Available online at www.sciencedirect.com

SCIENCE DIRECT®

European Journal of Operational Research 168 (2006) 870–879

EUROPEAN JOURNAL OF OPERATIONAL RESEARCH

www.elsevier.com/locate/ejor

Bucket brigades on in-tree assembly networks

John J. Bartholdi III [a,*], Donald D. Eisenstein [b], Yun Fong Lim [a]

[a] *School of Industrial and Systems Engineering, Georgia Institute of Technology, Atlanta, GA 30332-0205, USA*
[b] *Graduate School of Business, The University of Chicago, Chicago, IL 60637, USA*

Available online 13 September 2004

Abstract

In a *network* of subassembly lines, balance becomes more difficult to achieve as it requires that all subassembly lines be synchronized to produce at the same rate. We show how to adapt the "bucket brigade" protocol of work-sharing so that balance emerges spontaneously.
© 2004 Elsevier B.V. All rights reserved.

Keywords: Bucket brigade; Assembly line; Assembly network; Work-sharing; Dynamical systems; Self-organizing systems

1. In-tree assembly networks

Consider a network of subassembly lines such as shown in Fig. 1, where each arc represents a subassembly line that produces a subcomponent (or, equivalently, a part). The subassembly lines merge until a last line produces the final product, each instance of which we call an *item*. We restrict consideration to assembly networks that are *in-trees*: each node has exactly one leaving arc, except for the root node (node K in Fig. 1), which has none. The flow of assembly is *in* toward a final assembly line from which finished product emerges. See for example [1] for a discussion of directed in-trees.

Arc j is that one emanating from node j and it represents the work-content on subassembly line j, on which subcomponent j is assembled. Each subassembly line j has a buffer j located at the end of the line to store completed instances of subcomponent j. Fig. 2 shows the relationship between node j, line j, and buffer j.

A *leaf node* is a node with no entering arcs on the in-tree, such as A, B or C. Assembly of different subcomponents is initiated at leaf nodes, and these subcomponents are assembled in moving along the subassembly lines devoted to them. At an interior node such as F, subcomponents are joined to the assembly of a larger subcomponent F.

[*] Corresponding author.
E-mail address: john.bartholdi@isye.gatech.edu (J.J. Bartholdi III).

doi:10.1016/j.ejor.2004.07.034

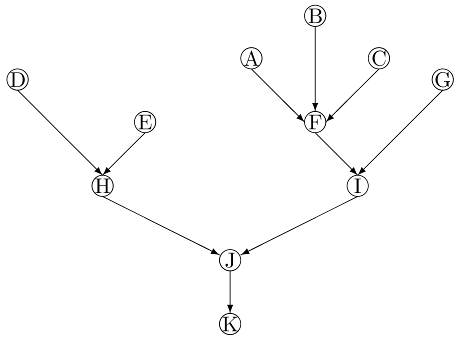

Fig. 1. An assembly network is represented by a connected directed graph, which represents the constraints on the sequence in which subcomponents are assembled.

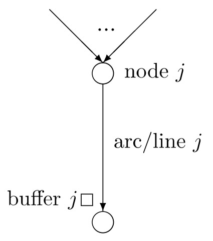

Fig. 2. Subcomponent j is assembled on line j and deposited in buffer j.

Each instance of the finished product is completed and leaves the network at the root node, in this example K.

We assume that the work-content on each subassembly line has already been determined and is described by the following.

Assumption 1. The work to process each subcomponent is deterministic and is spread continuously and uniformly over the corresponding subassembly line. Furthermore the work-content at any station is preëmptible without significant loss of work.

An example of such a line is one with relatively many work stations, each of which has a small piece of the total work-content. As discussed in previous work, these assumptions are not strictly necessary but they greatly simplify analysis [4,6,7].

Let there be n workers on the assembly in-tree. Our model of them is intended to capture simple, unskilled assembly in which workers are fully cross-trained:

Assumption 2. Each worker $i = 1,\ldots,n$ is characterized by a work velocity v_i that is fixed and constant over all subassembly lines.

Assume distinct workers labeled so that $v_1 < v_2 < \cdots < v_n$.

2. Balance

An in-tree assembly network is *balanced* if the following three conditions are satisfied:

Repetition: Each worker repeats the same portion of work-content on each successive instance of the product.

Efficiency: Workers are utilized to their fullest, without creating undue work-in-process.

Synchronization: All subassembly lines produce at a common rate.

Balance is desirable because then the skills of each worker are reinforced by repetition, the effort of each worker is directly realized as output of finished product, and production is regular, which simplifies downstream processes.

Traditional assembly line balancing forces repetition by assigning workers to zones. It tries to achieve efficiency by computing perfect shares of work-content, accounting for differences in the velocities of the workers. (For example, see [8,13], though these ignore, as does almost the entire modern literature, the significant differences among worker velocities.) Synchronization may be sought by trying to make work-shares similar across all subassembly lines.

Unfortunately, this approach requires the construction of a detailed model of work-content and knowledge of the exact velocities of the workers. But, even if balance is achieved on average—and even this is hard, as witness this journal—it would be difficult to maintain balance under the vicissitudes of the factory floor. Consequently, as a practical matter, synchronization is achieved, if at all, by sacrificing efficiency and accepting waste in the form of either some idleness among workers or else in growing work-in-process, especially between subassembly lines.

Bartholdi and Eisenstein [4] proposed the "bucket brigade" protocol to balance linear assembly lines (that is, assembly lines that may be represented as single arcs). A linear assembly line running the bucket brigade protocol seeks balance by allowing workers to move freely on the line. This allows the system to *dynamically* share work among workers by moving them to where the work is.

The goal of this paper is to explore how the bucket brigade protocol can be extended to make an in-tree of assembly lines self-balancing.

In traditional linear assembly under a bucket brigade protocol, there are more stations than workers and workers carry their work from station to station. Each worker carries a single item and each item is progressively assembled as it passes from worker to worker. Workers are not allowed to pass one another and so their sequence along the assembly line remains unchanged. When the last worker finishes his item at the end of the line, he walks back upstream and takes over the work of his predecessor, who in turn walks back and takes over the work of *his* predecessor, and so on, until the first worker begins a new item at the start of the line.

In contrast to traditional assembly line balancing, workers in bucket brigade assembly are not restricted to any particular zone of the assembly line or assignment of work-content. This allows the system to dynamically reallocate work by moving workers. If workers are sequenced from slowest to fastest (according to their work velocities) along the direction of material flow, the system spontaneously converges to a stationary state in which every worker repeatedly executes the same portion of work on every item produced [4]. We say the line balances itself. The system is also efficient—the number of units of work-in-process never exceeds the number of workers and the throughput (number of products finished per unit time) is the maximum possible [4].

The idea of bucket brigades seems to be robustly applicable. Even though we have made some simplifying assumptions about the nature of work and the workers, these need not strictly hold for bucket brigades to be effective. For example, Bartholdi et al. [7] showed that bucket brigades remain effective in the presence of variability of processing times. More generally, a selection of case studies on the use of bucket brigades in industry can be found in [5]. See [2] and [6] for more theoretical treatments of bucket brigades.

Up to now, bucket brigades have been defined and studied only on linear assembly lines. To use the same idea on an in-tree we must specify exactly how the workers should move among the subassembly lines.

3. What should a worker do next?

We implicitly determine what workers do next (that is, where to move) by conceptually transforming the in-tree of subassembly lines into a single linear assembly line that represents a common sequence of work-content to be completed by the workers in the course of production. Later we will identify some new complications this introduces into the familiar form of bucket brigades and show how to adapt bucket brigades to handle these complications.

Consider an in-tree of subassembly lines, such as the one illustrated in Fig. 1. We assume that the physical layout of the assembly network is fixed (we are not laying out the assembly line). To adapt the bucket brigade protocol to an in-tree structure, we impose a linear sequence on the work-content of the in-tree. This linear sequence is chosen to be consistent with the partial ordering imposed by the in-tree itself; but because the sequence is a total ordering, we can treat it as a linear assembly line to which (some form of) bucket brigades might be applied.

Fig. 3 shows one total order of the subassembly lines from among the many that are consistent with the partial order of lines of the in-tree. For example, line D precedes line H in any total order because processing a subcomponent H requires a subcomponent D. The total order of work is illustrated in Fig. 4, which is a conceptual representation of the sequence in which the work will be completed. We refer to the total ordering shown in Fig. 4 as the *serialized assembly line*.

Having totally-ordered all the work-content of the in-tree, the main points of the bucket brigade protocol are now well-defined. In particular, both

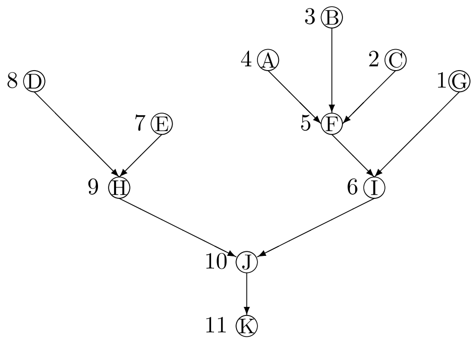

Fig. 3. A total ordering of the subassembly lines: each node has been assigned an ordinal number representing the order of the subassembly line emanating from that node. Each worker follows this sequence of work when he proceeds forward in the assembly process. Note that this total order of nodes is consistent with the partial order of subassembly lines defined by the in-tree.

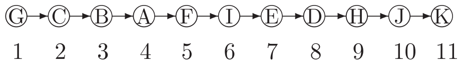

Fig. 4. The serialized assembly line is a conceptual representation of a total order of work on the in-tree of Fig. 3.

the initialization (sequence workers from slowest to fastest) and the basic rule of movement (work "forward" and walk "back" to get more work) are understood in relation to this total order.

Thus under bucket brigades for in-trees, the partial order conveyed by the in-tree is serialized, and each worker follows the rules given in Table 1.

Thus, in Fig. 4 the slowest worker in a team is always the first worker we encounter when proceeding from node *G*. As in the standard bucket brigade protocol, when the last worker (the fastest worker) finishes his work at node *K*, he walks back upstream and takes over the work of his predecessor (the next slower worker), who will be located at some point on a subassembly line. This slower worker in turn walks back and takes over the work of *his* predecessor, and so on, until the first worker begins a new subcomponent at node *G*.

4. Two new issues

This adaptation of the bucket brigade protocol introduces two new issues that do not arise in simple linear assembly lines. First is the issue of travel. When following the bucket brigade protocol on the serialized assembly line, workers must travel from one subassembly line to another and these subassembly lines may be physically far apart. It is clear that the larger the distances between subassembly lines, the greater the potential for wasted productive capacity as workers travel between lines. For now we shall assume that this travel is not "too large"; later we shall show how to reduce such travel.

The total time to complete an instance of the product produced by the in-tree assembly network under the extended bucket brigade protocol is the sum of *productive time* and *unproductive time*. The productive time is the time spent exclusively in the assembly of the product, while the unproductive time is that spent traveling but not assembling (either forward to the next subassembly line or backward to get more work). For now we adopt the following assumption.

Assumption 3. The unproductive time is negligible compared to the productive time.

The second new issue is that the forward progress of a worker on the serialized assembly line can be halted if, on starting work on a subassembly line, he finds that a required subcomponent is

Table 1
Bucket brigade rules

Forward: Assemble your item moving from one subassembly line to the next according to the serialized ordering. If you are the last worker and complete an item or if your item is taken over by your successor then go **Back**. If you are *starved*—that is, if a required subcomponent is unavailable—then drop your work in the buffer at the end of the subassembly line you just completed and go **Back**.
Back: If you are the first worker then return to the first subassembly line according to the serialized ordering and start a new item and go **Forward**; otherwise walk back to your predecessor, take over his item, and go **Forward**.

The bucket brigade rules determine what each individual team member should do. This version of the rules accounts for a new phenomenon, the possibility of "starvation".

not available (the corresponding buffer is empty). For example, when a worker finishes a subcomponent D, according to the sequence of work shown in Fig. 4, he should proceed to line H. However, he might not be able to continue his work if buffer E is empty. We say the worker is *starved* and we extended the original bucket brigade protocol to handle this situation. The possibility of starvation raises the concern that worker capacity could be systematically diverted to the assembly of subcomponents without proportional completions of the final product. Is the bucket brigade system stable? Can buffers grow without bound?

5. Starvation is transient

With respect to any total ordering, the item currently being assembled by worker i partitions the subassembly lines of the in-tree into two sets: the set $L_i^{<}$ of subassembly lines whose completion points (and buffers) occur prior to the current position of the worker; and the set $L_i^{\geqslant}$, the current and remaining lines on the in-tree.

Definition 1. The item being assembled by worker i is fully provisioned if the following condition holds: each of the buffers of the subassembly lines in $L_i^{<}$ that are required for assembly in $L_i^{\geqslant}$ holds a subcomponent uniquely reserved for the assembly of that item. A bucket brigade is fully provisioned if each of the items being assembled is fully provisioned.

This definition can be checked worker-by-worker, examining the buffers in each case and labeling the contents. This requires $O(mn)$ steps, where m is the number of subassembly lines.

The importance of this idea lies in the following, which is a direct result of the definition.

Observation 1. If the bucket brigade is fully provisioned then the item represented by each worker will be completed without starvation among the workers.

The bucket brigade protocol works to establish a natural association between subcomponents in the buffers and instances of the product as follows: imagine that, when a new item is begun, it is assigned a unique, fixed identification number and this number is revealed to each worker when they take over production of the item. Imagine further that each worker labels any subcomponent he assembles with the identification number of the item for which he is currently responsible and thus any necessary subcomponents retrieved for the item also share the unique identifier.

Lemma 1. *Each new item that enters the bucket brigade system is fully provisioned and remains so as it progresses to completion.*

Proof. Because the serialized line observes the precedence constraints, each required subcomponent is assembled before it is needed and left in its buffer. □

Corollary 1. *If a bucket brigade is fully provisioned then it will remain fully provisioned.*

Proof. This follows because the natural association persists between items in the hands of workers and the subcomponents that were assembled for them. □

Lemma 2. *There can be no more than n instances of starving.*

Proof. By the extended bucket brigade protocol, each instance of starving triggers a walk-back, which starts a new instance of the product. After n walk-backs all n items in the system (in the hands of bucket brigade workers) are fully provisioned and so there can be no more starving. □

Theorem 1. *If workers are sequenced from slowest to fastest in the direction of the serialized assembly line then a bucket brigade on an in-tree will balance itself.*

Proof. Lemmas 1 and 2 ensure that a bucket brigade on an in-tree will quickly achieve a fully provisioned state regardless of the initial state of the system, hence any starvation is transient. Corollary 1 guarantees that the bucket brigade, upon reaching a fully provisioned state, will remain in a fully provisioned state. Thus due to Assumption 3, a bucket brigade operating in a fully provisioned state on an in-tree eventually behaves as a traditional bucket brigade on a linear assembly line independent of its initial state (see Theorem 3 of [4]). □

The following tells us that, under bucket brigades, the buffers cannot grow without bound. This will be important to establish the throughput of the system.

Lemma 3. *From any initial conditions (positions of the workers, state of buffers) the total number of subcomponents in buffers can never grow by more than mn before the bucket brigade is fully provisioned. After it is fully provisioned, the total population of subcomponents cannot grow beyond an additional mn.*

Proof. By the extended bucket brigade protocol, each instance of starvation triggers a walk-back, in which the starved worker in effect abandons the item on which he was working and goes back to take over the item of his predecessor. This leaves in the buffers no more than m subcomponents intended for the abandoned item. By Lemma 2 there can be at most n instances of starvation, during which the population of the buffers cannot grow more than mn.

After starving has ceased each new subcomponent in a buffer corresponds to a unique instance of the product; and at any time that instance of the product is carried by a unique worker who is downstream of the buffer. But there are only n workers and m buffers, so the total population of the buffers cannot grow by more than mn. □

Theorem 2. *If unproductive time is negligible in comparison to work-content, the bucket brigade line on an in-tree spontaneously balances itself, so that eventually each worker repeats the same interval of work-content and all subassembly lines produce at a common rate* $\sum_i v_i$ *which is the largest possible.*

Proof. From Assumptions 1 and 3 workers are always productive, and since by Lemma 3 the amount in the buffers is finite, eventually all productive capacity will be realized as output. By Theorem 3 of [4], each worker repeats an interval of work-content at a common cycle time and so each subassembly line produces the common rate $\sum_i v_i$. □

By Theorem 2 the long-run dynamics of bucket brigades on assembly in-trees are similar to those for linear assembly lines. Nevertheless, it is worth remarking that the short-term dynamics can differ in that bucket brigades on assembly in-trees are susceptible to a new form of disruption: if, in the midst of production, a subcomponent is found to be flawed and unsuitable for use, bucket brigades can proceed through a sequence of n instances of starvation before reëstablishing a fully provisioned state (Lemma 2), and, in the process, can generate spurious subcomponents. For example, consider M subassembly lines meeting at a common node at which final assembly begins. Assume the M lines are sequenced $1, 2, \ldots, M$ and imagine a single worker assembling each of these subcomponents in turn and depositing it in the appropriate buffer. If the subcomponent in buffer 1 is later found to be defective, the worker will not be able to proceed to final assembly, but will instead, under the bucket brigade protocol, retrace his path and then assemble another of each of the subcomponents $1, 2, \ldots, M$. The bucket brigade will have abandoned the item for which the flawed subcomponent was intended, along with all other subcomponents intended for it, and gone blindly back to begin construction of another item. In short, rather than try to recover in the fastest possible way, bucket brigades maintain simplicity and consistency of movement. Nevertheless, Lemma 2 guarantees that bucket brigades will recover quickly and Lemma 3 guarantees that not "too many" spurious subcomponents will be produced during recovery.

6. What is the best total order?

There are many ways to realize a total order of subassembly lines that is consistent with the partial order of these lines on the in-tree. Here, we discuss two criteria by which one total order may be preferred to another. The first criterion is *Unproductive Time*, which is a concern if travel between subassembly lines are significant (that is, Assumption 3 fails to hold). Total orders that minimize travel distance are preferred. The second criterion, which we call *Progress Toward Subgoals*, favors total orders in which partially completed subcomponents are moved along as far as possible before new subcomponents are introduced.

6.1. Unproductive time

Following the extended bucket brigade protocol, workers on the in-tree assembly network can either travel forward (moving in the direction consistent with the total order of lines to assemble products) or travel backward (moving in the reverse direction to get more work). Forward travel can be productive or unproductive. Backward travel is always unproductive.

We assume that the best (minimum distance) path from the end of a subassembly line to the start of any other subassembly line is known. In addition, we assume that these shortest paths are used in both forward and backward travel:

Assumption 4. When traveling backward, a worker retraces the forward path.

This assumption is conservative, in that it ignores opportunities to take short-cuts when walking back to get work from a predecessor. However, in practice such opportunities are often not available—potential short-cuts may be hampered by conveyors or shelving that limit movement between lines. Furthermore, a worker must *find* his predecessor to get more work, so that retracing the forward path may be necessary simply to preserve the bucket brigade protocol.

Due to Assumption 4, the total forward and backward travel required to complete each item is independent of the number of workers on the line. Consequently it is sufficient to study a 1-worker system.

Observation 2. The problem of finding a total order of subassembly lines for which the travel is minimized for an n-worker system is equivalent to the same problem for a 1-worker system.

The problem of finding a total ordering of minimum length can now be modeled as the stacker crane problem with precedence constraints. In the original, unconstrained stacker crane problem [10], we are given a set $\mathscr{N}$ of nodes (together with the corresponding distance matrix) and a set $\mathscr{A}$ of arcs, where each arc is an ordered pair of nodes. The objective is to search for a tour $(j_1, j_2, \ldots, j_m)$, where $j_1 = j_m$, with minimum length subject to the constraint that for each arc $(u, v) \in \mathscr{A}$, there is some q such that $j_q = u$ and $j_{q+1} = v$. Note that a tour may visit a node (other than node j_1) in $\mathscr{N}$ more than one time. The stacker crane problem with precedence constraints is the stacker crane problem with additional constraints requiring that certain arcs must be visited before others. In our problem these precedence constraints represent the precedence relations between the subassembly lines on the in-tree.

The stacker crane problem is NP-complete [10]. Frederickson et al. [10] suggested two polynomial time approximation algorithms to solve the stacker crane problem. In particular, the first algorithm (called the *large arcs* algorithm) is suitable for the case when the total length of arcs in $\mathscr{A}$ is large compared to the optimal tour length. It returns a solution with cost no greater than $3C^* - 2C_{\mathscr{A}}$, where C^* and $C_{\mathscr{A}}$ represent the length of the optimal tour and the total length of arcs in $\mathscr{A}$ respectively. Furthermore, the algorithm terminates in $O(|\mathscr{A}|^3)$ time. Interested readers can refer to [10] for the details.

The large arcs algorithm can be adapted to solve the stacker crane problem with precedence constraints described above. The adaptation is straightforward and we omit the details here other than to observe that the adapted algorithm has the same worst-case performance guarantee and time complexity as the original. Interested readers can refer to [12] for more details.

Although it is NP-complete to find a total order on a subtree that requires minimal travel, all real in-tree assembly networks with which we are familiar are fairly simple and would not require actual application of the algorithmic procedure described above to generate practical solutions.

6.2. Progress toward subgoals

Another criterion that may be considered (instead of, or in addition to, travel distance) is selecting a total order that maintains progress toward subgoals. Here we favor total orders in which each subcomponent is completed as much as possible before new subcomponents are begun. To illustrate this criterion, suppose there is a worker working on line *D* in Fig. 1 and assume that all the buffers on the in-tree are empty. When the worker finishes his work on line *D*, he may begin a new subcomponent at any of nodes *E*, *A*, *B*, *C*, or *G*. Taking the continuity of work flow into account, it is preferable for him to begin to work on line *E* and then line *H* before proceeding to another subtree. The intuition is to complete the current subcomponent before processing other less-directly-related subcomponents. This may be formalized in the following way: *Before assembling subcomponent j, recursively assemble all subcomponents that immediately precede j.* The reader will recognize this as a *postorder traversal* of the arcs of the in-tree [9].

We define the *preferred set* as the set of total orders that maintain progress towards subgoals in the assembly process. According to this definition, the total orders of nodes shown in Figs. 3 and 5(a) are in the preferred set. Fig. 5(b) gives an example that is not in the preferred set because a worker finishing a subcomponent *E* will begin a subcomponent *A* instead of completing the work on the left subtree rooted at node *J*.

7. Related work

In 1994 Gershwin observed that "there has been relatively little written on Assembly/Disassembly networks compared with the vast queuing theory literature" [11]. We find the situation unchanged today. The literature of which we are aware models the assembly network as a queuing network, typically with iid exponential processing times (for example, [11]). Typically the emphasis is on describing steady state behavior of the system, if any, and measuring work-in-process and throughput.

Some previous work seeks balance by the assignment of work-content. Most relevant to our work is that of Baker et al., who considered an assembly system in which two subcomponents are produced at different stations and then joined at third station [3]. The processing times at all stations are stochastic and follow the same distribution. No buffers are used in the system, but the assembly station can accept completed subcomponents while awaiting other subcomponents.

Baker et al. focused on the cases in which the total work-content is constant and is equal to the number of stations in the system and the coefficient of variation at each station is fixed. For this model Baker et al. investigated how to allocate the total work-content (sum of mean processing times at all stations) to the stations so that the throughput of the system is maximized. They found that the assembly system should be "unbalanced" in the direction of assigning less work to the assembly

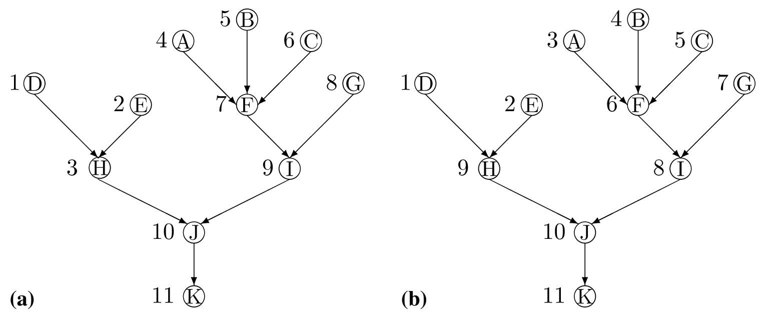

Fig. 5. (a) A total order of nodes that is in the preferred set: the work sequence constructed in this order maintains progress towards subgoals in the assembly process. (b) A total order of nodes that is not in the preferred set: this order does not maintain progress towards subgoals because a worker finishing his work on line *E* will proceed to line *A* instead of line *H*. This violates the condition of continuity of work flow.

station and more to component stations. They also considered a generalized system in which two feeder lines merge at an assembly station. Each feeder line consists of an initial work station and a final work station, where the latter feeds the assembly station. Their results indicate that throughput is maximized by allocating decreasing amounts of work-content closer to assembly.

In contradistinction, we have assumed that the work-content has already been allocated among the work stations, but that—as to be expected in the real world—the resulting balance and synchronization will be imperfect, both moment-to-moment due to stochastic variance and in the long-run, due to impossibility of dividing the work perfectly. Bucket brigades can smooth over those imperfections to achieve, and dynamically maintain, near perfect balance.

8. Conclusions

The main contribution of this paper is to show how the idea of bucket brigades [4] can be applied to in-tree assembly networks, so that even relatively complex assembly systems can enjoy the benefits of self-balancing. We achieve this by conceptually converting the work-content on an assembly network to a one-dimensional sequence of work upon which the adapted bucket brigade protocol can be executed. The system converges to a stationary state in which every worker repeats the same portion of work-content (possibly across several subassembly lines) on each successive instance of the product. Furthermore, to a good approximation, the throughput of the system attains the maximum possible.

It is worth remarking that some managerial issues that are trivial on simple, linear bucket brigades may present practical problems on complicated in-trees. For example, workers must follow a common sequence of work, which means that the sequence of subassembly lines must be either memorized or else marked clearly for both forward and backward movement. Similarly, on walking back to get more work, workers must be able to find and recognize their predecessors.

If travel time is significant then of course the throughput of the assembly system would be diminished because workers would spend more time in unproductive travel and less in actual assembly. In addition, significant travel times could disrupt the self-balancing, and in particular the tendency of workers to repeat the same interval of work-content. This could happen if, for example, the in-tree is such that when a worker walks back from position x the time to travel back is very different from the time required from position $x+\epsilon$.

In practice it may be quite easy to implement bucket brigades on in-tree assembly systems because real assembly trees are fairly simple, at least in our experience. For example, a motivating application for this paper was an assembly in-tree to produce large screen televisions for Mitsubishi [5]. It was a simple "Y" shape and so the appropriate total ordering was obvious.

Acknowledgements

Bartholdi was supported in part by the Office of Naval Research (grant #N00014-89-J-1571), the National Science Foundation (grant #DMI-9908313), The Logistics Institute at Georgia Tech and its sister organization, The Logistics Institute Asia Pacific in Singapore. Eisenstein was supported by the Graduate School of Business at the University of Chicago.

References

[1] R.K. Ahuja, T.L. Magnanti, J.B. Orlin, Network Flows: Theory, Algorithms, and Applications, Prentice Hall, 1993, ISBN 0-136-17549-X.

[2] D. Armbruster, E.S. Gel, Bucket brigades when worker speeds do not dominate each other uniformly, Working paper, Department of Industrial Engineering, Arizona State University, 2002.

[3] K.R. Baker, S.G. Powell, D.F. Pyke, Optimal allocation of work in assembly systems, Management Science 39 (1) (1993) 101–106.

[4] J.J. Bartholdi III, D.D. Eisenstein, A production line that balances itself, Operations Research 44 (1) (1996) 21–34.

[5] J.J. Bartholdi III, D.D. Eisenstein, The Bucket Brigade Web Page, www.isye.gatech.edu/~jjb/bucket-brigades. html.

[6] J.J. Bartholdi III, L.A. Bunimovich, D.D. Eisenstein, Dynamics of two and three-worker "bucket brigade"

production lines, Operations Research 47 (3) (1999) 488–491.

[7] J.J. Bartholdi III, D.D. Eisenstein, R.D. Foley, Performance of bucket brigades when work is stochastic, Operations Research 49 (5) (2001) 710–719.

[8] I. Baybars, A survey of exact algorithms for the simple assembly line balancing problem, Management Science 32 (8) (1986) 909–932.

[9] T.H. Corman, C.E. Leiserson, R.L. Rivest, Introduction to Algorithms, MIT Press, 1990, ISBN 0262-03141-8.

[10] G.N. Frederickson, M.S. Hecht, C.E. Kim, Approximation algorithms for some routing problems, SIAM Journal on Computing 7 (2) (1978) 178–193.

[11] S.B. Gershwin, Manufacturing Systems Engineering, PTR Prentice Hall, 1994, ISBN 0-13-560608-X.

[12] Y.F. Lim, Ph.D. Thesis, School of Industrial and Systems Engineering, Georgia Institute of Technology, 2003.

[13] A. Scholl, Balancing and Sequencing of Assembly Lines, second ed., Physica-Verlag, 1999, ISBN 3-790-81180-7.