, Dan Landy,

SCHEDULING SEMICONDUCTOR BURN-IN OPERATIONS TO MINIMIZE TOTAL FLOWTIME

DORIT S. HOCHBAUM AND DAN LANDY

University of California, Berkeley, California

(Received August 1994; revisions received June 1995; accepted November 1995)

This paper addresses a problem of batch scheduling which arises in the burn-in stage of semiconductor manufacturing. Burn-in ovens are modeled as batch-processing machines which can handle up to B jobs simultaneously. The processing time of a batch is equal to the longest processing time among the jobs in the batch. The scheduling problem involves assigning jobs to batches and determining the batch sequence so as to minimize the total flowtime. In practice, there is a small number m of distinct job types. Previously, the only solution techniques known for the single-machine version of this problem were an $O(m^3B^{m+1})$ pseudopolynomial algorithm, and a branch-and-bound procedure. We present an algorithm with a running time of $O(m^2 3^m)$, which is independent of B, the maximum batch size. We also present a polynomial heuristic for the general problem (when m is not fixed), which is a two-approximation algorithm. For any problem instance, this heuristic provides a solution at least as good as that given by previously known heuristics. Finally, we address the m-type problem on parallel machines, providing an exact pseudopolynomial algorithm and a polynomial approximation algorithm with a performance guarantee of $(1 + \sqrt{2})/2$.

This paper addresses a batch scheduling problem that arises in the manufacture of integrated circuits. The final stage in the production of circuits is the burn-in operation, in which chips are loaded onto boards which are then placed in an oven and exposed to high temperatures. The purpose of this operation is to subject the circuits to thermal stress, thereby weeding out the chips which might experience an early failure under normal operating conditions.

Each chip has a prespecified minimum burn-in time, which may depend on its type and/or the customer's requirements. The burn-in oven has a limited capacity, so the circuit boards holding the chips must be divided into batches. Since chips may stay in the oven for a period longer than their minimum required burn-in time, it is possible to place different products in the oven simultaneously. The processing time of each batch will therefore be equal to the longest minimum exposure time among all the products in the batch. Scheduling burn-in operations thus involves assigning jobs to batches and deciding upon the sequence in which the batches will be processed so as to optimize some objective function.

In this paper we consider the problem of scheduling to minimize total flowtime (i.e., the sum of job completion times). This objective, which is equivalent to minimizing the average time spent in the system by a job, increases throughput and reduces work-in-process inventories. This is especially important in the scheduling of burn-in operations, which are often a bottleneck in the final stage of semiconductor production due to their long processing times relative to the other testing operations.

Chandru et al. (1993a) proposed a branch-and-bound algorithm to solve the problem exactly, but the procedure is only effective for small problem instances. They also described two heuristics which find good approximate solutions in pseudopolynomial time, and showed how to extend these heuristics to the parallel machines burn-in problem.

Chandru et al. (1993b) also considered a restricted version of the problem on a single machine, in which there is a fixed number of job types, and jobs of the same type have the same processing time. We shall refer to this problem as the m-type burn-in problem. The assumption of a fixed number of job types accurately describes the real world of semiconductor manufacturing, since a typical facility produces fewer than 10 distinct circuit types at a time. Chandru et al. provided a dynamic programming algorithm (henceforth referred to as the CLU algorithm) with running time $O(m^3B^{m+1})$, where m is the number of job types and B is the maximum batch size (oven capacity). Although the authors consider m to be fixed, their algorithm still depends upon the value of B, making it pseudopolynomial.

The usefulness of the CLU algorithm in practice depends heavily on the size of B, the oven capacity. The authors assume that the manner in which chips are loaded onto circuit boards is given, so that a board is taken to be a single job (with processing time determined by the chip on the board with the longest required burn-in time), and the capacity B is defined as the number of boards that fit in the oven. Even with this simplified model, the fact that the capacity of a typical burn-in oven is between 100 and 200 boards means that the running time of $O(m^3B^{m+1})$ is practical only for small values of m. If the assumption is dropped—that is, if individual chips are considered as

Subject classifications: Production/scheduling, applications: semiconductor burn-in operations. Dynamic programming, applications. scheduling semiconductor burn-in operations
Area of review: Manufacturing, Operations and Scheduling.

jobs—then B, the oven capacity, is a number in the thousands.

The only other exact solution procedure for the burn-in problem is the branch-and-bound procedure described by Chandru et al. (1993a). The running time of this procedure depends both on B and n, the total number of jobs, and computational results indicate that the procedure is only effective for solving problem with a small number (30 or fewer) of jobs.

In this paper we present an algorithm for the m-type burn-in problem which has a running time of $O(m^2 3^m)$. This running time is independent of B, and may therefore be used to model the more open-ended scenario in which chips can be placed on boards as desired. The running time is also independent of n, and may thus be used to solve problems with any number of jobs. We present computational results that verify that the algorithm is quite fast even for large problems.

We also consider the general burn-in problem on a single machine, in which the number of distinct job types is not fixed. Although the complexity of this problem is unknown, it can be solved in polynomial time when $B = 2$ (Hochbaum and Landy 1995). We present a dynamic programming based heuristic for the general problem which guarantees a solution that is at most twice the value of the optimal solution. Furthermore, the heuristic always finds a solution at least as good as those provided by the heuristics of Chandru et al. (1993a), and empirical tests indicate that it provides significantly better solutions in practice. Finally, we consider the m-type burn-in problem on parallel machines, and show that it can be solved exactly by a pseudo-polynomial algorithm, or solved within a factor of $(1 + \sqrt{2})/2$ of the optimal solution in polynomial time.

Related research has been done by Lee et al. (1992), who provided dynamic programming algorithms for the single machine burn-in scheduling problem with several objective functions, including minimizing the maximum lateness and minimizing the number of tardy jobs, under various assumptions about processing times, release times, and due dates. A survey of other planning and scheduling problems in the semiconductor industry can be found in (Uszoy et al. 1990).

Other authors have examined various types of scheduling problems on a single batch processing machine. In one such problem, the processing time of each batch depends on a fixed setup time and on the sum of the processing times of the jobs in the batch. This problem has been addressed with the objective function of minimizing the sum of (weighted) completion times by Dobson et al. (1987), Coffman et al. (1989), Albers and Brucker (1993), Brucker (1991), Coffman et al. (1990), Naddef and Santos (1988), and Shallcross (1992). The objective of minimizing the weighted number of tardy jobs was analyzed by Hochbaum and Landy (1994). In a batch scheduling problem described by Ikura and Gimple (1986), all jobs have equal processing times and agreeable release times and deadlines, and the processing time of a batch is independent of the number of jobs in the batch. Ahmadi et al. (1992) study a flowshop with two or three machines in which one machine is a batch processor and the batch processing time is the same for all jobs. Glassey and Weng (1991) present heuristics for the problem of minimizing total completion time on a batch processing machine when jobs arrive dynamically over time.

Several researchers have addressed the problem of scheduling jobs on parallel machines to minimize the weighted flowtime, which is closely related to the parallel machine burn-in problem. The parallel machine weighted flowtime problem was shown to be NP-hard by Bruno et al. (1974), and pseudopolynomial algorithms were described by Lawler and Moore (1969), Rothkopf (1965), and Lee and Uzsoy (1992). Kawaguchi and Kyan (1986) presented an $O(n \log n)$ time approximation algorithm with a worst-case performance ratio of $(\sqrt{2} + 1)/2 \approx 1.2$. We shall make use of some of these results in our analysis of the parallel machine burn-in problem.

1. THE m-TYPE BURN-IN PROBLEM

Suppose there are m job types numbered $1 \ldots m$, and for each type $t \in \{1 \ldots m\}$ there are n_t jobs, each having processing time p_t, where $p_1 < p_2 < \cdots < p_m$. We will say that type t is *smaller* than type w if $t < w$ (i.e., if $p_t < p_w$). Let B be the maximum batch size. A schedule $S = (B_1, \ldots, B_k)$ is a sequence of batches, where each batch B_j is a set containing $|B_j|$ jobs, with $|B_j| \leq B$ for each $j \in \{1 \ldots k\}$. The processing time of batch B_j, denoted by $p(B_j)$, is equal to the largest processing time among the jobs in the batch:

$$p(B_j) = \max\{p_t : \text{batch } B_j \text{ contains a job of type } t\}.$$

The *cost* of schedule $S = (B_1, \ldots, B_k)$, denoted by $C(S)$, is the sum of job completion times. Since the completion time of each job is simply the completion time of the batch in which it appears, the cost of the schedule is given by:

$$C(S) = \sum_{i=1}^{k} |B_i| \left(\sum_{j=1}^{i} p(B_j) \right).$$

The *m-type burn-in problem* is to find a schedule S that minimizes $C(S)$.

Example 1. Consider an instance with $B = 3$, and four types of jobs:

Type	p_t	n_t
1	3	1
2	5	1
3	8	2
4	9	2

Representing jobs by their processing times, we can write the optimal schedule as the following sequence of three batches: {3, 5}, {8, 9, 9}, {8}. There are two jobs with a completion time of 5, three jobs with a completion time of 14, and one job with a completion time of 22, so the cost of this schedule is $2 \cdot 5 + 3 \cdot 14 + 1 \cdot 22 = 74$.

Throughout the paper we will make use of an alternative method for calculating the cost of a schedule. For a schedule $S = (B_1, \ldots, B_k)$, we may calculate the cost of S by multiplying each batch processing time by the number of jobs which are either in the batch or in one of the subsequent batches, as follows:

$$C(S) = \sum_{i=1}^{k} p(B_i) \left(\sum_{j=i}^{k} |B_j| \right).$$

Each of the k terms in this sum represents the contribution that the processing time of a batch makes to the completion times of all the jobs that follow this batch, inclusive.

Applying this method of calculation to the numerical example above, we see that the first batch contributes a processing time of 5 to a total of six jobs, the second batch contributes a processing time of 9 to four jobs, and the last batch contributes a processing time of 8 to one job. The cost of the schedule is therefore $5 \cdot 6 + 9 \cdot 4 + 8 \cdot 1 = 74$.

Using the notation introduced by Chandru et al. (1993b), a batch is called *full* if it contains B jobs, and *homogeneous* if it contains jobs of a single type. We expand this terminology as follows. A batch will be called *t-pure* if it is homogeneous with jobs of type t. A batch that is not full will be called a *partial* batch. Finally, if $p_t = p(B_j)$ is the largest processing time in a batch B_j, t is called the *dominant type* of B_j; alternatively, we say type t *dominates* batch B_j.

1.1. Structural Properties of Optimal Schedules

Here we state several structural properties of optimal schedules which were proved by Chandru et al. (1993b). These are then augmented with some further properties which will provide the basis for the algorithm that follows.

Lemma 1 (CLU). *Suppose a schedule $S = (B_1, \ldots, B_k)$ has k batches, where $p(B_i)$ and $|B_i|$ are the processing time and the number of jobs in batch B_i, respectively. Then the sequence of batches is optimal if*

$$\frac{p(B_1)}{|B_1|} \leqslant \frac{p(B_2)}{|B_2|} \leqslant \cdots \leqslant \frac{p(B_k)}{|B_k|}.$$

Adopting the terminology of Chandru et al. (1993b), we say that a sequence of batches which satisfies the preceding property is in batch weighted shortest processing time (BWSPT) order.

Lemma 2 (CLU). *There exists an optimal solution containing $\lfloor n_t/B \rfloor$ homogeneous and full batches of type t, and these batches will be sequenced consecutively, for each $t \in \{1 \ldots m\}$.*

By Lemma 2, it may be possible to immediately schedule some jobs in full homogeneous batches. Assuming this has been done, there are $n_t \pmod B$ jobs remaining for each $t \in \{1 \ldots m\}$. These jobs will be referred to as *leftover jobs*, and the batches that contain these job will be referred to as *leftover batches*. The following lemma states that there is always an optimal schedule in which all leftover jobs in the same batch are consecutive with respect to their processing times.

Lemma 3 (CLU). *There exists an optimal solution in which the jobs in each leftover batch are consecutive with respect to processing times, that is: for any three leftover jobs with processing times $p_i \leqslant p_j \leqslant p_k$, if a batch contains the jobs with processing times p_i and p_k, it must also contain the job with processing time p_j.*

It is easy to verify that the preceding structural properties are not mutually exclusive; that is, there always exists an optimal solution that satisfies all of the properties. Using only these properties, Chandru et al. (1993b) devised a dynamic programming algorithm that solves the burn-in problem in $O(m^3B^{m+1})$ operations. Before presenting a faster algorithm, we must prove some additional structural properties of the leftover batches in optimal solutions.

Lemma 4. *In every optimal schedule that satisfies the properties of Lemmas* 1–3, *each job type $t \in \{1 \ldots m\}$ dominates at most one leftover batch.*

Proof. Suppose that type t dominates two different leftover batches in an optimal schedule that satisfies the properties of Lemmas 1–3. Denote the batches by B_1 and B_2, and suppose that B_1 precedes B_2 in the schedule. If $|B_1| + |B_2| \leqslant B$, then the cost of the schedule can be strictly decreased by moving all of the jobs in B_2 into B_1, and eliminating batch B_2, thus contradicting the optimality of the schedule. Suppose $|B_1| + |B_2| > B$. Then either B_1 or B_2 must contain some jobs of a type other than t. (Recall that there are $n_t \bmod B$ leftover jobs of type t.) The fact that type t dominates both batches, together with the consecutivity property of Lemma 3, implies that *exactly one* of B_1 and B_2 contains jobs that are not of type t, and these jobs have type(s) smaller than t (i.e., these jobs have shorter processing times). Consider two possible cases:

Case 1. B_2 is t-pure; B_1 contains jobs of smaller type(s). The schedule may be improved in two steps. First move type-t jobs from B_2 into B_1 until B_1 is full (or do nothing if B_1 is already full). If any jobs are moved then the cost of the schedule is strictly decreased, because the completion times of the moved jobs are decreased while the completion times of all other jobs are unchanged. Next swap the remaining jobs in B_2 (which are type-t) with an equal number of jobs in B_1 of type smaller than t. This move strictly decreases the cost of the schedule, because it decreases the processing time of batch B_2. The strict decrease in cost contradicts the optimality of the original schedule. Note that there are always enough jobs of type other than t to make this swap, because there are fewer than B leftover jobs of type t in total.

Case 2. B_1 is t-pure; B_2 contains jobs of smaller type(s). Moving all the type-t jobs from B_2 to B_1 strictly decreases the cost of the schedule, contradicting its optimality. Such a move is always possible because B_1 can not be full, since there are fewer than B leftover jobs of type t. □

An immediate consequence of this result is that an optimal schedule will have at most m leftover batches. The following lemma establishes the fact that when these batches are not full they will be sequenced in order of increasing processing times. Only full batches will not agree with this ordering.

Lemma 5. *In any optimal schedule, if there are two leftover jobs i and j with processing times $p_i < p_j$ such that the batch containing job j precedes the batch containing job i, then job j must be in a full batch.*

Proof. Suppose j precedes i but j's batch is not full. Then moving job i into the same batch with job j strictly decreases the cost of the schedule, since the completion time of job i is reduced while all other job completion times are unchanged. This strict decrease in cost contradicts the optimality of the original schedule. □

Let a *basic* schedule be one that satisfies the properties described in Lemmas 1–5. Since there is always an optimal schedule that is basic, an algorithm for the m-type burn-in problem may restrict its search to such schedules. Consider the following general strategy for finding an optimal basic schedule:

1. For each $t \in \{1 \ldots m\}$, create $\lfloor n_t/B \rfloor$ full homogeneous batches of type t.
2. List the leftover jobs in order of increasing processing time.
3. Divide the list into m or fewer batches (of size B or less) by indicating where each batch begins and ends. (By the consecutivity property of Lemma 3, this suffices to define the batches).
4. Arrange the batches (from Steps 1 and 3) in order of increasing $p(B_t)/|B_t|$ ratios, and evaluate the cost of the resulting schedule.
5. Repeat Steps 3 and 4 as necessary to create candidate schedules, and choose the one with lowest cost.

Except for number 3, all these steps are straightforward and can be done quickly. The key to developing an efficient algorithm thus depends upon the approach used in Step 3. In general, the large number of possible ways of batching the jobs in Step 3 means that a complete enumeration will not be efficient. In the next section we show that the optimal solution can be determined by enumerating only a small subset of all the possibilities. To this end we introduce the following terminology.

A *batching* is a list of leftover batches, where every leftover job appears in exactly one batch. A batching is called *basic* when it satisfies the properties described in Lemmas 3 and 4, namely:

- jobs are consecutive with respect to processing time in each batch, and
- each job type dominates at most one leftover batch.

Note that a basic schedule is composed of a basic batching together with some number of full homogeneous batches, where the batches (both homogeneous and leftover) are sequenced according to Lemma 1.

Every basic batching can be represented by listing all leftover jobs in order of increasing processing times from left to right and then inserting breakpoints into the list, indicating where each batch begins and ends. With this representation in mind, we may describe both batches and jobs as being to the left or right of other batches and jobs. Note that these references to left and right only apply to the batching (in which batches are ordered by dominant type), and not to the final schedule, in which batches are ordered according to Lemma 1. Consider, for instance, the optimal schedule in Example 1 above. The corresponding batching is: {3, 5}, {8}, {8, 9, 9}. Although batch {8} is to the left of {8, 9, 9} in the batching, it becomes the last batch in the final schedule.

1.2. Leftmost Batchings

1.2.1. Definitions. One way of describing a batching is to indicate which job type dominates each batch. More specifically, we might indicate which types dominate full batches and which dominate partial batches. Given two sets of job types, $F, P \subseteq \{1 \ldots m\}$, such that $F \cap P = \emptyset$, we say that a basic batching $\mathcal{A}$ *agrees with* the pair (F, P) if for each job type $t \in \{1 \ldots m\}$:

- $t \in F$ if and only if t dominates a full batch in $\mathcal{A}$,
- $t \in P$ if and only if t dominates a partial batch in $\mathcal{A}$,
- $t \in (F \cup P)^c$ otherwise.

In general there may be more than one basic batching that agrees with a given (F, P) pair, or there may be no such basic batching, in which case the pair (F, P) will be called *infeasible*. If there is at least one basic batching that agrees with (F, P) we may define a new batching, denoted by $L(F, P)$, which will be called the *leftmost batching* with respect to (F, P). $L(F, P)$ is the *unique* basic batching that agrees with (F, P), and in which all jobs are pushed to the "left" as much as possible. This means that if the batches are arranged from left to right in order of increasing processing time (i.e., dominating type), then it is impossible to move a job from its current batch into a batch to the left and still have a batching that agrees with (F, P).

1.2.2. Constructing the Leftmost Batching $L(F, P)$. Given the pair (F, P), constructing the corresponding leftmost batching is fairly straightforward. The basic idea is to proceed from left to right, adding as many jobs as possible to each batch while maintaining agreement with (F, P). At some point it may happen that a certain batch is required to be full, but adding the next job will change the dominant type of the batch, preventing agreement with (F, P). In this case it will be necessary to *pull* jobs from the batches to the left in order to fill the current batch. A detailed description of the procedure follows.

The dominating types in $F \cup P$ are sorted into a list in order of increasing processing times; the leftover jobs are sorted into another list, also in order of increasing processing times. At each stage of the procedure, a partial batching that agrees with (F, P) has already been created; it consists of a sequence of batches arranged from left to right in order of increasing dominant type. Let t be the next dominating type in the list which has not yet been assigned to a batch. t is assigned to the current batch, and jobs which are not yet in the batching are added to this batch in order of increasing processing time. The way that jobs are added depends upon whether the batch is to be partial or full (i.e., whether $t \in F$ or $t \in P$).

If $t \in P$, jobs are added to the batch dominated by t until either:

P1. The batch contains $B - 1$ jobs, or

P2. The last job added is of type t, and the next job to be added is of type $\hat{t} > t$.

If $t \in F$, jobs are added to the batch dominated by t until either:

F1. The batch contains B jobs, or

F2. The batch contains fewer than B jobs, and the next job to be added is of type $\hat{t} > t$.

If case **P1** or **F1** occurs, and the last job that was added to the batch is of type t, or if case **P2** occurs, then the batch agrees with (F, P). Furthermore, it is impossible to add another job to the batch and maintain agreement with (F, P) (since adding another job will either change the dominant type or make the batch full when it should be partial). At this point the batch is appended to the batching (at the right end), and the procedure continues by starting a new batch with the next dominant type in the list.

If case **P1** or **F1** occurs but the last job added to the batch was of type $\hat{t} < t$, then the procedure stops. It is impossible to add more jobs to the batch (while keeping it partial or full as required) without pushing some jobs into earlier batches. But since the earlier batches were packed as full as possible, there is no way to add jobs to them while maintaining agreement with (F, P). Hence there is no basic batching that agrees with (F, P).

Finally, suppose case **F2** occurs, and the batch has $B - k$ jobs, for some $k > 0$. The batch is not full, but adding the next job will cause the dominant type to be $\hat{t} > t$. In this case the only possibility for filling the batch while maintaining agreement with (F, P) is to *pull* k jobs from preceding batches (i.e. those to the left of the current batch).

A pull begins at the nearest (rightmost) preceding batch that is *not full*, because if jobs are removed from a full batch then it will become partial and the batching will no longer agree with (F, P). If there is no partial batch preceding the current batch, then there is no pull available and hence no basic batching that agrees with (F, P). Suppose there is a preceding partial batch, and let B_j be the nearest such batch. If $|B_j| \leq k$, then there are not enough jobs for the pull and there is no basic batching that agrees with (F, P). To see that this is true, note that if all jobs are moved out of B_j (to the right), then they must be replaced by jobs from batches to the left, which will be of different type than the dominating type of B_j. Thus the resulting batching will not agree with (F, P). If $|B_j| > k$, the pull is performed by moving the k largest jobs in B_j into the batch to the right. Then the k largest jobs in this batch are moved to the right, and so on until k jobs have been moved into the current batch, filling it. If this pull changes the dominant type of any of the batches between B_j and the current batch, then it was unsuccessful and there is no basic batching that agrees with (F, P). If, on the other hand, all batches maintain their original dominating types, then the resulting batching agrees with (F, P). (Note that after the pull, batch B_j is still a partial batch (but with k fewer jobs), the intervening batches have the same number of jobs as before the pull, and the current batch is now full with the addition of the k jobs).

The procedure continues until either all jobs have been added to the batching, or until the infeasibility of (F, P) has been established.

Lemma 6. *Given the pair (F, P), the procedure described above either constructs $L(F, P)$, or determines that (F, P) is infeasible, in $O(m^2)$ operations.*

Proof. The correctness of the procedure is essentially implicit in the description above, so a formal proof is omitted. It is easy to see that jobs are added to batches that are as far left as possible. This principle is only violated when it is impossible to maintain agreement with (F, P); in this case, jobs must be pulled from the left to the right. The pull is always executed so that the minimum number of jobs necessary to establish agreement with (F, P) is moved.

The bound on the procedure's running time is based upon the observation that jobs can be added to each batch in groups which share the same type, rather than one at a time. Since each type can appear in at most two different batches, there will be $O(m)$ additions. Pulling a group of k jobs from the left can be done with $O(m)$ operations, since there are at most m batches that jobs must move through. (Note also that the k jobs moved out of each batch must be of the same type, otherwise the dominant type of the batch will change and the batching will no longer agree with (F, P).) Since there is at most one pull per batch, the total time required for pulling is $O(m^2)$. Sorting the types in $F \cup P$ and sorting the leftover jobs can both be done in $O(m \log m)$ time. Thus the total procedure requires $O(m^2)$ operations. □

1.3. The m-type Burn-in Algorithm

Having defined leftmost batchings and described the procedure for constructing them, we may now state the result which is the basis for the algorithm that follows.

Theorem 7. *There exists an optimal schedule S for the m-type burn-in problem such that S is basic and the leftover batches of S form a leftmost batching.*

Proof. By Lemmas 1–5 there exists an optimal schedule S that is basic. Suppose that S has batching $\mathcal{A}$ with dominant type sets F^* and P^*. Since S is basic, it is clear that $\mathcal{A}$ is basic. We will show, by way of contradiction, that $\mathcal{A} = L(F^*, P^*)$ (i.e., $\mathcal{A}$ is leftmost with respect to (F^*, P^*)).

Suppose $\mathcal{A}$ is not leftmost with respect to (F^*, P^*). Then it is possible to push a leftover job to the left, yielding a new basic batching $\mathcal{A}'$ which agrees with (F^*, P^*). Such a push must be from a partial batch to a partial batch, because any other kind of move would yield a batching that does not agree with (F^*, P^*). Since S satisfies the property of Lemma 5, this push to the left in the batching corresponds to moving a job from a later batch to an earlier batch in the schedule, which means that the cost of schedule S is strictly reduced. (Since the resulting schedule has the same dominant types as S, the only change in cost is the decrease in the completion time of the moved job.) This decrease in cost contradicts the optimality of S; thus no such push can exist, so $\mathcal{A}$ must be leftmost with respect to (F^*, P^*). □

This result shows that in searching for an optimal basic schedule, we need only consider leftmost batchings of the leftover jobs. This fact is exploited in the following algorithm for the m-type burn-in problem:

Algorithm MTB

1. For each $t \in \{1 \ldots m\}$, create the $\lfloor n_t/B \rfloor$ full homogeneous batches of type t.
2. Arrange the leftover jobs in order of increasing processing time.
3. For each possible (F, P) pair, perform the following two steps:
   (a) Create the leftmost batching $L(F, P)$.
   (b) Arrange the batches (from Steps 1 and 3a) in order of increasing $p(B_t)/|B_t|$ ratios, and evaluate the cost of the resulting schedule.
4. From among the schedules created in Step 3, return the one with the lowest cost.

Theorem 8. *Algorithm MTB solves the m-type burn-in problem in $O(m^2 3^m)$ operations.*

Proof. The correctness of the algorithm follows immediately from Theorem 7. The running time is determined as follows. Creating the homogeneous full batches requires $O(m)$ operations. The leftover jobs are then batched by finding $L(F, P)$ for every possible (F, P) pair. Since every job type can either be assigned to F, to P, or to neither, the number of (F, P) pairs is 3^m. For each such pair, finding the leftmost batching requires $O(m^2)$ operations. Sorting all batches can be done in $O(m \log m)$ time. Thus the total running time of the algorithm is $O(m^2 3^m)$ operations. □

2. THE GENERAL BURN-IN PROBLEM ON A SINGLE MACHINE

We now turn to the general burn-in problem, in which the number of distinct job types is not fixed. Initially we will assume that an instance of this problem is described by n processing times, $p_1 \ldots p_n$ and a maximum batch size B. (Later we will consider the high-multiplicity version of the problem, i.e., the m-type problem when m is not fixed.) Chandru et al. (1993a) developed a branch-and-bound procedure for finding an optimal schedule, but it is effective only for small problem sizes. They also proposed two heuristics which find good approximate solutions in pseudopolynomial time.

In the sections that follow, we present a two-approximation algorithm for the general burn-in problem. In practice, this algorithm finds solutions which are very close to optimal, and we will prove that it guarantees a solution at least as good as those provided by the two heuristics described by Chandru et al. (1993a), for which no performance bound is given. Finally, we describe how the procedure can be modified so that its running time depends only on m, the number of distinct job types, making it a strongly polynomial procedure.

Before presenting our heuristic, we define a new restricted version of the burn-in problem. In the *fixed-sequence* burn-in problem, the goal is to find the lowest cost schedule in which jobs are scheduled in order of increasing processing time. In other words, a schedule for the fixed-sequence problem must satisfy the following property: if $p_i < p_j$, then job i cannot be in a batch later than the batch of job j. It is easy to see that in every such schedule, the jobs in each batch are consecutive with respect to processing times.

Our heuristic for the general burn-in problem is composed of two steps: first it finds an optimal schedule for the fixed-sequence version of the problem, and then it orders the batches by the BWSPT rule. In the next section we describe how the fixed-sequence burn-in problem can be solved using a straightforward dynamic programming procedure.

2.1. Dynamic Programming Solution for the Fixed-sequence Problem

Let the jobs be ordered so that $p_1 \leq p_2 \ldots, \leq p_n$. Define $f(i)$ as the minimum cost of a fixed-sequence schedule for jobs i through n. If the first batch in this schedule is known to contain jobs i through $k - 1$, then we can express $f(i)$ as $p_{k-1}(n - i + 1) + f(k)$. The first term is the contribution to the cost made by the batch containing jobs i through $k - 1$ (the processing time of the batch is multiplied by the number of jobs in the schedule following the batch, including those in the batch), and the second term is the cost of the remaining schedule for jobs k through n. Finding the minimum cost at each stage thus amounts to choosing the appropriate value of k, and f can be defined recursively, as follows:

Procedure SP

Base Case: $f(i) = 0$, *for all* $i > n$

Recurrence: $f(i) = \min_{\{k:k-i \leq B\}}\{f(k) + p_{k-1}(n - i + 1)\}$

Solution: $f(1)$

Each evaluation of f requires the comparison of at most B values, so the running time of the procedure is $O(nB)$. This procedure is the first step in our Fixed-Sequence (FS) heuristic for the burn-in problem:

Heuristic FS

1. Use procedure SP to find an optimal schedule for the fixed-sequence problem.
2. Arrange the batches from this solution in BWSPT order.

The following section establishes the fact that the optimal fixed-sequence schedule found by the dynamic programming procedure has a cost at most twice that of the optimal schedule for the burn-in problem. Since rearranging the batches can only improve the schedule, heuristic FS is a two-approximation for the burn-in problem.

2.2. A Two-approximation Algorithm

Theorem 9. *Heuristic FS provides a two-approximation for the burn-in problem.*

Proof. Let $S^*(K)$ be a basic optimal schedule for an instance of the burn-in problem in which K is the set of jobs, where $|K| = n$. Let $D^*(K)$ be the optimal fixed-sequence schedule for the same problem instance (before arranging the batches in BWSPT order). We will show by induction that for any set of jobs K, $C(D^*(K))/C(S^*(K)) \leq 2$.

Throughout the proof, we will make use of the following fact. If $S(K)$ is a schedule for a set of jobs K, with $|K| = n$, in which the first batch contains jobs $1 \ldots j$, then the cost of $S(K)$ can be expressed as: $C(S(K)) = np + C(S(J))$, where $p = p_j$ is the processing time of the first batch and J is the set of jobs $K - \{1, \ldots j\}$. Also note that if $S(K)$ is the optimal schedule for the jobs in K, then $S(J)$ is optimal for the jobs in J. (If this were not the case, rebatching the jobs in J would improve the total schedule, contradicting the optimality of $S(K)$).

Base Case. $|K| = 1$. In this case both schedules $S^*(K)$ and $D^*(K)$ consist of a single batch containing a single job. Thus $C(D^*(K))/C(S^*(K)) = 1 \leq 2$.

Induction Step. Suppose that $C(D^*(J))/C(S^*(J)) \leq 2$ for all $J \subset K$. We consider three possible cases. Case 1 occurs when the first batch in the schedule contains jobs 1 through j for some $j \leq B$. Cases 2 and 3 occur when the first batch is a full "out-of-order" batch. That these are the only three possibilities follows from Lemma 5.

Case 1. The first batch of $S^*(K)$ contains jobs $1 \ldots j$, for some $j \leq B$.

Let $p = p_j$, the processing time of the first batch. Then $C(S^*(K)) = np + C(S^*(J))$, where $J = K - \{1 \ldots j\}$ and the second term on the right is the optimal value of a schedule for the $n - j$ largest jobs. The dynamic programming procedure will find a schedule at least as good as the one in which jobs $1 \ldots j$ are in the first batch, followed by the optimal fixed-sequence schedule for the remaining jobs. Thus, $C(D^*(K)) \leq np + C(D^*(J))$, and since $C(D^*(J))/C(S^*(J)) \leq 2$ for all $J \subset K$, we have

$$\frac{C(D^*(K))}{C(S^*(K))} \leq \frac{C(D^*(J)) + np}{C(S^*(J)) + np} \leq 2.$$

Case 2. The first batch of $S^*(K)$ contains jobs $j + 1 \ldots j + B$, where $j < B$.

Letting $p = p_{j+B}$, the processing time of the first batch, we have $C(S^*(K)) = np + C(S^*(J))$, where $J = K - \{j + 1, \ldots j + B\}$, and the second term on the right is the optimal value of a schedule for the $n - B$ jobs after the first batch. The dynamic programming procedure will always give a solution at least as good as the one with jobs $1 \ldots j$ in the first batch and jobs $j + 1 \ldots j + B$ in the second batch, followed by the optimal fixed-sequence schedule for the remaining jobs. Thus we have

$$\begin{aligned} C(D^*(K)) &\leq np_j + (n - j)p + C(D^*(J - \{1 \ldots j\})) \\ &\leq (2n - j)p + C(D^*(J)), \end{aligned}$$

since $p_j \leq p = p_{j+B}$ and $D^*(J - \{1 \ldots j\}) \leq D^*(J)$.

Combining this with the induction hypothesis gives:

$$\frac{C(D^*(K))}{C(S^*(K))} \leq \frac{C(D^*(J)) + (2n - j)p}{C(S^*(J)) + np} \leq 2.$$

Case 3. The first batch of $S^*(K)$ contains jobs $j + 1 \ldots j + B$, where $j \geq B$.

Somewhere in schedule $S^*(K)$ are the jobs $1 \ldots B$, in some number of (not necessarily consecutive) batches. Note that the batch with job B may also contain bigger jobs. We consider two subcases:

Case 3a. The batch containing job 1 is preceded by fewer than $n/2$ jobs in $S^*(K)$.

Suppose the batch containing job 1 has a total of j jobs, so it has processing time $p = p_j$. Moving this batch to the front of the schedule increases the cost of the schedule by an amount less than $n/2p$, since fewer than $n/2$ jobs are delayed by an additional p units of processing time. Denote the resulting schedule by $S'(K)$. Since $C(S'(K)) < C(S^*(K)) + n/2p$, we have:

$$\begin{aligned} C(S^*(K)) &> C(S'(K)) - \frac{n}{2}p \\ &> C(S^*(J)) + np - \frac{n}{2}p = C(S^*(J)) + \frac{n}{2}p, \end{aligned}$$

where the second inequality follows from the fact that $C(S'(K)) = np + C(S''(J))$ for some schedule $S''(J)$, and $C(S''(J)) \geq C(S^*(J))$ by the optimality of $S^*(J)$.

We also know that $C(D^*(K)) \leqslant np + C(D^*(J))$, and thus

$$\frac{C(D^*(K))}{C(S^*(K))} < \frac{C(D^*(J)) + np}{C(S^*(J)) + \frac{n}{2}p} \leqslant 2.$$

Case 3b. The batch containing job 1 is preceded by more than $n/2$ jobs in $S^*(K)$.

Consider the batch that contains job B, and suppose this batch also contains k jobs with smaller processing times (and possibly some jobs with larger processing times). Let $p = p_B$. Moving job B and the preceding jobs in the batch to the front of the schedule increases the cost by at most np, since at most n jobs are delayed by an additional p units of processing time. Now move all the remaining jobs with index less than B into this first batch, yielding schedule $S'(K)$, in which the first batch contains jobs $1 \dots B$. Each job moved in this way has its completion time reduced by at least $n/(2B)p$, since there are at least $n/2$ jobs preceding job 1, and hence at least $n/(2B)$ batches, each with a processing time greater than $p = p_B$. Since there are B jobs moved in this way, there is a total decrease in cost of at least $n/2p$. Thus the total increase in the cost of the schedule is less than $np - n/2p = n/2p$, and we have:

$$C(S^*(K)) > C(S'(K)) - \frac{n}{2}p$$

$$> C(S^*(J)) + np - \frac{n}{2}p = C(S^*(J)) + \frac{n}{2}p,$$

where the second inequality follows from the fact that $C(S'(K)) = np + C(S''(J))$ for some schedule $S''(J)$ and $C(S''(J)) \geqslant C(S^*(J))$ by the optimality of $S^*(J)$.

We also know that $C(D^*(K)) \leqslant np + C(D^*(J))$, and thus

$$\frac{C(D^*(K))}{C(S^*(K))} < \frac{C(D^*(J)) + np}{C(S^*(J)) + \frac{n}{2}p} \leqslant 2. \quad \square$$

Thus heuristic FS is a two-approximation algorithm for the general burn-in problem. In the next section we compare it with two previously proposed heuristics.

2.3. Comparison with Other Heuristics

In this section we compare heuristic FS with the two heuristics for the burn-in problem described by Chandru et al. (1993a).

Heuristic FBSPT (Chandru et al. 1993a)

1. Form $k = \lceil n/B \rceil$ batches where batch i contains the jobs indexed $(i - 1)B + 1$ to $\min\{n, iB\}$.
2. Schedule the batches in BWSPT order.

Heuristic GR (Chandru et al. 1993a)

1. Set $i = 1$.
2. Find job k such that

$$\frac{p_k}{(k - i + 1)} = \min\left\{p_i, \frac{p_{i+1}}{2}, \frac{p_{i+2}}{3}, \dots, \frac{p_{i+B-1}}{B}\right\}.$$

Place jobs i through k in a batch together and schedule this batch on the machine. If all jobs have been scheduled, stop. Otherwise, set $i = k + 1$ and repeat Step 2.

In order to compare FS with these two heuristics, we first observe that both heuristics generate schedules which are feasible for the fixed-sequence problem—that is, jobs are scheduled in order of increasing processing time. (Although Step 2 of heuristic FBSPT involves arranging the batches in BWSPT order, it is easy to verify that the batches created in Step 1 are already in this order, so the job sequence does not change.) Since heuristic FS finds the *optimal* schedule for the fixed-sequence problem and then improves it by rearranging the batches, it is guaranteed to find a solution with the same or lower cost than those produced by the other heuristics. In Section 4 below, we present computational results which show that heuristic FS does indeed find better schedules.

Note that all three heuristics have running times that depend on n, the number of jobs, and B, the maximum batch size. Any procedure which depends on B is only pseudopolynomial. Furthermore, if the problem instance can be described by $n_1, n_2, \dots, n_m$, where each n_i is the number of jobs of type i, then a procedure is only polynomial if its running time is a polynomial function of log $n_1, \dots,$ log n_t. Thus all three heuristics discussed above run in pseudopolynomial time. The dynamic programming procedure of heuristic FS can be modified so that its running time depends only on m, the number of distinct job types, thereby making it a truly polynomial algorithm. (Due to space considerations, we have omitted the description of this modification.)

3. THE PARALLEL MACHINES BURN-IN PROBLEM

The parallel machine burn-in (PMB) problem is defined as follows: given m job types, where for each type $t \in \{1, \dots, m\}$ there are n_t jobs with processing time p_t, and given a maximum batch size B, and a number of machines M, find a schedule (an assignment of jobs to batches and batches to machines) to minimize the sum of job completion times. This problem was addressed by Chandru et al. (1993a), where the authors proved some structural properties of optimal schedules and showed how to extend their heuristics for the single-machine burn-in problem to the parallel machines version.

In the sections that follow we address the m-type PMB problem, in which the number of distinct job types is fixed. We show that the structural properties established in Section 1.1 for the single-machine problem are easily extended to the parallel machines problem, and hence the search for an optimal schedule can be restricted to leftmost batchings. Once a particular leftmost batching has been determined, the remaining problem is to assign batches to machines. This batch assignment problem is equivalent to the classic scheduling problem of minimizing the weighted flowtime of a set of jobs on parallel machines, which is NP-hard. Nonetheless, pseudo-polynomial procedures can be used to solve the problem exactly, and there are polynomial approximation algorithms that guarantee solutions that are close to optimal.

3.1. Structural Properties

Here we extend the results of Section 1.1 for the single-machine problem to the parallel-machine problem. The proofs of these results are omitted, since they are virtually identical to those of Section 1.1. The five structural properties of optimal schedules are extended to the PMB problem as follows:

1. There exists an optimal schedule for the PMB problem in which the batches on each machine are in BWSPT order.
2. There exists an optimal solution for the PMB problem containing $\lfloor n_t/B \rfloor$ homogeneous and full batches of type t.
3. There exists an optimal solution in which the jobs in each leftover batch are consecutive with respect to processing times, that is, for any three leftover jobs with processing times $p_i \leqslant p_j \leqslant p_k$, if a batch contains the jobs with processing times p_i and p_k, it must also contain the job with processing time p_j.
4. In every optimal schedule that satisfies the three preceding properties, each job type $t \in \{1 \dots m\}$ dominates at most one leftover batch.
5. In any optimal schedule, if there are two leftover jobs i and j with processing times $p_i < p_j$ such that the batch containing job j has an earlier completion time than the batch containing job i, then job j must be in a full batch.

The proofs of properties 1–3 and 5 are straightforward. In the proof of property 4, it is necessary to specify the meaning of "batch B_1 precedes batch B_2." In the case of parallel machines, where two batches may be on different machines, we say batch B_1 *precedes* batch B_2 if the completion time of batch B_1 is less than or equal to the completion time of batch B_2.

Let a *basic* schedule be one that satisfies five properties given above. Since there is always an optimal schedule that is basic, an algorithm for the m-type PMB problem may restrict its search to such schedules. In Section 1.2 we defined leftmost batchings. Using the same definition, we may now prove that the PMB problem, like the single machine problem, always has an optimal schedule which is a leftmost batching.

Theorem 10. *There exists an optimal schedule S for the m-type PMB problem such that S is basic, and the leftover batches of S form a leftmost batching.*

Proof. There is always an optimal schedule S that is basic. Suppose that S has batching $\mathcal{A}$ with dominant type sets F^* and P^*. Since S is basic, it is clear that $\mathcal{A}$ is basic. We will show, by way of contradiction, that $\mathcal{A} = L(F^*, P^*)$ (i.e., $\mathcal{A}$ is leftmost with respect to (F^*, P^*).

Suppose $\mathcal{A}$ is not leftmost with respect to (F^*, P^*). Then it is possible to push a leftover job to the left, yielding a new basic batching $\mathcal{A}'$ which agrees with (F^*, P^*). Such a push must be from a partial batch to a partial batch, because any other kind of move would yield a batching that does not agree with (F^*, P^*). Since S satisfies property 5, this push to the left in the batching corresponds to moving a job from a later batch to an earlier batch in the schedule, which means that the cost of schedule S is strictly reduced. (Since the resulting schedule has the same dominant types as S, the only change in cost is the decrease in the completion time of the moved job.) This decrease in cost contradicts the optimality of S; thus no such push can exist, so $\mathcal{A}$ must be leftmost with respect to (F^*, P^*).

This theorem suggests the following solution procedure for the m-type parallel machine burn-in problem:

Procedure G-PMB: A General Approach to the PMB Problem

1. For each $t \in \{1 \dots m\}$, create the $\lfloor n_t/B \rfloor$ full homogeneous batches of type t.
2. Arrange the leftover jobs in order of increasing processing time.
3. For each possible (F, P) pair, perform the following two steps:
   (a) Create the leftmost batching $L(F, P)$.
   (b) Assign the batches from Steps 1 and 3a to the M machines, and evaluate the cost of the resulting schedule.
4. From among the schedules created in Step 3, return the one with the lowest cost.

In the next section we address the question of how to perform Step 3b in the above algorithm.

3.2. Assigning Batches to Machines

Once jobs have been assigned to batches, the problem of scheduling the batches on the M machines may be viewed as an instance of the weighted flowtime problem, which may be described as follows: given n jobs, each with processing time p_i and weight w_i for $i \in \{1, \dots, n\}$, and M machines, find a schedule to minimize the weighted sum of job completion times. It is clear that the problem of assigning batches to machines is identical to this scheduling problem—each batch B_i may be viewed as a job with processing time $P(B_i)$ and weight $|B_i|$.

Although the weighted flowtime problem on parallel machines is NP-hard (see Bruno et al. 1974), pseudopolynomial algorithms have been described by Lawler and Moore (1969), Rothkopf (1966), and Lee and Uzsoy (1992). The latter algorithm has a running time of $O(nMW^{M-1})$, where $W = \sum_{i=1}^{n} w_i$. In the PMB problem, the "weight" of a batch is the number of jobs it contains, and thus the sum of job weights W is simply n, the total number of jobs. Thus batches may be scheduled on M machines in $O(Mn^M)$ operations. Using this algorithm as the subroutine for Step 3b in the above procedure yields a total running time of $O(Mn^M m^2 3^m)$. Note that even when the number of types and the number of machines is fixed, this running time is still only pseudopolynomial, since a problem instance requires the specification of m rather than n processing times.

A much faster procedure may be used to find very good (though not necessarily optimal) solutions to the weighted flowtime problem. First, jobs are arranged in order of increasing p_t/w_t ratio. Each job is removed from this list and assigned to the first machine that becomes available for processing. A result due to Kawaguchi and Kyan (1986) is that this heuristic guarantees a solution no worse than $(\sqrt{2} + 1)/2 \approx 1.2$ times the optimal solution. Since the running time of this heuristic is dominated by sorting the jobs, it requires $O(n \log n)$ operations.

This heuristic may be used as the subroutine of Step 3b in the above procedure to find a very good solution for the PMB problem. At first glance it appears that the running time of the subroutine depends on the total number of batches which must be assigned to the machines, which may be large. The following observation suggests a method for streamlining the assignment.

If batches are arranged in BWSPT order and sequentially assigned to the next available machine, and if there are k_t full homogeneous batches of type t for each $t \in \{1, \ldots, m\}$, then at least $\lfloor k_t/M \rfloor$ of these batches will be assigned to each machine. To see why this is true, it is sufficient to note that at the time of scheduling the full batches of type t, every batch assigned to a machine so far has a processing time that is less than p_t (because if a batch with larger processing time had been assigned, that batch would have to be of a size greater than B, which is a contradiction). Thus the difference between the makespans of the partial schedules on any two machines is at most p_t, and it immediately follows that the full homogeneous batches of type t will be scheduled so that each machine gets a batch before any machine gets two batches. In other words, full homogeneous batches may be assigned to machines in groups, so that at most $M - 1$ batches of each type must be assigned one at a time.

Since there are at most m leftover batches (by property 4), and since it takes at most M operations to assign all the full homogeneous batches of each type to the M machines, the total number of operations required to sort and then assign batches is $O(m \log m + m(M + 1))$. Recall that finding a leftmost batching can be done in $O(m^2)$ operations, and that there are at most 3^m such batchings which must be checked. Thus the total running time of procedure G-PMB when the BWSPT rule is used for step 3b is $O(3^m \min \{m^2, m(M + 1)\})$. The resulting solution will be no more than $\sqrt{2} + 1/2 \approx 1.2$ times the optimal solution.

4. EMPIRICAL RESULTS

In this section we report the results of empirical comparisons of different solution procedures for the single machine burn-in problem. First we compare two procedures which find optimal solutions to the m-type burn-in problem: the branch-and-bound procedure of Chandru et al. (1993a), and algorithm MTB (described in Section 1 of this paper). We report on the CPU time required by the two procedures for finding solutions to problems with five distinct job types. Next we compare three heuristics for solving the general burn-in problem: two which were presented by Chandru et al. (1993a), and heuristic FS (described in Section 2.1 of this paper). In this case we compare the quality of the solutions obtained (rather than the CPU time required). Algorithm MTB and heuristic FS were implemented using the C programming language on a Sun SPARC II workstation.

4.1. Comparison of Exact Algorithms

We used the MTB algorithm to solve problems that were randomly generated using the following distribution of job types:

Type	Processing Time	Probability
1	15	0.25
2	96	0.15
3	120	0.25
4	150	0.25
5	240	0.10

This distribution was chosen by Chandru et al. (1993a) because it represents a realistic mix of products that might be found at a semiconductor fabrication plant.

We solved 10 problems for each value of B, the maximum batch size, and n, the total number of jobs. Table I presents the CPU time required to solve each set of 10 problems, and compares it with the CPU time required by the branch-and-bound algorithm of Chandru et al. (1993a). The results indicate that the running time of the MTB algorithm is essentially independent of the batch size and the total number of jobs, while the running time of the branch-and-bound procedure is dependent on both parameters and therefore cannot effectively solve problems with more than 30 jobs.

Note that these problems all had five distinct job types. If the number of distinct types were greater, the branch-and-bound algorithm would probably be impractical for even fewer than 30 jobs. On the other hand, in computational experiments the MTB algorithm was able to solve problems with up to 12 distinct jobs types in a few minutes of CPU time. This result held true regardless of the batch size or the total number of jobs. We conclude that the MTB algorithm can efficiently solve any problem that is likely to arise in practice.

4.2. Comparisons of Heuristics

Heuristic FS was used to solve the same randomly generated problem instances used by Chandru et al. (1993a) to

Table I
Comparison of Exact Algorithms for Problems with Five Job Types

Number of jobs	Branch and bound CPU time[a]	MTB Algorithm CPU time[a]
$B = 3$		
10	45	0.2
15	52	0.2
20	93	0.2
25	590	0.2
30	1255	0.2
$B = 5$		
10	47	0.2
15	53	0.3
20	136	0.2
25	547	0.3
30	1419	0.2
$B = 7$		
10	47	0.3
15	55	0.4
20	119	0.4
25	419	0.4
30	2325	0.4
$B = 100$		
100	—	0.2
1000	—	0.4
10000	—	1.4
$B = 200$		
100	—	0.2
1000	—	0.4
10000	—	1.4

[a]CPU time measured in seconds per 10 problems

Table II
Comparison of Heuristics for Problems with Uniform (1,100) Job Processing Times

Number of Jobs	Avg. FBSPT Solution	Avg. GR Solution	Avg. FS Solution
$B = 3$			
10	1131.6	1057.2	1047.3
15	1958.1	1931.9	1915.5
20	3237.3	3164.2	3137.9
25	4892.0	4900.6	4849.4
30	6573.0	6555.2	6490.8
35	8705.2	8637.1	8591.4
$B = 5$			
10	983.0	893.5	878.7
15	1585.0	1519.7	1484.2
20	2393.0	2362.3	2332.7
25	3501.5	3465.3	3430.2
30	4606.0	4547.9	4512.8
35	5950.0	5933.6	5905.5
$B = 7$			
10	924.6	837.2	825.1
15	1532.7	1379.1	1360.6
20	2053.4	2069.6	2001.4
25	3049.6	2937.1	2886.5
30	3855.1	3850.3	3771.5
35	4867.1	4837.9	4802.0

test heuristics FBSPT and GR. Table II shows the average solution value when each heuristic was applied to 10 problems with job processing times randomly generated from a uniform distribution on the interval [1, 100]. It is clear that FS finds the best solutions among the three heuristics. Although the improvement over the GR heuristic may appear small, it is important to note that the average deviation of the GR solutions from optimal is never more than 3.4 percent (Chandru et al. 1993a). Thus even an improvement of 1 percent over the GR solution is significant.

5. CONCLUSION

The $O(m^2 3^m)$ MTB algorithm presented above is a significant improvement over the $O(m^3 B^{m+1})$ dynamic programming procedure of Chandru et al. (1993b), and the branch-and-bound procedure of Chandru et al. (1993a) for two reasons: first, because it is independent of both the maximum batch size and the number of jobs; and second, because it can be used to solve practical problems which have a greater number (m) of distinct product types. The empirical results reported above verify that the algorithm is quite fast in practice.

When the number of distinct job types is large, heuristic methods can be used to find solutions that are very close to optimal. Of the heuristics that have been proposed for the general burn-in problem, heuristic FS (described in Section 2.1) is the only one with a proven worst-case performance bound. Furthermore, it is guaranteed to find a solution at least as good as those found by any heuristic that maintains jobs in order of increasing processing time, such as heuristics GR and FBSPT. Indeed, the empirical results reported above show that FS does provide better solutions in practice. These results also suggest that heuristic FS does significantly better than the proven worst-case performance ratio of 2. We have been unable to construct a problem instance in which this worst-case bound is achieved, and we conjecture that a tighter bound can be established with a more sophisticated proof.

Relatively little research has been done on the parallel machine burn-in problem. We have shown that when the number of distinct job types is fixed, it is possible to find the optimal solution in pseudopolynomial time, or to find a solution less than $(1 + \sqrt{2}/2)$ times the optimal value in polynomial time.

Resolving the complexity of the general single machine burn-in problem is perhaps the most interesting subject for future research. Hochbaum and Landy (1995) have shown that there is a polynomial algorithm for the special case $B = 2$. It can also be shown that there is a pseudopolynomial algorithm for the case when the total number of jobs is small relative to B (bounded above by cB for a constant c).

The complexity of the parallel machines problem is also unresolved. It seems likely that this problem is NP-complete, since the problem of sequencing jobs on parallel machines to minimize weighted flowtime is NP-complete, and the PMB problem appears at least as hard.

ACKNOWLEDGMENT

This research was supported in part by the Competitive Semiconductor Manufacturing project of the Sloan Foundation. The research of the first author was also supported by the Office of Naval Research under grant N00014-88-K-0377.

REFERENCES

AHMADI, J. H., R. R. AHMADI, S. DASU, AND C. S. TANG. 1992. Batching and Scheduling Jobs on Batch and Discrete Processors. *Opns. Res.* **40,** 750–763.

ALBERS, S., AND P. BRUCKER. 1993. The Complexity of One-machine Batching Problems. *Discrete Applied Mathematics,* **47,** 87–107.

BRUCKER, P. 1991. Scheduling Problems in Connection with Flexible Production Systems. *Proceedings, 1991 IEEE Int. Conf. on Robotics and Automation*, 1778–1783.

BRUNO, J., E. G. COFFMAN, AND R. SETHI. 1974. Scheduling Independent Tasks to Reduce Mean Finishing Time. *Comm. ACM,* **17,** 382–387.

CHANDRU, V., C. Y. LEE, AND R. UZSOY. 1993a. Minimizing Total Completion Time on Batch Processing Machines. *Int. J. Prod. Res.* **31,** 2097–2121.

CHANDRU, V., C. Y. LEE, AND R. UZSOY. 1993b. Minimizing Total Completion Time on a Batch Processing Machine with Job Families. *O. R. Lett.* **13,** 61–65.

COFFMAN JR., E., A. NOZARI, AND M. YANNAKAKIS. 1989. Optimal Scheduling of Products with Two Subassemblies on a Single Machine. *Opns. Res.* **37,** 426–436.

COFFMAN JR., E., M. YANNAKAKIS, M. MAGAZINE, AND C. SANTOS. 1990. Batch Sizing and Sequencing on a Single Machine. *Ann. Opns. Res.* **26,** 135–147.

DOBSON, G., U. S. KARMARKAR, AND J. L. RUMMEL. 1987. Batching to Minimize Flow Times on One Machine. *Mgmt. Sci.* **33,** 784–799.

GLASSEY, C. R. AND W. W. WENG. 1991. Dynamic Batching Heuristics for Simultaneous Processing. *IEEE Trans. Semiconductor Manufacturing* **4,** 77–82.

HOCHBAUM, D. AND D. LANDY. 1994. Scheduling with Batching: Minimizing the Weighted Number of Tardy Jobs. *O. R. Lett.* **16,** 79–86.

HOCHBAUM, D. AND D. LANDY. 1995. The Double Batching Problem. Working Paper. Engineering Systems Research Center, University of California, Berkeley.

IKURA, Y. AND M. GIMPLE. 1986. Scheduling Algorithms for a Single Batch Processing Machine. *O. R. Lett.* **5,** 61–65.

KAWAGUCHI, T. AND S. KYAN. 1986. Worst Case Bound of an lrf Schedule for the Mean Weighted Flow-time Problem. *SIAM J. Computing* **15,** 1119–1129.

LAWLER, E. AND J. MOORE. 1969. A Functional Equation and Its Application to Resource Allocation and Sequencing Problems. *Mgmt. Sci.* **16,** 77–84.

LEE, C. Y. AND R. UZSOY. 1992. A New Dynamic Programming Algorithm for the Parallel Machines Total Weighted Completion Time Problem. *O. R. Lett.* **11,** 73–75.

LEE, C. Y., R. UZSOY, AND L. A. MARTIN-VEIGA. 1992. Efficient Algorithms for Scheduling Semiconductor Burn-in Operations. *Opns. Res.* **40,** 764–775.

NADDEF, D. AND C. SANTOS. 1988. One-pass Batching Algorithms for the One Machine Problem. *Discrete Appl. Math.* **21,** 133–145.

ROTHKOPF, M. H. 1966. Scheduling Independent Tasks on Parallel Processors. *Mgmt. Sci.* **12,** 437–447.

SHALLCROSS, D. 1992. A Polynomial Algorithm for a One Machine Batching Problem. *O. R. Lett.* **11,** 213–218.

UZSOY, R., C. Y. LEE, AND L. A. MARTIN-VEIGA. 1990. A Review of Production Planning and Scheduling Models in the Semiconductor Industry. *IIE Trans.* **24,** 47– 60.