Distributed, Low Contention Task Allocation[*]

(EXTENDED ABSTRACT)

SARANTOS KAPIDAKIS[†] MARIOS MAVRONICOLAS[‡]

MAY 1, 1995

Abstract

Designing a good *task allocation* algorithm faces the challenge of allowing high levels of *throughput,* so that tasks are executed fast and processor parallelism is exploited, while still guaranteeing a low level of *memory contention,* so that performance does not suffer because of limitations on processor-to-memory bandwidth. In this work, we present a comparative study of throughput and contention guarantees provided by *load balancing networks,* a new class of distributed, asynchronous algorithms for real-time task allocation in shared memory multiprocessors. Load balancing networks generalize *balancing networks,* to accomodate tasks with varying completion times.

On the theoretical side, we formulate precise and crisp definitions for capturing the quality of load balancing provided by general task allocation algorithms; we use these definitions for formally evaluating the throughput performance of specific constructions of load balancing networks that we propose. Furthermore, we introduce a formal, complexity-theoretic measure of contention incurred by tasks with varying completion times, and use it to analyze the contention performance of these constructions. Our theoretical results display precise and subtle trade-offs between throughput and contention performances for load balancing networks.

On the practical side, we propose an experimental platform for evaluating the actual performance of load balancing networks through a series of carefully designed experiments that simulate these networks on real shared memory multiprocessor machines. Our experimental approach encompasses a rigorous methodology for randomly generating tasks that are not merely "random", but rather belong to common classes of tasks such as *periodic* and *sporadic*. Our experimental results reveal that load balancing networks substantially outperform in performance classical, centralized methods for task allocation.

---

[*] This work has been supported by ESPRIT Basic Research Project # 8144 (LYDIA – Load Balancing on High-Performance Parallel and Distributed Systems) and funds for the promotion of research at University of Cyprus (research project "Load Balancing Problems on Shared Memory Multiprocessor Architectures").

[†] Department of Computer Science, University of Crete, Heraklion 71110, Greece, and Institute of Computer Science, FORTH, Heraklion 71110, Greece. Tel No. + 308 - 1391678. Fax No. + 308 - 1391601. E-mail address: `sarantos@csi.forth.gr`

[‡] Department of Computer Science, University of Cyprus, Nicosia 1678, Cyprus. Tel No. + (357) 2365872. Fax No. + (357) 2360881. Part of the work of this author was performed while visiting Institute of Computer Science, Foundation for Research and Technology – Hellas. E-mail address: `mavronic@csi.forth.gr`

1 Introduction

1.1 Motivation–Overview

Managing the performance of distributed, real-time computing systems is currently a serious challenge because of the complexity of these systems and the diversity of new applications. In some of the simplest and most tractable instances, real-time task allocation and adaptive management of computational resources in a large, heterogeneous distributed system is required in order to enhance performance of concurrently running application programs. Traditional solutions to task allocation problems in large scale multiprocessors have aimed at maximizing *throughput,* the rate at which task execution is completed by processors (see, e.g., [5, 12, 18, 20, 23]). Throughput has been previously studied in the real-time systems community as a performance measure for real-time allocation of tasks (see, e.g., [7, 17, 19, 21, 27, 28]);

In this work, we propose a new approach to solving task allocation problems by introducing *load balancing networks,* a new class of networks that can be used to allocate tasks to processors in a way that computational load is balanced. Besides maximizing throughput, we are interested in minimizing *contention,* a new measure of performance that captures performance degradation due to simultaneous access of shared memory; this measure has recently been studied in the shared memory multiprocessing community (see, e.g., [3, 2, 1, 9, 11, 24]. Our work attempts to join these previous research efforts in these two different research communities by comparing and contrasting throughput and contention performance of load balancing networks.

1.2 Detailed Description

Load balancing networks, like balancing networks (see, e.g., [4, 8]) are constructed from simple computing elements called *load balancers,* with one or two inputs and two outputs, connected to one another by wires. However, while a balancing network counts on its output wires any number of input tokens, a load balancing network can handle any number of input tasks with a specified completion time and distribute their computational load evenly among its output wires. Thus, balancing networks are but a special case of load balancing networks where all tasks are of unit completion times and outputs are ordered, although in both cases tokens or tasks may arrive at arbitrary real times, be distributed unevenly among the input wires and propagate through the network asynchronously.

Figure 1 provides an example of a sequential execution of a six-input, six-output load balancing network. A load balancer is represented by two dots and a vertical line. Intuitively, a load balancer is a balancing mechanism, forwarding each input task to its top or bottom output wires, according to which of these wires has so far accepted the largest sum of completion times of input tasks. Thus, it balances sums of completion times of jobs on its output wires. In the example of Figure 1, input tokens arrive on the network's input wires and traverse the network

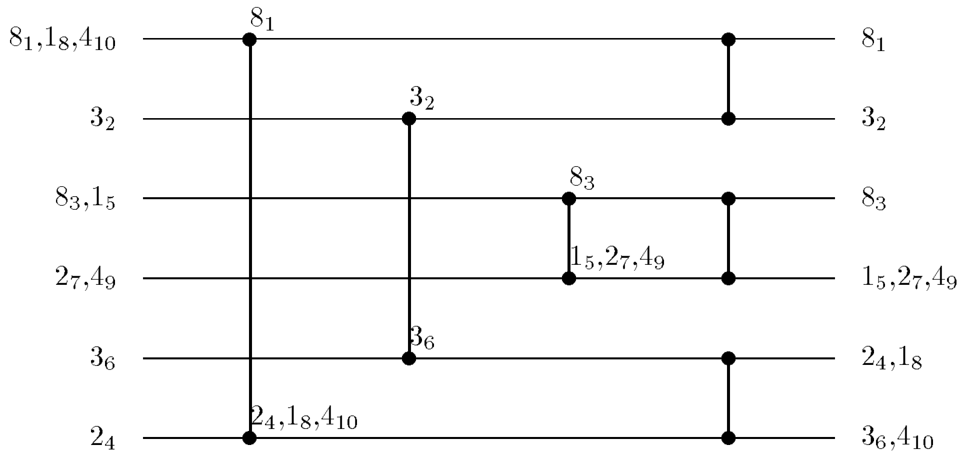

Figure 1: A load balancing network

one after the other. For convenience, we have indexed them by the order of their arrival (these indices are *not* used by the network).

Load balancing networks achieve high level of throughput by decomposing information about processor loads into pieces that can be processed in parallel. The obvious benefits of this are elimination of sequential bottlenecks and reduction of memory contention. Load balancing networks are also *non-blocking*: processes that undergo halting failures or delays while using a load balancing network do not prevent other processes from making progress.

We use load balancers to construct and analyze load balancing networks that reduce contention and permit concurrent and real-time assignment of tasks to processors. We evaluate our constructions against formal measures of the quality of load balancing achieved.

Furthermore, we introduce a new formal, complexity-theoretic measure of contention incurred by shared memory multiprocessor algorithms that solve task allocation problems. Our measure generalizes one proposed by Dwork *et al.* [11] to accomodate tasks with varying completion times. More specifically, each time a task passes through a load balancer, we charge a (real) number of stalls equal to its completion time to each task that is pending at the load balancer and awaiting to be processed. We consider a sequence of tasks and amortize the number of stalls charged during the passage of the tasks through the load balancing network by dividing by the total sum of durations of all the tasks. The case where all tasks are of unit completion times corresponds to the contention measure introduced by Dwork *et al.* [11]. We analyze each of our constructions with respect to the contention measure we introduce.

It is our belief that an experimental approach for evaluating task allocation techniques is a good way to go. To that end, we develop a simulator of a shared memory multiprocessor machine that operates asynchronously and receives tasks to be assigned to processors. Expressing the methodology of choosing input tasks is important in our approach – as merely a “random” sequence of input tasks may be "self-balancing" by itself, whereas in real multiprocessor machines, there is a "bursty" component where some tasks arrive with load requirements much larger than the average and a possible "periodic" component for long-lasting jobs. We do not believe that experimentations with "random" inputs (e.g., [14, 22]) are representative of the actual performance. We rather believe that a good experimental work should not only take "random" input tasks into account, but also "highly biased" (in a specified way) as well as "periodic" (highly correlated) and mixtures of these. To make experimentation less of a "myth", we carefully specify our choice of input tasks and assumptions required behind or justifying our choices. Our simulations reveal that load balancing networks significantly outperform conventional task allocation techniques on such realistic inputs.

1.3 Related Work

Aiello *et al.* [1] introduce the notion of a *w-balancer,* which guarantees that "the output flows at time t are balanced within maximum weight M of tokens output by t" [1, Section 4]; this is a formal throughput property of a load balancer which is satisfied by the MIMD shared memory implementation of it described in Section 3. The notions of load balancer and w-balancer are essentially identical, and the two works have been performed independently [25] (the first version of our work was publicized in June 1994).

Aiello *et al.* take as their performance index the property that the output of a weight balancing network is K-smooth for some constant K. Definition 3.4 provides an explicit generalization of this property that allows K to be any function of the maximal job. However, apart from the imbalance between different output wires, we also consider makespan (see Definitions 3.1 and 3.2) as an index of performance.

Among our proposed constructions and those in [1], the binary tree is the only one that allows the imbalance to be a constant multiple of the maximum job M (see discussion in Section 5 on improving actual performance of the binary tree).

Aiello *et al.* propose the w-reverse butterfly and (r,w)-butterfly constructions. The w-reverse butterfly and the bitonic merger network achieve asymptotically the same smoothness factor of $\Theta(M \lg w)$, although the smoothness of the bitonic merger is better by a factor of two, and, in our opinion, it enjoys a simpler construction. The (r,w)-butterfly employs randomization to achieve a better smoothness factor of $\Theta(M \lg \lg w)$ with a certain high probability.

The smoothness factor of $M \lg w$ shown for the bitonic merger network (Theorem 5.2) can sometimes be better than the smoothness factor of $\Theta(M^2)$ shown for an even "larger" network, namely the entire Batcher's bitonic network [1, Corollary 3.5]. This implies that the general result on the relation between the smoothness factor of a balancing network and its isomorphic load balancing network [1, Theorem 4.4] might not be the strongest possible in all cases. Clearly, Theorem 5.2 together with [1, Corollary 4.5] implies that Batcher's bitonic network achieves a smoothness factor of $\Theta(M \min\{M, \lg w\})$.

The rest of this paper is organized as follows. In Section 2, we present definitions for load balancing criteria. In Section 3, we introduce load balancing networks, while constructions of such networks are presented in Section 4. Theoretical and experimental performance analyses for these constructions are included in Sections 5 and 6, respectively.

2 Criteria for Load Balancing

We consider collections $\mathbf{P} = \{p_0, p_1, \ldots, p_{n-1}\}$ and $\mathbf{C} = \{c_0, c_1, \ldots, c_{w-1}\}$ of n producers and w consumers, respectively. The producers produce *jobs*[1] at some arbitrary rate; the jobs should be performed by the consumers. Associated with each job j is its *length* t_j representing the time it takes to completion by a consumer. Henceforth, we will abuse notation and use t_j to represent both job j and its length.

A *job set* $\mathbf{X}$ is a finite set of jobs. A *load balancing algorithm* is a one-to-many function $\mathcal{A}$ which takes as input a job set $\mathbf{X}$ and maps each job in the job set onto a consumer $c \in \mathcal{C}$. Any image $\lambda(\mathbf{X})$ of $\mathbf{X}$ under $\mathcal{A}$ will be called a *job assignment for* $\mathbf{X}$. Given a job assignment $\lambda(\mathbf{X})$ for $\mathbf{X}$, define, for each $i \in [w]$, $\lambda_i(\mathbf{X})$ to be the sum of the jobs in the set assigned to consumer c_i; call $\lambda_i(\mathbf{X})$ the *load at consumer* c_i.

Define the *makespan of the job set* $\mathbf{X}$ *induced by job assignment* $\lambda(\mathbf{X})$ to be the maximum over all $i \in [w]$ of $\lambda_i(\mathbf{X})$. Intuitively, the makespan represents the time, under a particular assignment of jobs to processors, at which the latest finishing job is completed. Since a load balancing algorithm may produce more than one job assignments on a given job set, one would be interested in the maximum over all of its possible assignements for a job set $\mathbf{X}$ of $\lambda(\mathbf{X})$, defined as the *makespan of the job set* $\mathbf{X}$ *induced by the load balancing algorithm* $\mathcal{A}$. Naturally, one would like to have a load balancing algorithm that minimizes makespan as much as possible. Define the *makespan of job set* $\mathbf{X}$ to be the minimum, over all load balancing algorithms $\mathcal{A}$, of the makespan of $\mathbf{X}$ induced by $\mathcal{A}$.

Definition 2.1 *A load balancing algorithm* $\mathcal{A}$ minimizes makespan *if for any job set* $\mathbf{X}$*, the makespan of* $\mathbf{X}$ *induced by* $\mathcal{A}$ *is equal to the makespan of* $\mathbf{X}$*.*

One may relax Definition 2.1 by only insisting that a certain bound holds on the makespan guaranteed by a particular load balancing algorithm.

Definition 2.2 *For any function* $g : \mathbf{N} \rightarrow \mathbf{N}$*, a load balancing algorithm* $\mathcal{A}$ bounds makespan by g *if for any job set* $\mathbf{X}$*, the makespan of* $\mathbf{X}$ *induced by* $\mathcal{A}$ *is at most the image of the makespan of* $\mathbf{X}$ *under* g*.*

---

[1] Throughout, we interchangeably use the terms job and task.

Define the *maximal difference of the job set* $\mathbf{X}$ *induced by job assignment* $\lambda(\mathbf{X})$ to be the maximum over all pairs i and $j \in [w]$ of $|\lambda_i(\mathbf{X}) - \lambda_j(\mathbf{X})|$. Intuitively, the maximal difference is a measure of the "imbalance" between different processors under a particular assignment of jobs to them. The maximum, over all possible job assignements of a load balancing algorithm $\mathcal{A}$ for a job set $\mathbf{X}$, of $|\lambda_i(\mathbf{X}) - \lambda_j(\mathbf{X})|$ will be called the *maximal difference of the job set* $\mathbf{X}$ *induced by the load balancing algorithm* $\mathcal{A}$. Naturally, one would like to have a load balancing algorithm that minimizes maximal difference as much as possible. Define the *maximal difference of job set* $\mathbf{X}$ to be the minimum, over all load balancing algorithms $\mathcal{A}$, of the maximal difference of $\mathbf{X}$ induced by $\mathcal{A}$.

Definition 2.3 *A load balancing algorithm* $\mathcal{A}$ minimizes maximal difference *if for any job set* $\mathbf{X}$*, the maximal difference of* $\mathbf{X}$ *induced by* $\mathcal{A}$ *is equal to the maximal job of* $\mathbf{X}$*.*

One may relax Definition 2.3 by only insisting that a certain bound holds on the maximal difference guaranteed by a particular load balancing algorithm.

Definition 2.4 *For any function* $g : \mathbf{N} \rightarrow \mathbf{N}$*, a load balancing algorithm* $\mathcal{A}$ bounds maximal difference by g *if for any job set* $\mathbf{X}$*, the maximal difference of* $\mathbf{X}$ *induced by* $\mathcal{A}$ *is at most the image of the maximal job of* $\mathbf{X}$ *under* g*.*

3 Load Balancing Networks

Load balancing networks are constructed from wires and computing elements called load balancers, in a way similar to the one in which balancing networks [4, 8] are constructed from wires and balancers, and sorting networks [10] are constructed from wires and comparators.

A *load balancer* is a computing element with two output wires and either one or two input wires. Input and output wires are occupied by producers and consumers, respectively. Producers push jobs onto the input wires at arbitrary times, and the load balancer outputs the jobs on its output wires to be processed by consumers. Intuitively, one may think of a load balancer as a job scheduler, effectively balancing sums of completion times of jobs that have been output on its output wires.

The following definitions are tailored for a two-input balancer. Denote by X_i, $i \in \{0, 1\}$, the set of jobs ever received on its ith input wire, and similarly by Y_i, $i \in \{0, 1\}$, the set of jobs ever output on its ith output wire. Throughout the paper, we will abuse this notation and use X_i (resp., Y_i) as both the set of jobs and the set of completion times of jobs ever received (resp., output) on the ith input (resp., output) wire. Let the *state* of a two-input load balancer at a given time be defined as the sets of jobs on its input and output wires. We can now formally state the safety, liveness and throughput properties for a two-input load balancer. (1) In any state, $Y_0 \cup Y_1 \subseteq X_0 \cup X_1$ (i.e., a load balancer never creates output jobs), (2) given any finite sets X_0 and X_1 of jobs input to the load balancer, it is guaranteed that within a finite amount of time, it will reach a *quiescent* state: a state in which the union of the input sets of jobs is equal to the union of the output sets of jobs; that is, in a quiescent state, $Y_0 \cup Y_1 = X_0 \cup X_1$, and (3) in any quiescent state, the absolute difference $|\sum Y_0 - \sum Y_1|$ between sums of jobs on the output wires is at most the maximum job ever received on an input wire. Corresponding definitions for a one-input balancer follow immmediately.

A *load balancing network of input width t and output width w* is a collection of balancers, where output wires are connected to input wires, having t designated input wires $x_0, x_1, \ldots, x_{t-1}$ (not connected to output wires of balancers), w designated output wires y_0, y_1, …, y_{w-1} (not connected to input wires of balancers), and containing no cycles. Let the state of a balancing network at a given time be defined as the collection of the states of all its component balancers. The safety and liveness properties of a network follow naturally from its definition and the corresponding properties of balancers.

On a MIMD shared memory multiprocessor architecture, a load balancing network is implemented as a shared data structure, where balancers and wires are represented by records and wires from one record to another, respectively. Each record has three fields: $diff$ is an integer variable, holding the absolute difference $|\sum Y_0 - \sum Y_1|$ between sums of jobs on its output wires; *toggle* is a Boolean variable signifying the output wire with the currently maximum sum of jobs ever output on it; *next* is a two-element array of pointers to successor balancers. Each of the machine's n asynchronous producers runs a program that repeatedly traverses the data structure from some input pointer (either preassigned or chosen at random) to some output pointer, each time shepherding a new job through the network. Each time a job j arrives at a load balancers, its length t_j is compared to $diff$. If t_j is found to be greater than or equal to $diff$, $diff$ is updated to $t_j - diff$ and *toggle* is toggled and the job proceeds to the corresponding next balancer according to *next*; if t_j is found to be less than $diff$, $diff$ is updated to $diff - t_j$, and the job proceeds to the corresponding next balancer, halting when it reaches a "leaf".

The limitation on the number of concurrent producers implies a limitation on the number of jobs concurrently traversing the network at any given time: $\sum_{i=0}^{t-1} \sum X_i - \sum_{j=0}^{w-1} \sum Y_i \leq n$. Consider an execution of a load balancing network $\mathcal{N}$ entering a quiescent state after m jobs pass through it. Each time a job passes through a load balancer, all tokens pending at this load balancer incur a *real stall,* with value equal to the completion time of the passing job, modeling their delay due to contention with each other. The number of stall steps has been introduced in [11] as a measure of contention for the special case where all jobs are of unit completion times. The *contention incurred by the traversal of m jobs through the network $\mathcal{N}$ at concurrency n,* denoted $cont(m, n, \mathcal{N})$, is the maximum number of real stalls, over all possible executions, induced by an adversary scheduler. The *amortized contention of the network $\mathcal{N}$ at concurrency n,* denoted $cont(n, \mathcal{N})$, is the limit of $cont(m, n, \mathcal{B})$ divided by m, as m goes to infinity.

Criteria for throughput guarantees of load balancing algorithms introduced in Section 2 specialize in a straightforward way to load balancing networks and yield:

Definition 3.1 *A load balancing network* $\mathcal{N}$ minimizes makespan *if for any set of job sets* X_0, X_1, …, $\mathbf{X}_{t-1}$, *the makespan of* $\mathbf{X}$ *induced by* $\mathcal{N}$ *in any of its quiescent states is equal to the makespan of* $X_0 \cup X_1 \cup \ldots \cup X_{t-1}$.

Definition 3.2 *For any function* $g : \mathbf{N} \rightarrow \mathbf{N}$, *a load balancing network* $\mathcal{N}$ bounds makespan by g *if for any set* $\mathbf{X}$ *of job sets* X_0, X_1, …, $\mathbf{X}_{t-1}$, *the makespan of* $\mathbf{X}$ *induced by* $\mathcal{N}$ *in any of its quiescent states is at most the image of the makespan of* $X_0 \cup X_1 \cup \ldots \cup X_{t-1}$. *under* g.

Definition 3.3 *A load balancing network* $\mathcal{N}$ minimizes maximal difference *if for any set* $\mathbf{X}$ *of job sets* X_0, X_1, …, X_{t-1}, *the maximal difference of* $\mathbf{X}$ *induced by* $\mathcal{N}$ *in any of its quiescent states is equal to the maximal job of* $\mathbf{X}$.

Definition 3.4 *For any function* $g : \mathbf{N} \rightarrow \mathbf{N}$, *a load balancing network* $\mathcal{N}$ bounds maximal difference by g *if for any set* $\mathbf{X}$ *of job sets* X_0, X_1, …, X_{t-1}, *the maximal difference of* $\mathbf{X}$ *induced by* $\mathcal{N}$ *is at most the image of the maximal job of* $\mathbf{X}$ *under* g.

4 Constructions of Load Balancing Networks

Naturally, load balancing networks are interesting only if they can be constructed. In this Section, we present two constructions of load balancing networks and analyze their load balancing guarantees.

4.1 Binary Tree

Our first construction is extremely simple: it consists of a binary tree $\mathcal{B}_w$ with input width one and output width w, where w (the number of its leaves) is a power of two; a one-input two-output balancer occupies each of its nodes.

Observation 4.1 *Define the function* $g : \mathbf{N} \rightarrow \mathbf{N}$ *by* $g(x) = x$. *Then, the network* $\mathcal{B}_w$ *does not bound maximal difference by* g.

Proof: We mention a counterexample to the opposite argument: Consider a sequential execution of the network $\mathcal{B}_4$ on the (ordered) input $\mathbf{X} = \{4_1, 5_2, 4_3, 5_4, 4_5, 5_6\}$. It is straightforward to verify that $Y_0 = \{4_1, 4_5\}$, $Y_1 = \{4_3\}$, $Y_2 = \{5_2, 5_6\}$ and $Y_3 = \{5_4\}$. We have that $\sum Y_2 - \sum Y_3 = 6$, but the maximum job of $\mathbf{X}$ is 5. Thus, on this example, the network $\mathcal{B}_4$ does not bound maximal difference by g. ■

Observation 4.2 *The network* $\mathcal{B}_w$ *does not minimize makespan.*

Proof: We mention a counterexample to the opposite argument. We use the same counterexample as in the proof of the previous observation. Consider a sequential execution of the network $\mathcal{B}_4$ on the (ordered) input $\mathbf{X} = \{4_1, 5_2, 4_3, 5_4, 4_5, 5_6\}$. It is straightforward to verify that $Y_0 = \{4_1, 4_5\}$, $Y_1 = \{4_3\}$, $Y_2 = \{5_2, 5_6\}$ and $Y_3 = \{5_4\}$. The makespan of this is $\sum Y_2 = 10$, which is not the minimum makespan, as at least the following assignment would yield a smaller makespan, 9: $Y_0 = \{4_1, 5_6\}$, $Y_1 = \{5_2\}$, $Y_2 = \{4_3, 4_5\}$ and $Y_3 = \{5_4\}$. Thus, on this example, the network $\mathcal{B}_4$ does not minimize makespan. ■

On the positive side, we show by induction on w:

Theorem 4.1 *Define the function $g : \mathbf{N} \rightarrow \mathbf{N}$ by $g(x) = 2x$. Then, the network $\mathcal{B}_w$ bounds maximal difference by g.*

4.2 Bitonic Merging Network

In this Section, we describe the construction and show some properties of Batcher's classical bitonic network [6]; our presentation borrows from [9].

Fix throughout w to be an integer of the form 2^{k+1}, for any integer $k \geq 0$.

The construction of the bitonic network $\mathcal{S}^{(w)}$ uses the bitonic merger network $\mathcal{M}^{(w)}$, whose construction is described next, as a basic module.

The load balancing network $\mathcal{M}^{(w)} : \mathbf{X}^{(w)} \rightarrow \mathbf{Y}^{(w)}$, called *bitonic merger,* is defined inductively as follows. For the base case, where $w = 4$, $\mathcal{M}^{(4)}$ is the "cascade" of two layers:

- A layer $\mathcal{M}_2^{(4)} : \mathbf{X}^{(4)} \rightarrow \mathbf{Z}^{(4)}$ consisting of two load balancers $b_0^{(2)}, b_1^{(2)}, \ldots, b_{p-1}^{(2)}$, where load balancer $b_i^{(2)}$ receives inputs x_i and x_{4-1-i} and produces outputs z_i and z_{4-1-i}, $i \in \{0, 1\}$.
- A layer $\mathcal{M}_{2'}^{(4)} : \mathbf{Z}^{(4)} \rightarrow \mathbf{Y}^{(4)}$ consisting of two load balancers $b_{up}^{(2)} : \mathbf{Z}_{up}^{(2)} \rightarrow \mathbf{Y}_{up}^{(2)}$, and $b_{down}^{(2)} : \mathbf{Z}_{down}^{(2)} \rightarrow \mathbf{Y}_{down}^{(2)}$.

Assume inductively that we have constructed $\mathcal{M}^{(w/2)}$, where $w \geq 8$; we show how to construct $\mathcal{M}^{(w)}$. The network $\mathcal{M}^{(w)}$ is the "cascade" of:

- a network $\mathcal{N}^{(w)} : \mathbf{X}^{(w)} \rightarrow \mathbf{Z}^{(w)}$ which is the "parallel composition" of two networks $\mathcal{M}_{eo}^{(w/2)} : \mathbf{X}_{eo}^{(w/2)} \rightarrow \mathbf{Z}_e^{(w/2)}$ and $\mathcal{M}_{oe}^{(w/2)} : \mathbf{X}_{oe}^{(w/2)} \rightarrow \mathbf{Z}_o^{(w/2)}$;
- a layer $\mathcal{L}^{(w)} : \mathbf{Z}^{(w)} \rightarrow \mathbf{Y}^{(w)}$ consisting of $w/2$ load balancers $b_0^{(2)}, b_1^{(2)}, \ldots, b_{w/2-1}^{(2)}$, where load balancer b_i receives inputs z_{2i} and z_{2i+1} and produces outputs y_{2i} and y_{2i+1}, $i \in [w/2]$.

Notice that the construction of $\mathcal{M}^{(w)}$ implies that $depth(\mathcal{M}^{(4)}) = 2$, while for $w > 4$,

$$depth(\mathcal{M}^{(w)}) \;=\; depth(\mathcal{N}^{(w)}) + depth(\mathcal{L}^{(w)}) \;=\; depth(\mathcal{M}^{(w/2)}) + 1 \;,$$

implying:

Proposition 4.2 *For all $w \geq 4$, $depth(\mathcal{M}^{(w)}) = \lg w$.*

We can show:

Theorem 4.3 *The networks $\mathcal{M}_w$ and $\mathcal{S}_w$ bound maximal difference by the same function g.*

Theorem 4.4 *For any constant $c > 0$, define the function $g_c : \mathbf{N} \rightarrow \mathbf{N}$ by $g_c(x) = cx$. Then, the network $\mathcal{M}_w$ does not bound maximal difference by g_c.*

Sketch of proof: By a counterexample where the maximal difference will be $\lg w$ times the length of the maximum job; as $\lg w$ is not bounded by a constant, then there can be no constant c that bounds maximal difference by g_c. ■

By induction on w, we can show:

Theorem 4.5 *For any integer w which is a power of 2, define the function $g_w : \mathbf{N} \rightarrow \mathbf{N}$ by $g_w(x) = x \lg w$. Then, the network $\mathcal{M}_w$ bounds maximal difference by g_w.*

Theorem 4.5 implies that the network $\mathcal{M}_w$, viewed as a balancing network, is a $\lg w$-*smoothing* network [4].

5 Contention Analysis

In this Section, we provide an analysis of the worst-case contention of our constructions.

5.1 The Binary Tree

Apparently, the root of the binary tree $\mathcal{B}_w$ will become a hot-spot and a sequential bottleneck that is no better than a centralized load balancing algorithm using a single memory location. As all jobs go through the root, the contention can reach the maximum number n of concurrent jobs, n. Also, apparently, the contention cannot exceed n. Hence, it follows:

Theorem 5.1 $cont(n, \mathcal{B}_w) \;\in \Theta(n)$

However, it is possible to improve performance of the network $\mathcal{B}_w$ by using a technique of combining independent jobs passing through a load balancer: A "prism" mechanism [22] is created in front of the *toggle* and *diff* fields of each load balancer. The prism is distributed over many memory locations, and randomization is used to ensure that each pair of jobs uses a different location. A pair of jobs t_j and t_k "meeting" at a memory location assign themselves to output wires of the load balancer without accessing the fields in its implementation as follows: each of the four possibilities is checked (both on top or bottom output wire, etc), and the one achieving the least value for the variable *diff* is chosen. This mechanism enables to get a highly parallel load balancer with very low contention. This technique is inspired by and extends the technique of *software combining trees* of Yew *et al.* [26] and Goodman *et al.* [13], used also in [22].

5.2 The Bitonic Merger Network

The network $\mathcal{M}_w$ is found to guarantee better performance:

Theorem 5.2 $cont(n, \mathcal{M}_w) \in \Theta(n \lg w / w)$

Sketch of proof: By exploiting the recursive construction of $\mathcal{M}_w$ in order to derive and solve a recurrence relation for $cont(n, \mathcal{M}_w)$. ■

Our contention results for the networks $\mathcal{B}_w$ and $\mathcal{M}_w$, along with Theorems describing the load balancing properties of these networks imply a subtle trade-off between quality of load balancing and memory contention for these networks.

6 Performance

In this Section, we present experimental results for evaluating the performance of load balancing networks.

6.1 A Simulator for a Shared Memory Multiprocessor

Processors' speeds

We take 1 and 10 to be lower and upper bounds, respectively, on the time that a single processor takes to move a task that it sheperds through the network from one load balancer to the next. In general, this time is a function of the processor's speed which may depend on factors such as cache misses, operating system swamps or speed varying instructions. We simulate this behavior by assuming that this time is uniformly distributed in the real-time interval $[1, 10]$.

Whenever a processor passes through a load balancer, our software simulator chooses this time uniformly at random, and the corresponding *task arrival time* is calculated at which the task will arrive at the next load balancer. In this way, the software simulator produces a random sequence of task arrival times.

Load Balancers

We take 1 and 10 to be lower and upper bounds, respectively, on the time that a single load balancer takes to output an input task on any of its output wires once enabled to do so. This time is measured from the time an input task "arrives" at the load balancer and is assumed to be uniformly distributed in the real time interval $[1, 10]$. In this way, the software simulator produces a random sequence of *task departure times.*

Tasks

We consider "short" and "long" tasks, with corresponding completion slightly varying around 1 and 100, respectively. We wish to study how performance depends on the (reduced) relative proportion p of these two classes of tasks, $0 \leq p \leq 1$. We simulate relative proportion by having each processor that sheperds a new task into the network to choose its completion time to be "long" with probability p, and "short" with probability $1 - p$. Having each processor assigned to a unique input wire guarantees that "long" and "short" tasks are distributed on input wires according to their relative proportion. The case where $p = 0$ will be called *biased,* since all tasks are biased to be "bursty," while the case where $p = 1$ will be called *correlated,* since it seems to correspond to the case where there is a hidden correlation between "short" and "long" tasks. All intermediate values of p correspond to mixtures of these two cases.

Methodology

We used a benchmark of 2×10^6 tasks. Given the sequences of task arrival and departure times, the simulator identifies the earliest deadline among all times in these sequences, acts accordingly and continues on. The last deadline is the departure time of the task last to exit the network. We study the following measures of interest:

- *Throughput:* this is the ratio of 2×10^6 by the last deadline, and it represents the number of jobs exiting the load balancing network per unit time;
- *Contention slope:* this is the rate of increase of amortized contention (as formally defined in Section 5) with concurrency;
- *Contention delay:* this is the total time spent by tasks bypassed by other tasks passing through the same balancer; notice that this is different from contention, since it does neither handle nor charge stalls, and also different from total dealy since it does not take into account "moving" costs;

We compared bitonic merger networks of different sizes (all powers of 2 from 4 to 256 including) for concurrency levels ranging from 1 to 600.

Results

Our experimental results (see figures) indicate the following:

1. Relative proportion of “long” and “short” tasks is almost immaterial for all our measures of interest.
2. For sufficiently high concurrency, the larger the network, the longer it takes for it to reach a steady throughput state, and the larger this throughput is.
3. For sufficiently high concurrency, all networks reach a state of constant concurrency slope. The larger the network, the smaller this slope is, which means that the contention in large networks remains constant as concurrency increases, and, therefore, performance is not affected by further concurrency increases.
4. Contention delay is approximately a linear function of concurrency with a coefficient that decreases as the size of the network increases.

Acknowledgements:

Moti Yung suggested using highly biased and highly correlated input tasks (also, mixtures of these two) as test cases for the experimental evaluation of our constructions; our experimental results on such inputs would have never been obtained if it were not for his suggestion. Lampros Ganas provided helpful coding assistance in building up our simulator of a distributed shared memory multiprocessor. We would also like to thank Costas Busch and Christos Nikolau for helpful discussions.

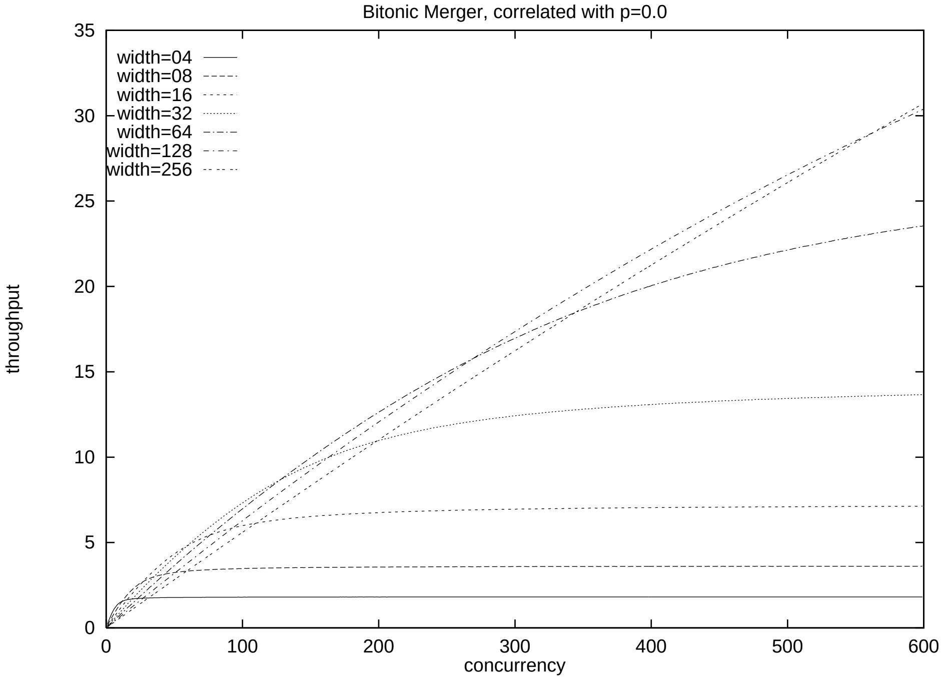

Figure 2: Variance of throughput with concurrency for $p = 0.0$

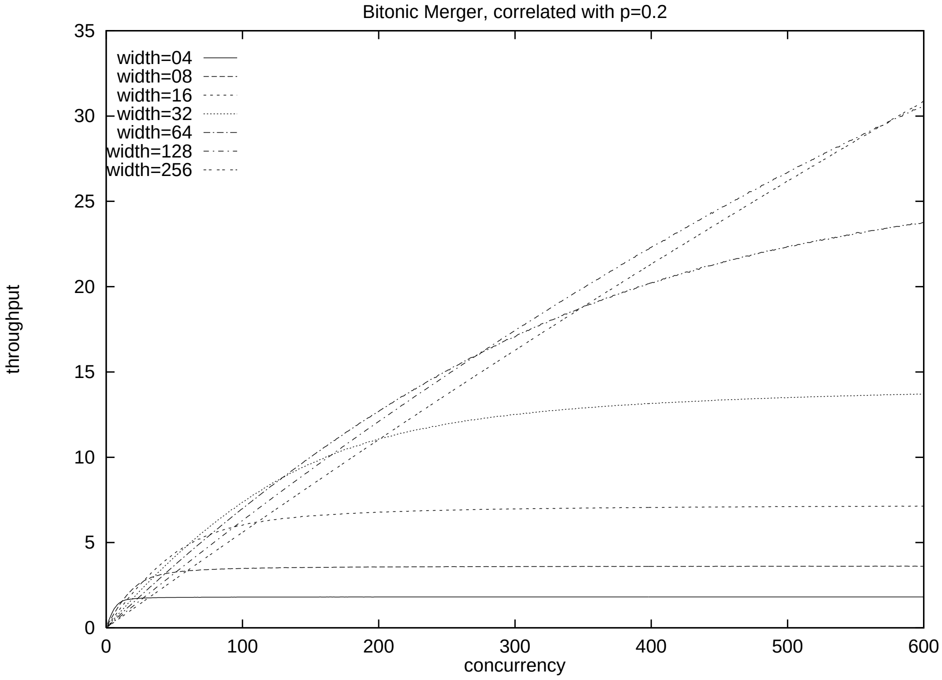

Figure 3: Variance of throughput with concurrency for $p = 0.2$

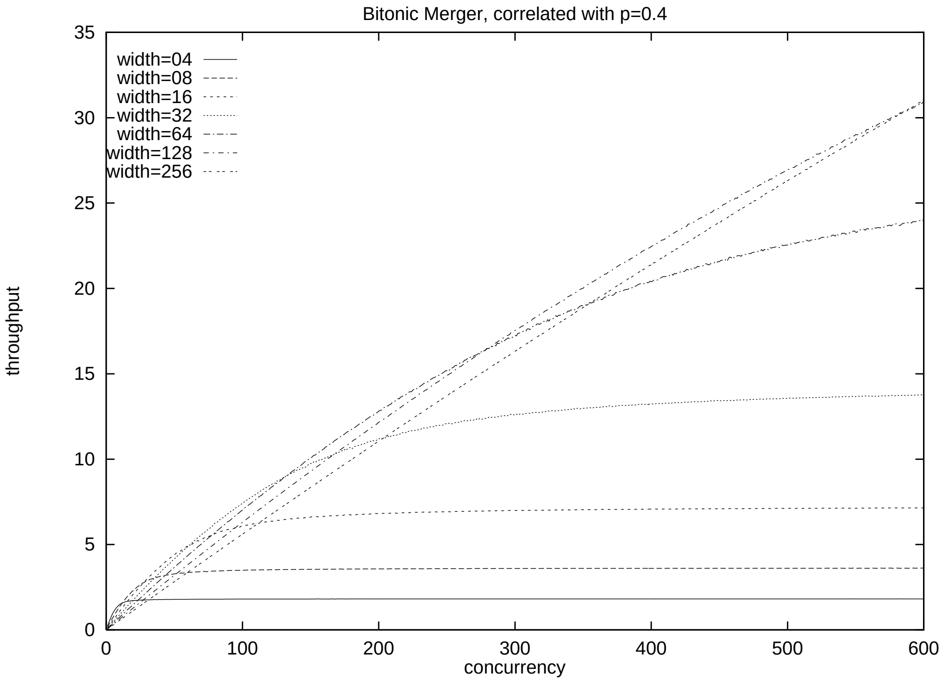

Figure 4: Variance of throughput with concurrency for $p = 0.4$

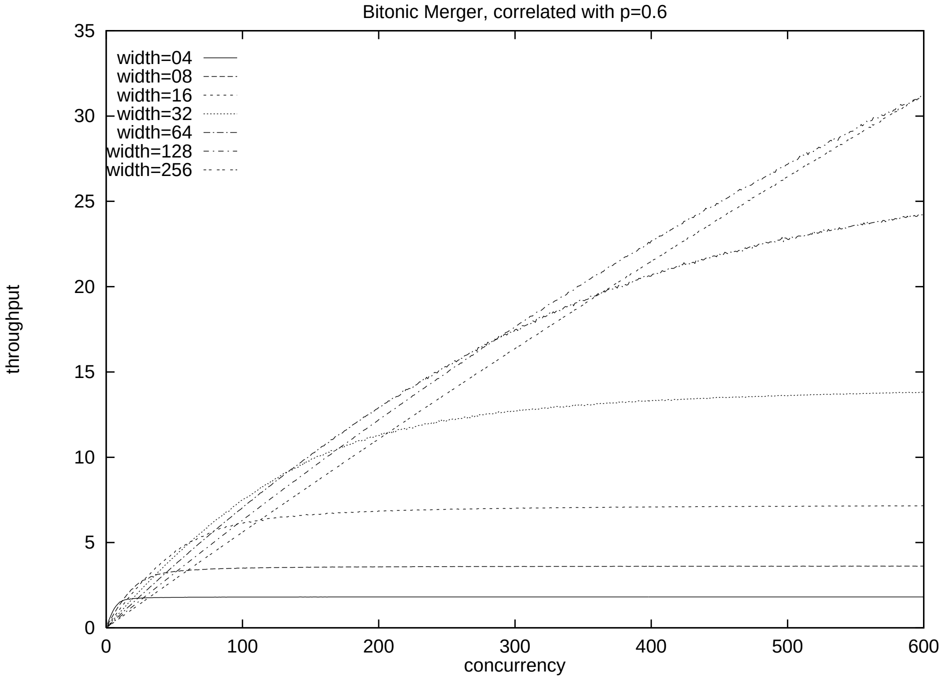

Figure 5: Variance of throughput with concurrency for $p = 0.6$

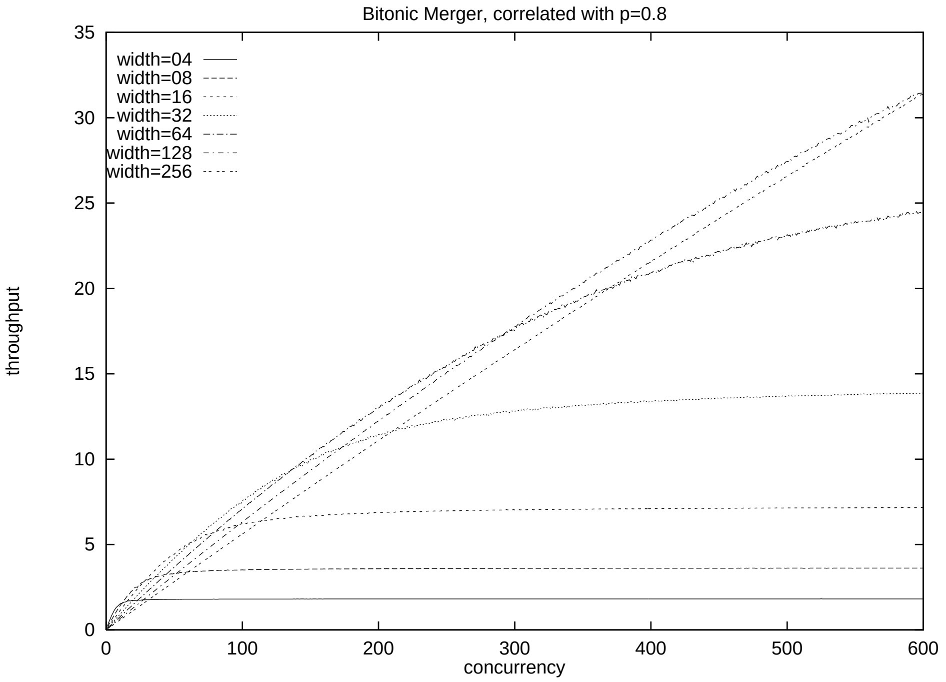

Figure 6: Variance of throughput with concurrency for $p = 0.8$

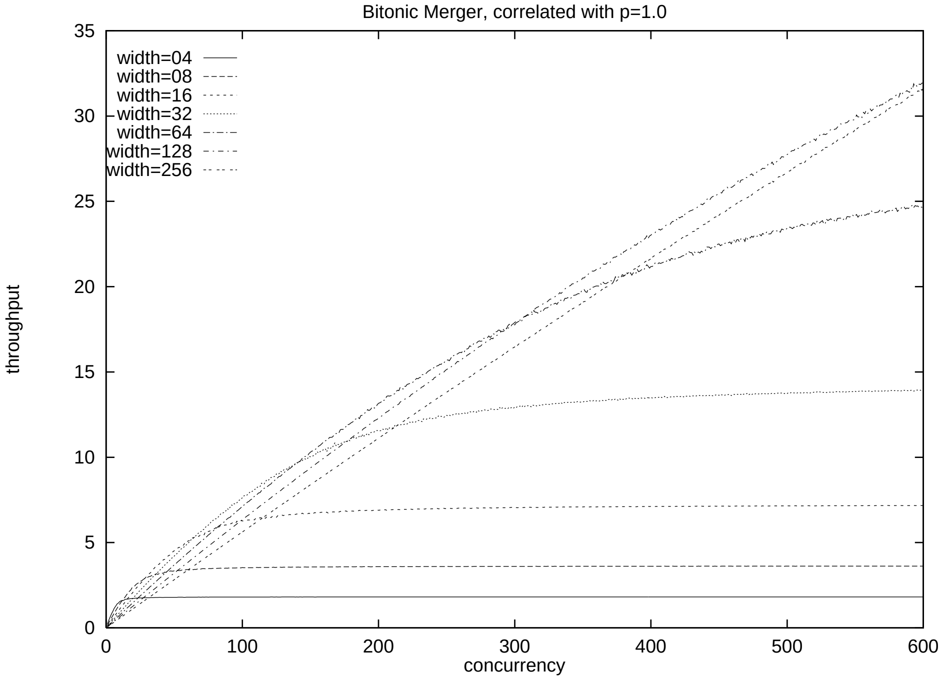

Figure 7: Variance of throughput with concurrency for $p = 1.0$

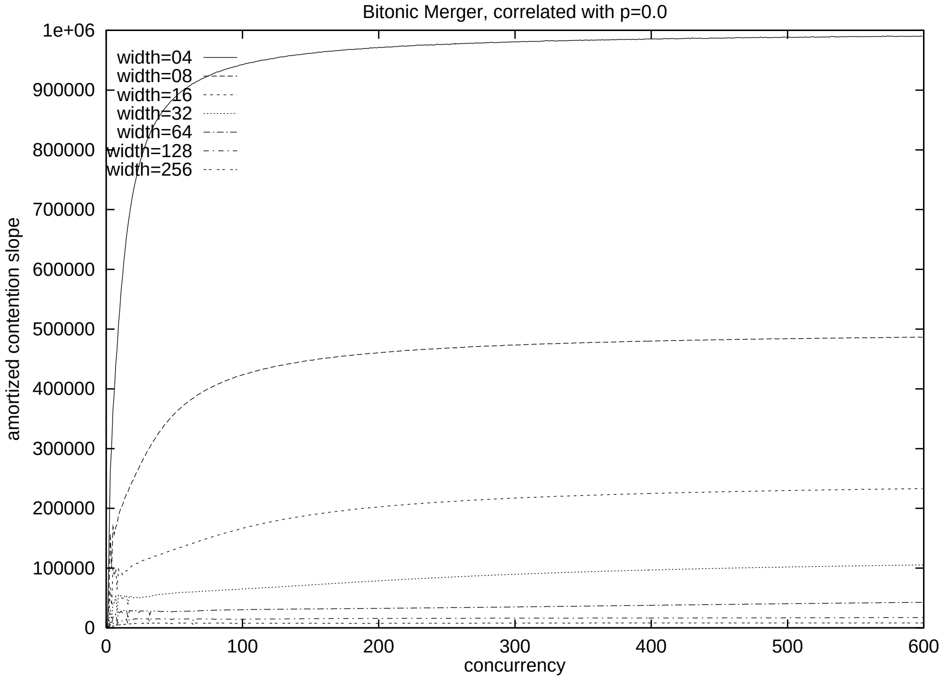

Figure 8: Variance of contention slope with concurrency for $p = 0.0$

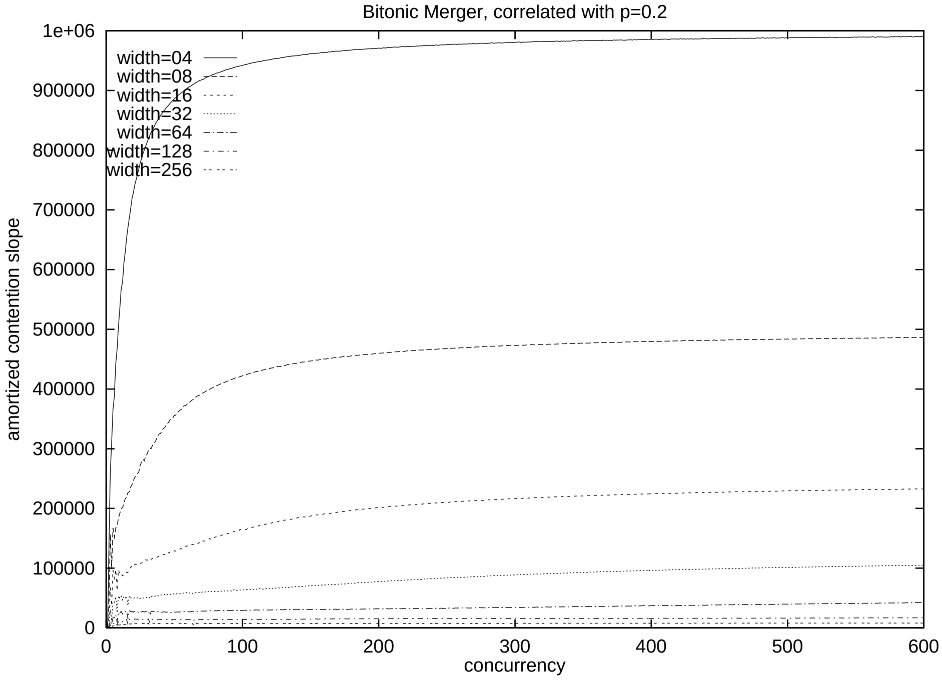

Figure 9: Variance of contention slope with concurrency for $p = 0.2$

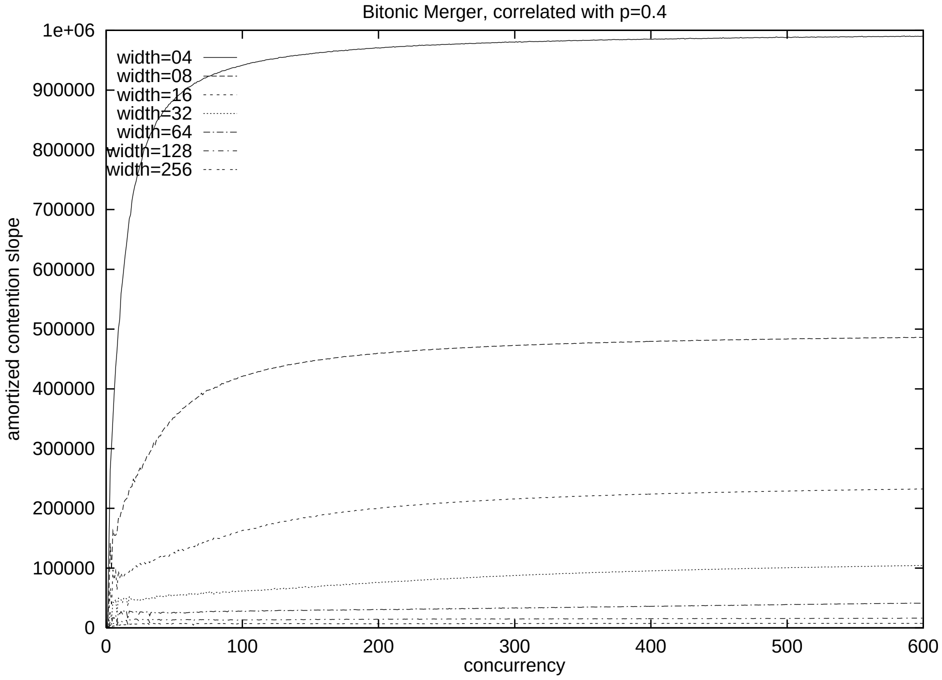

Figure 10: Variance of contention slope with concurrency for $p = 0.4$

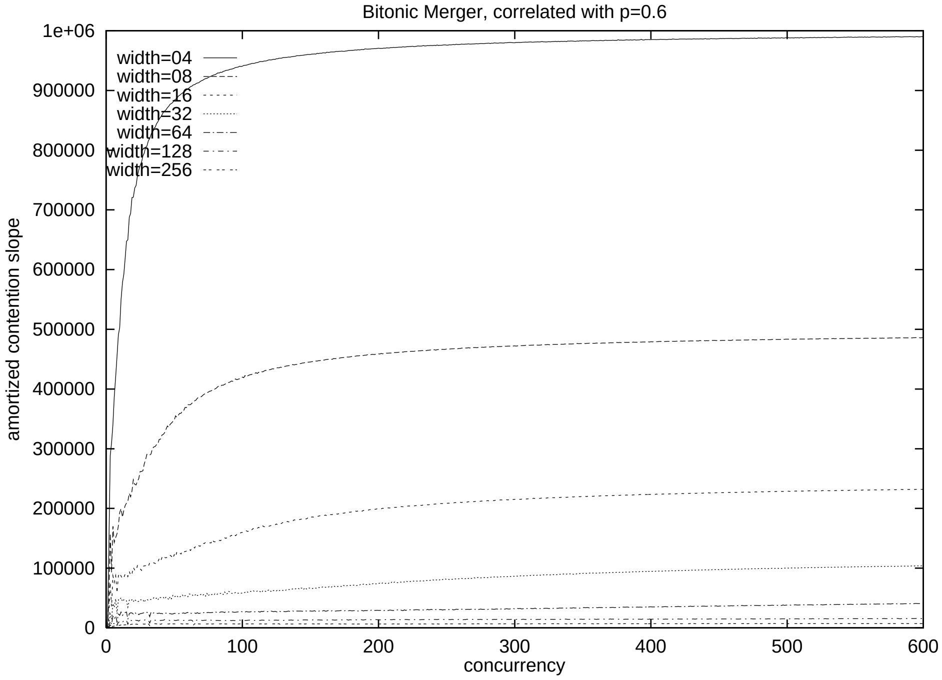

Figure 11: Variance of contention slope with concurrency for $p = 0.6$

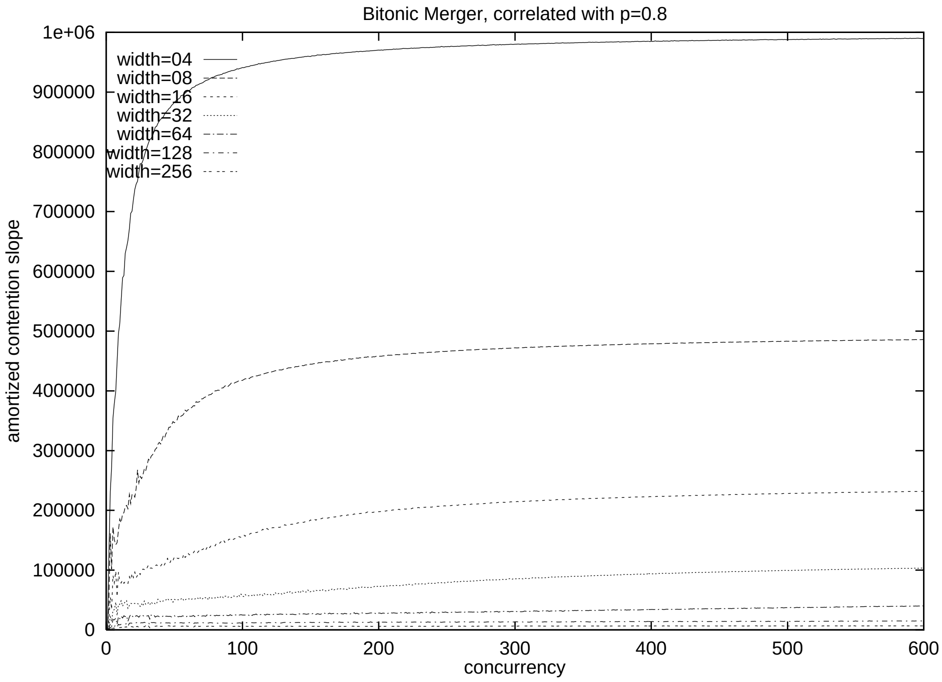

Figure 12: Variance of contention slope with concurrency for $p = 0.8$

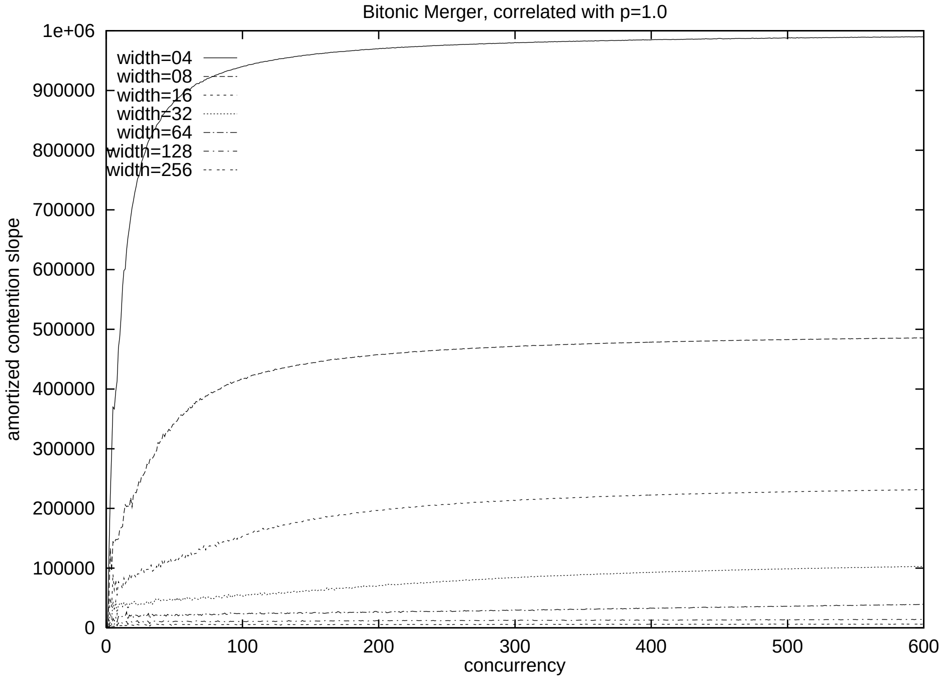

Figure 13: Variance of contention slope with concurrency for $p = 1.0$

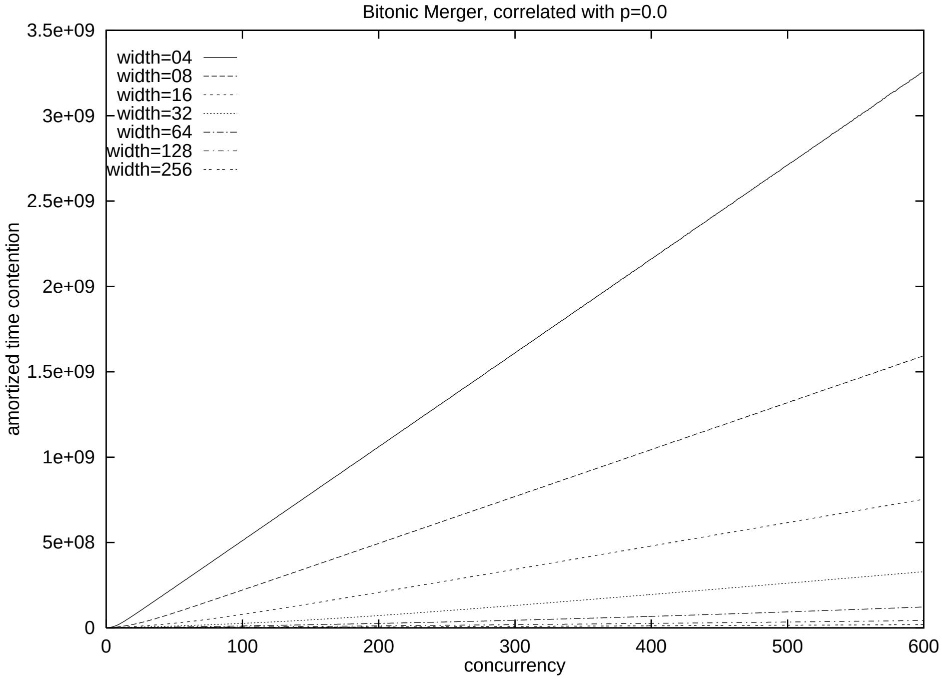

Figure 14: Variance of contention delay with concurrency for $p = 0.0$

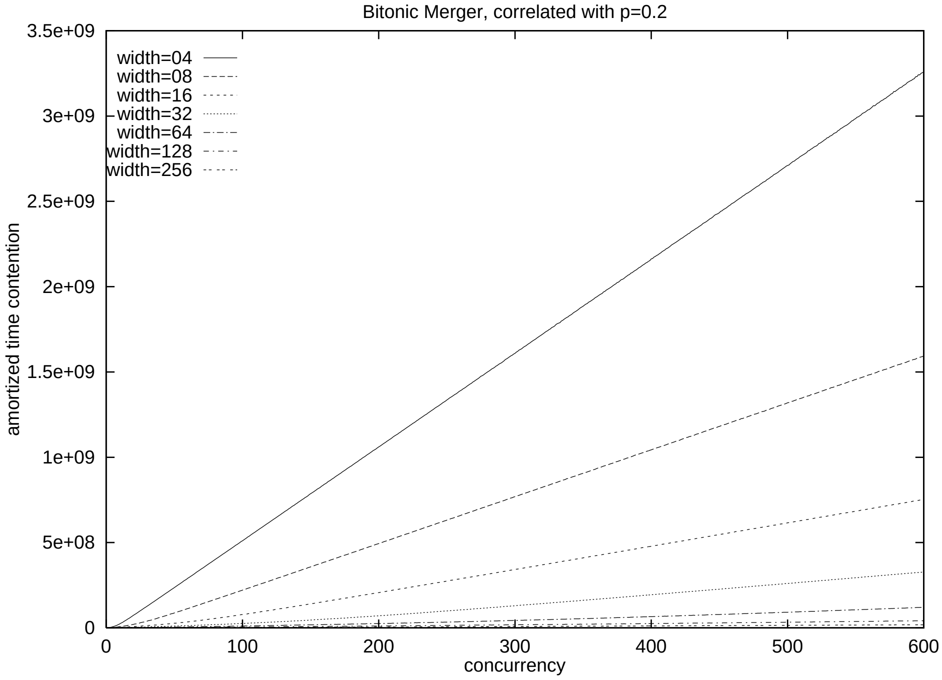

Figure 15: Variance of contention delay with concurrency for $p = 0.2$

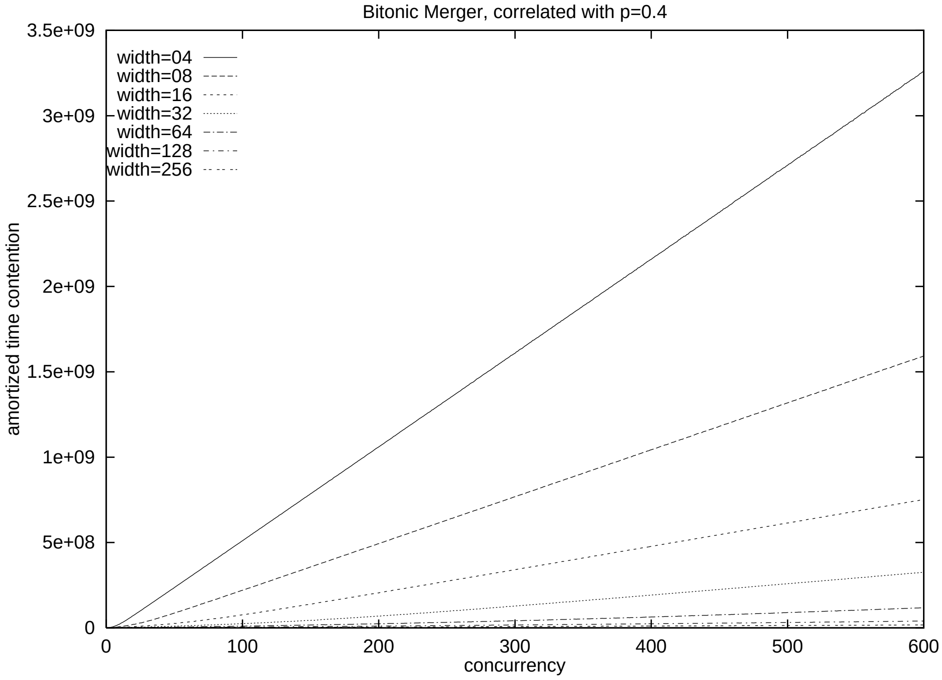

Figure 16: Variance of contention delay with concurrency for $p = 0.4$

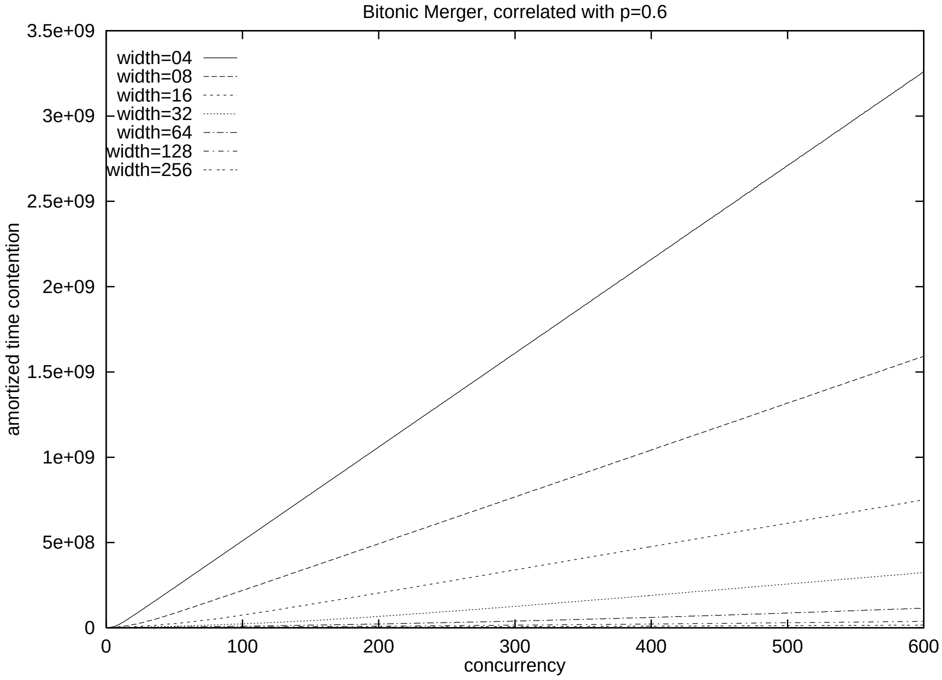

Figure 17: Variance of contention delay with concurrency for $p = 0.6$

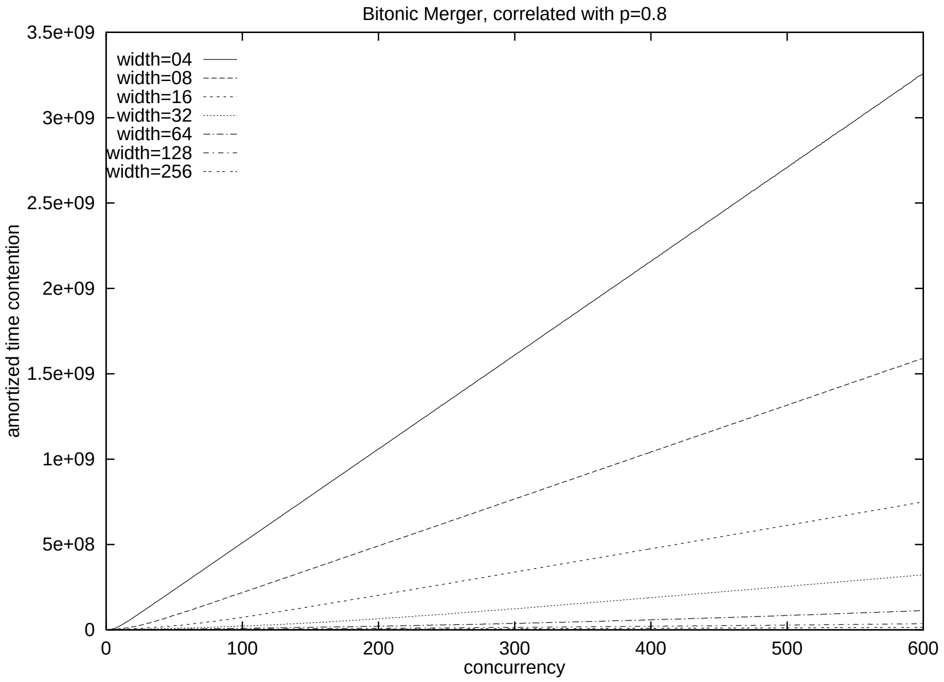

Figure 18: Variance of contention delay with concurrency for $p = 0.8$

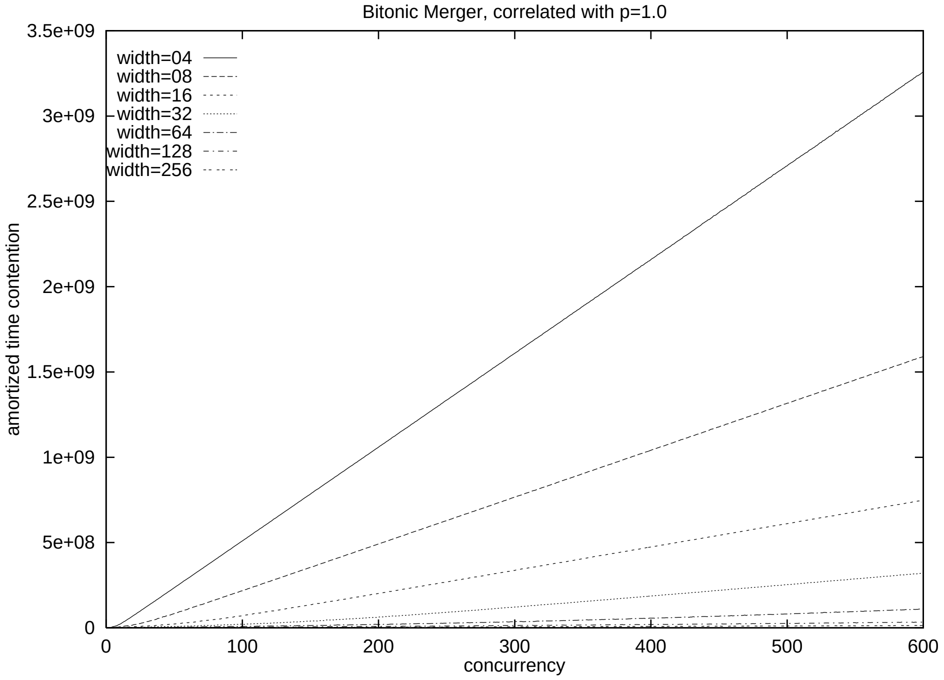

Figure 19: Variance of contention delay with concurrency for $p = 1.0$

References

[1] W. Aiello, R. Venkatesan and M. Yung, "Coins, Weights and Contention in Counting Networks," *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing,* pp. 193–205, August 1994.

[2] R. Alur and G. Taubenfeld, "Results about Fast Mutual Exclusion," *Proceedings of the 13th IEEE Real-Time Systems Symposium,* pp. 12–21, December 1992.

[3] J. H. Anderson and M. Moir, "Using k-Exclusion to Implement Resilient, Scalable Shared Objects," *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing,* pp. 141–150, August 1994.

[4] J. Aspnes, M. Herlihy and N. Shavit, "Counting Networks," *Journal of the ACM,* Vol. 45, No. 5, pp. 1020–1048, September 1994.

Preliminary version: "Counting Networks and Multi-Processor Coordination," *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing,* pp. 348–358, May 1991.

[5] A. Barak and A. Shiloh, "A Distributed Load Balancing Policy for a Multicomputer," *Software Practice and Experience,* Vol. 15, No. 9, pp. 901–913, September 1985.

[6] K. E. Batcher, "Sorting Networks and their Applications," *Proceedings of AFIPS Spring Joint Computer Conference,* pp. 307–314, 1968.

[7] B. Blake and K. Schwan, "Experimental Evaluation of a Real-Time Scheduler for a Multiprocessor System," *IEEE Transactions on Software Engineering,* Vol. 17, No. 1, pp. 34–44, January 1991.

[8] C. Busch and M. Mavronicolas, "A Combinatorial Treatment of Balancing Networks," *13th Annual ACM Symposium on Principles of Distributed Computing,* pp. 206–215, August 1994.

[9] C. Busch, N. Hardavellas and M. Mavronicolas, "Contention in Counting Networks," *13th Annual ACM Symposium on Principles of Distributed Computing,* pp. 404, August 1994.

[10] T. Cormen, C. Leiserson and R. Rivest, *Introduction to Algorithms,* Mc-Graw Hill and MIT Press, 1990.

[11] C. Dwork, M. Herlihy and O. Waarts, "Contention in Shared Memory Algorithms," *Proceedings of the 25th Annual ACM Symposium on Theory of Computing,* pp. 174–183, May 1993.

[12] D. Ferguson, C. Nikolau and Y. Yemini, "Microeconomic Algorithms for Dynamic Load Balancing in Distributed Computer Systems," *Proceedings of the 8th International Conference on Distributed Computing Systems,* June 1988.

[13] J. R. Goodman, M. K. Vernon and P. J.Woest, "Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors," *Proceedings of the 3rd Annual ACM Symposium on Architectural Support and Programming Languages for Operating Systems,* pp. 64–75, April 1989.

[14] M. Herlihy, B.-C. Lim and N. Shavit, "Low Contention Load Balancing on Large-Scale Multiprocessors," *Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures,* pp. 219–227, July 1992.

[15] K. Jeffay, D. F. Stanat and C. U. Martel, "On Non-Preemptive Scheduling of Periodic and Sporadic Tasks," *Proceedings of the 12th IEEE Real-Time Systems Symposium,* pp. 129–139, December 1991.

[16] H. F. Jordan, "Performance Measurements on HEP – a Pipelined MIMD Computer," *Proceedings of the 10th Annual ACM Symposium on Computer Architecture,* pp. 202–212, June 1983.

[17] G. Koren and D. Shasha, "D^{over}: An Optimal On-Line Scheduling Algorithm for Overloaded Real-Time Systems," *Proceedings of the 13th IEEE Real-Time Systems Symposium,* pp. 290–299, December 1992.

[18] L. M. Ni, C. W. Xu and T. B. Gendreau, "A Distributed Drafting Algorithm for Load Balancing," *IEEE Transactions on Software Engineering,* SE-11(10), October 1985.

[19] J. K. Ousterhout, "Scheduling Techniques for Concurrent Systems," *Proceedings of the 3rd IEEE International Conference on Distributed Computing Systems,* pp. 22–30, October 1982.

[20] L. Rudolph, M. Slivkin and E. Upfal, "A Simple Load Balancing Scheme for Task Allocation in Parallel Machines," *Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures,* pp. 237–245, July 1991.

[21] K. Schwan, H. Zhou and A. Gheith, "Multiprocessor Real-Time Threads," *ACM Operating Systems Reviews,* Vol. 25, No. 4, October 1991.

[22] N. Shavit and A. Zemach, "Diffracting Trees," *6th Annual ACM Symposium on Parallel Algorithms and Architectures,* pp. 167–176, July 1994.

[23] Y.-T. Wang and R. J. Morris, "Load Sharing in Distributed Systems," *IEEE Transactions on Computers,* C-34(3), March 1985.

[24] J.-H. Yang and J. Anderson, "Fast Scalable Synchronization with Minimal Hardware Support," *Proceedings of the 12th Annual ACM Symposium on Principles of Distributed Computing,* pp. 171–182, August 1993.

[25] M. Yung, electronic mail message, September 1, 1994.

[26] P. C. Yew, N. F. Tzeng and D. H. Lawrie, "Distributing Hot-Spot Addressing in Large-Scale Multiprocessors," *IEEE Transactions on Computers,* pp. 388–395, April 1987.

[27] J. Zahorjan and C. McCann, "Processor Scheduling in Shared Memory Multiprocessors," *ACM SIGMETRICS '90,* pp. 214–225, May 1990.

[28] H. Zhou, K. Schwan and I. F. Akyildiz, "Performance Effects of Information Sharing in a Distributed Multiprocessor Real-Time Scheduler," *Proceedings of the 13th IEEE Real-Time Systems Symposium,* pp. 46–55, December 1992.