An algorithm for the asynchronous write-all problem based on process collision

An algorithm for the asynchronous *Write-All* problem based on process collision

by

J.F. Groote, W.H. Hesselink, S. Mauw and R. Vermeulen

99/15

ISSN 0926-4515

editors: prof.dr. J.C.M. Baeten
prof.dr. P.A.J. Hilbers

Reports are available at:
http://www.win.tue.nl/win/cs

Computing Science Reports 99/15
Eindhoven, September 1999

An algorithm for the asynchronous *Write-All* problem based on process collision*

Jan Friso Groote[1,2], Wim H. Hesselink[3], Sjouke Mauw[1,2], and Rogier Vermeulen[1]

[1] Eindhoven University of Technology, P.O. Box 513, NL-5600 MB Eindhoven, The Netherlands.
[2] CWI, P.O. Box 94079, NL-1090 GB Amsterdam, The Netherlands.
[3] University of Groningen, P.O. Box 800, NL-9700 AV Groningen The Netherlands.
Email: jfg@cwi.nl, wim@cs.rug.nl, sjouke@win.tue.nl, rogierv@win.tue.nl

Abstract

The problem of using P processes to write a given value to all positions of a shared array of size N is called the *Write-All* problem. We present and analyze an algorithm with work load $\mathcal{O}(N \cdot P^{\log(\frac{x+1}{x})})$, where $x = N^{1/\log(P)}$. Our algorithm is a generalization of the naive two-processor algorithm where the two processes each start at one side of the array and walk towards each other until they collide.

Keywords: write-all problem, wait-free, distributed algorithms, work load, PRAM, dynamic load balancing.

1 Introduction

The *Write-All* problem is defined as follows. Use P processes (or processors) to write a given value to all positions of a shared array of size N. Without loss of generality, we shall assume that the array is an integer array and that 1 is the value to be written to all its positions.

If the processes are reliable and run equally fast, it is easy to come up with straightforward, optimal solutions for this problem. The situation is quite different, however, if processes can be faulty or run at widely varying speeds while at least one process remains active. Kedem et al. [9] have shown that under these circumstances an $N + \Omega(P \log N)$ lowerbound exists on the amount of work processes must carry out when processes can fail. This means that even when all processes run fully in parallel and no process is actually failing, at least $\Omega(\log N)$ time is required to set the array.

The original motivation for the *Write-All* problem comes from [6]. Here it was shown that any program on a P process synchronous PRAM (Parallel Random Access Machine) can be executed on any unreliable PRAM with as overhead the complexity of any algorithm solving the *Write-All* problem. In [8] an overview is given of the algorithms and PRAM simulations that have been developed so-far.

Our motivation is quite different. It comes from the design of wait-free or asynchronous algorithms [3, 4, 5], to obtain fast, reliable programs for general purpose parallel computers with typically a few dozen processes that run under widely varying loads.

A common problem on such machines is to carry out a task, consisting of N independent subtasks, with P processes, as quickly as possible. Such tasks are for instance copying an array, searching an unordered table, and applying a function to all elements of a matrix. We encountered this problem

*Corresponding author: Dr. S. Mauw, Department of Mathematics and Computing Science, Eindhoven University of Technology, P.O. Box 513, NL-5600 MB Eindhoven, The Netherlands, Tel. +31 40 2472908, (secr. 2474124), Fax +31 40 2463992, E-mail sjouke@win.tue.nl

when we had to find a parallel solution to refresh a hashtable by copying all valid elements to a new array [3].

If we abstract from the nature of the subtasks, the problem of executing N independent tasks is adequately characterized by the *Write-All* problem.

In this paper we present a rather straightforward algorithm to solve the *Write-All* problem on an asynchronous PRAM, i.e. a machine on which the processes can be stopped and restarted at will. This means that it is also suitable for all other fault models as mentioned in Kanellakis and Shvartsman, page 13 [8]. Using different terminology we can say that our algorithm is wait-free, which means that one process will be able to finish the whole task, within a predetermined amount of steps, independent of the actions (or failures) of other processes.

For a shared array of size N and P processes, our algorithm has to carry out $\mathcal{O}(N\,P^{\log(\frac{x+1}{x})})$ amount of work where $x = N^{\frac{1}{\log P}}$. The complexity of parallel algorithms is generally characterized by the total amount of steps that all processes must execute, which is called the *work* of the algorithm, instead of the execution time, which under ideal circumstances, can be obtained by dividing the work by the number of available processes. Note that when the number of processes is given, which is generally the case, the amount of work becomes $\mathcal{O}(N)$. Furthermore, it should be noted that the worst case behaviour leading to this upperbound can only be achieved under a rare lock step scenario of the processes. So, we expect average complexity to be much better, which has been confirmed by experiment.

There are a number of existing solutions to the *Write-All* problem (see [8] for an excellent overview). We compare our algorithm to the algorithms X, X', AW, AW^T and Y that are all suitable for asynchronous PRAMs, ignoring the solutions suitable for more restricted fault models. From different perspectives our algorithm improves upon all of these.

Algorithm X is the first asynchronous algorithm for the *Write-All* problem [2]. It is designed for the situation where $P \geq N$ and has work $\mathcal{O}(N\,P^{\log(\frac{3}{2})})$. In [8] a generalisation of X, called X' is presented for the case $P \leq N$ which has the same lowerbound for the amount of work. For $N = P$ the algorithm presented here performs as well as X'. For $P < N$ our algorithm is an improvement over X'.

In [1] two particularly clever algorithms are proposed, called AW and AW^T. Both have good work estimates, but, as stated in [8], are not very practical.

Algorithm AW requires work $\mathcal{O}(P^2 + N \log P)$. When $P \leq \sqrt{N}$ this reduces to $\mathcal{O}(N \log N)$ which is particularly good. However, this bound can only be achieved assuming that a set of permutations of $1 \ldots P$ with a specific property is given, which requires exponential time to calculate. Such a set can be generated at random, but then the result 'only' holds with high probability. In order to overcome this problem algorithm Y has been proposed [7]. Algorithm Y is conjectured to have (non probabilistic) work upperbound $\mathcal{O}(N \log N)$, which is confirmed by experiments, but which is unproven.

Algorithm AW^T needs work $\mathcal{O}(q\,P\,N^{\epsilon})$ where $\epsilon = \log_q \log q^c$ for some constant q that can be freely chosen, and a constant c which according to the proof in [8] can be chosen to be 2. As $\log_q \log q^2$ goes to 0 when q goes to infinity, algorithm AW^T has superior complexity. However, the constant amount of work that must be done in the preprocessing phase (which is independent of N and P) is exponential in q (see [1]). In order to outperform algorithm X' for any N and P, it must be the case that $\epsilon < \log(\frac{3}{2})$. From this it follows that q must be larger than 80. Therefore, to ourperform our algorithm, q must be chosen even larger. In the setting for which we developed our algorithm, we generally have $P < \sqrt{N}$ (and thus $x \geq 2$), so one must choose $\epsilon < \log \frac{5}{4}$ to make algorithm AW^T perform better than our algorithm. This means that q needs to be larger than 10^5. This is the reason why we expect that our algorithm performs much better under practical circumstances.

The present paper has the following structure. In Section 2 we present the algorithm. In Section 3 we prove its correctness and show space and time bounds. Section 4 contains some considerations on using a non-uniform tree as the shared data structure. Finally, Section 5 is reserved for conclusions

and further considerations.

Acknowledgements. We thank Dragan Bošnački, André Engels, Peter Hilbers, Jan Jongejan, Johan Lukkien, and Alex Shvartsman for comments, ideas and references.

2 A collision based algorithm

2.1 Basic case

Although the asynchronous *Write-All* problem in its general setting is far from trivial, the case that there are only two processes ($P = 2$), allows for a very intuitive and optimal solution. This algorithm solves the problem for any value of N in $\mathcal{O}(N)$ steps. One process starts at the left of the array and walks to the right, in the meanwhile setting the values of the array elements encountered to 1. The other process does the same from right to left. If the two processes collide, the whole array is processed and the processes can stop. In the worst case, one element of the array is processed twice. We will call this algorithm the *Basic Collision* algorithm.

In [2] an extension of this algorithm is described, which works with three processes. It is called algorithm T. Two processes have the same behaviour as described above, but the third process behaves differently. It starts in the middle of the array and fills the array alternately to the left and to the right. If the first two processes collide, it means that the whole array is processed. If e.g. the first and the third process collide, it means that the left part of the array is processed. Therefore they move to the segment of the array that is not processed yet. The first process starts at the left of this segment, the third process starts again in the middle of this segment, and the second process is still busy filling the segment from the right. This procedure repeats until the array is completely processed. This algorithm is also optimal and has a work load $\mathcal{O}(N)$. Algorithm T does not appear to be generizable to larger numbers of processes.

2.2 Generalized case

Our algorithm generalizes the *Basic Collision* algorithm in a different way. We will call it the *Generalized Collision* algorithm. It is best explained by looking at a simple example with four processes ($P = 4$). We choose $N = 25$ in our example.

The processes operate in pairs. Every pair of processes executes the *Basic Collision* algorithm on successive segments of the array. Each segment has length 5, so there are 5 segments. The four processes start at the locations indicated in Figure 1. The arrows indicate the direction in which each process traverses the segment.

Figure 1: Initial configuration

Every time that a segment of the array has been processed by a pair, operation continues at the next segment. The first process of a pair to finish a segment can directly continue with the next segment, without having to wait for the other process. In this way, the pairs walk towards each other through the array in steps of length 5 until they collide. A typical path of the four processes in our example is shown in Figure 2. This figure shows just one possible path, in which all processes roughly operate at the same speed. The algorithm, however, is completely robust with respect to process delays, failures and restarts. This is because every process potentially visits all array elements. As long as one process survives, the whole array will be processed.

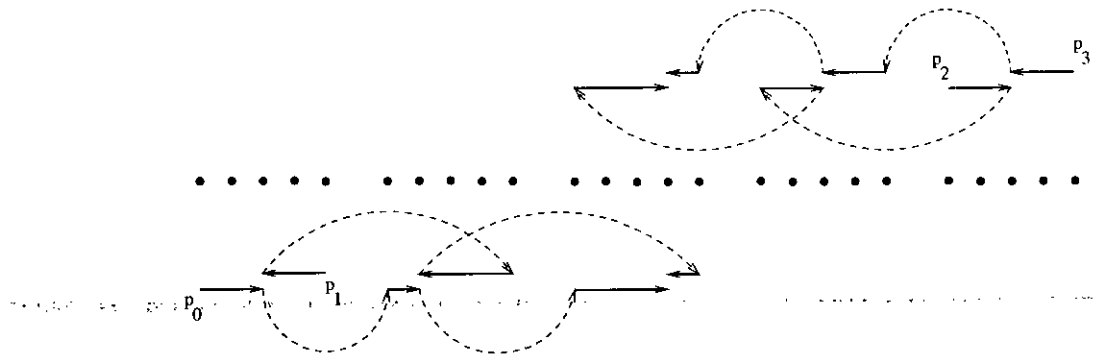

Figure 2: Possible paths of the processes

From a higher point of view, the four processes also execute the *Basic Collision* algorithm where the grainsize of the work is 5. To see this, we have to consider every pair as a single aggregated process and every segment of length 5 as a single aggregated array element. A collision now takes place at a complete segment, rather than at a single array element. This explains why the middle segment in Figure 2 is processed twice.

It is now clear how to generalize this example if we double the numbers of processes and assume 125 array elements. We simply add one level to the hierarchy and have clusters of four processes operate on segments of length 25, until the clusters collide.

This implies that our algorithm works for any number of processes which is a power of two, so $P = 2^k$ for some $k \geq 1$. Furthermore, we have that the length of the array is the length of a basic segment to the same power, so $N = x^k$ for some $x \geq 2$. In the above example we have chosen $k = 2$ and $x = 5$.

In Figure 3 the generalization of the *Basic Collision* algorithm is illustrated in a cube which has to be filled with 1's by 8 processes. The picture shows pairs of processes, clusters of 2 processes, and clusters of 4 processes racing each other. In this example $k = 3$ (the dimension of the cube) so there are 8 processes, and the length of an edge of the cube is x, so that there are x^3 cells to be filled. This is the biggest example that we can easily visualize in this way. An example with 16 processes would require a 4-dimensional figure.

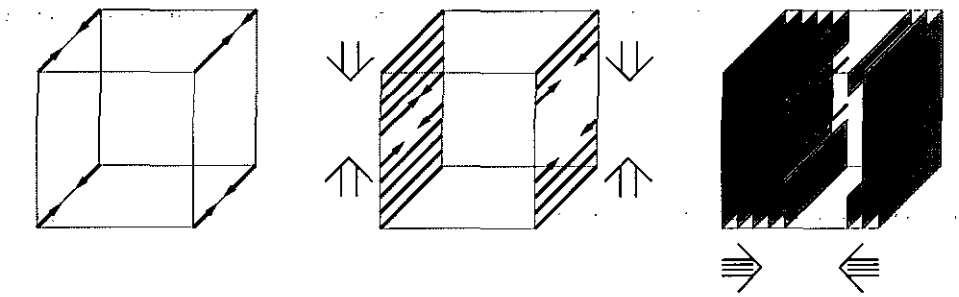

Figure 3: Generalization of the collision principle illustrated in a cube.

2.3 Datastructures

Additional datastructures are needed in order to enable the processes to decide which array element should be processed next. First of all, every process has a process identifier (*pid*) consisting of a bitstring of length k. The set of all process identifiers is called *PID*. We use the functions *head* and *tail* to return the first element of a bitstring and the bitstring with the first element deleted. The bitstrings will be used to direct the processes to different parts of the array. There is a nice relation between the pids of the processes and the initial position of the processes in the cube from Figure 3. If we consider the general Boolean k-dimensional hypercube, the pids correspond to the processes's initial co-ordinates.

Next, we assume that the processes share a tree of depth k. According to the above explanation the tree should have a uniform fan out x. This would mean that there are exactly x^k leaves, which correspond with the elements of the array. However, we will formulate our algorithm in such a way that it also works for trees with a non-uniform fan-out, for reasons explained in Section 4.

Every leaf l has an attribute $l.value : int$ that must be set to 1. The relation between the tree and the cube that we used to illustrate the generalization of the collision principle, is straightforward. Each level in the tree corresponds with a dimension, and a cell (c_0, c_1, c_2) of the cube corresponds with the leaf that we arrive at if we travel down the tree first taking the c_0-th branch, then the c_1-th branch, and finally the c_2-th branch.

The internal nodes of the tree maintain information on how far the corresponding subtree has been processed already. Every internal node n has the following three attributes.

- $n.fan : int$
  This constant denotes the number of children of the node.
- $n.nl : 0..n.fan := 0$
  This variable denotes the number of child nodes that have already been processed, from left to right.
- $n.nr : 0..n.fan := 0$
  This variable denotes the number of child nodes that have already been processed, from right to left.

Note that the subtree of node n has been processed completely if $n.nl + n.nr \geq n.fan$.

The root of the tree is denoted by *root* and the predicate *is_leaf* determines if a node is a leaf. Similar to algorithm T in [2], we make use of an atomic *tcompare-and-swap*-like instruction (see e.g. [4]). In the algorithm below this is denoted by placing angular brackets around the statement ('⟨' and '⟩'). We mention that for correctness of the algorithm, atomicity is not really needed, but for our work load calculations it is.

2.4 The algorithm

All processes operate in parallel and perform the same recursive procedure *traverse* with as the first argument the process identifier and the second argument the root of the tree. The recursive calls have as arguments smaller bit strings and other nodes of the tree. We use notation from [8] to express this.

forall *pid* **in** *PID* **parbegin**
  traverse(*pid*,*root*)
parend

Procedure *traverse* is defined below.

```
procedure traverse(bs,node)
var i: 0..node.fan;
begin
    if is_leaf(node) then
      node.value := 1
    else
      if head(bs) = 0 then
        i := node.nl;
        while i + node.nr < node.fan do
          traverse(tail(bs),child(node,i));
          ⟨ if node.nl = i then node.nl := i + 1 fi ⟩;
          i := node.nl
        od
      else
        i := node.nr;
        while node.nl + i < node.fan do
          traverse(tail(bs),child(node,node.fan − 1 − i));
          ⟨ if node.nr = i then node.nr := i + 1 fi ⟩;
          i := node.nr
        od
      fi
    fi
end
```

In the base case where the node is a leaf, the procedure writes the intended value in the array. Otherwise, the procedure treats the children of the node in a repetition from left to right or from right to left. The choice between starting left or right is irrelevant for correctness. For the sake of the work load, we let the choice depend on the head of the first argument *bs*, which is a suffix of the process's *pid*. The recursive calls have the tail of the bit string *bs* as first argument, so that the processes start their actions at different points in the array. The updates of *node.nl* and *node.nr* are conditional atomic updates, for reason of efficiency only. Otherwise a delayed process might have to treat a large part of the array again. Private variable i is introduced to allow modification of the shared variables *node.nl* and *node.nr* by other processes.

This code expects the process identifiers *pid* to have length equal to (or greater than) the depth of the tree. One may prefer to use process identifiers of type integer with the conventions that $head(bs) = bs \bmod 2$ and $tail(bs) = bs \operatorname{div} 2$.

3 Analysis of the algorithm

3.1 Correctness

The proof of correctness of the distributed algorithm consists of two steps. First we prove partial correctness (i.e. if one of the processes successfully finishes, the whole tree has been processed) and next we prove termination (at least one process finishes successfully). If all leaves of a (sub)tree have been set to 1, we say that the (sub)tree has been processed.

Partial correctness follows from the next lemma.

Lemma 1

1. *For every internal node* n *of the shared tree, invariably that* n.nl *subtrees of node* n *from left to right have been processed. Likewise* n.nr *subtrees have been processed from right to left.*
2. *If a call* traverse(σ,n) *(for some bitstring σ and some node n finishes successfully, the subtree rooted in node* n *has been processed.*

Proof We prove the two parts with simultaneous induction on the depth of node n. The base case, where node n is a leaf is trivial. For the inductive case, we suppose that node n is an internal node.

1. The value of shared variable $n.nl$ is only incremented from i to $i+1$ if procedure *traverse* has finished on the i^{th} subtree of node n. By induction we then have that this subtree has been processed, which certifies this invariant. The variable $n.nr$ is treated similarly.
2. If a call *traverse(σ,n)* finishes, one of the guards $i + n.nr < n.fan$ and $n.nl + i < n.fan$ must be false. Notice that in the first case we have $i \leq n.nl$ and in the second case $i \leq n.nr$. This is due to the fact that the values of $n.nr$ and $n.nl$ are non-decreasing. Therefore, if the call *traverse(σ,n)* finishes we have $n.nl + n.nr \geq n.fan$. Using the induction hypothesis, we can conclude that all subtrees of node n are processed, so the tree rooted in n is processed.

Next, we will prove termination of the algorithm. For this we formulate the following termination function:

$$\sum_{n \in internalnodes} n.nl + n.nr$$

The fact that this is a proper termination function follows from the following observations.

First, the function is bound, since for every internal node n the values of $n.nl$ and $n.nr$ are bounded by the constant n.fan. Second, recall that the fault model implies that all but one process may fail. Since there are no blocking statements, the surviving process will continually invoke calls to procedure *traverse*. After every call of this procedure (to, say, node n), the value of $n.nl + n.nr$ is strictly larger than before this call. Namely, it is incremented with 1 by the calling process, or it is incremented with at least 1 by one or more other processes.

In conclusion, we have that at least one call of *traverse(pid,root)* finishes successfully (termination) and that this implies that the complete tree rooted in *root* has been processed.

3.2 Space usage

The shared data structure consists of the given array of N bits, together with the data at the $N-1$ internal nodes of the tree (see e.g. [8] for a description of how to represent a tree in a heap without overhead). Every internal node n holds two shared variables of size $\log n.fan$. So the shared memory has size of $O(N \log N)$.

Every process needs a private data structure with space for k stack frames, since the recursion depth is k. Each stack frame holds a local variable of size $\log node.fan$ and two parameters of size k (or at least $\log k$) and $\log N$.

So the processes have only moderate space requirements.

3.3 Work load

As was mentioned before, the work load of a parallel algorithm is the worst case total amount of work performed by the processes involved. With 'total amount of work' one generally means the number of instructions executed by all processes. We measure the work load by counting the total number

of calls of procedure *traverse* in a worst case scenario. The program text clearly shows that only a constant number of instructions is executed in each call of *traverse*, so the total number of procedure calls is an appropriate estimate here.

In the calculations below we will assume that the number of processes is 2^k and that the length of the array is x^k for some $k \geq 1$ and $x \geq 2$. This allows for the construction of a tree with a uniform fan-out. We will briefly consider the case of a tree with non-uniform fan-out in Section 4.

Because of the recursive structure of the input of the algorithm, the shared tree with fan out x, we define the work load inductively, i.e. express the work load associated with a tree of height i in terms of the work load associated with its subtrees of height $i-1$. Note, that the number of processes, which of course plays an important role in determining the work load, is fixed when we know the height of the tree, namely 2^i (given a tree of height i). In our first inductive definition of work load, however, we will decouple the number of processes and the height of the input tree, because, as we will see, subtrees of the input tree can be overloaded with processes (and *will* be overloaded in a worst case scenario).

We introduce numbers $W_{i,j}$ to estimate the work load for 2^j processes on a tree of height $i \leq j$. These numbers are defined recursively by

$$W_{0,j} = 2^j \tag{1}$$

$$W_{i+1,j} = 2^j + (x-1)W_{i,j-1} + W_{i,j} \tag{2}$$

Equation (1) is justified by the observation that, when 2^j processes start to work on a tree of height 0, a single leaf, they will all call procedure *traverse* once to set the array item associated with the leaf to 1, resulting in a work load of 2^j.

We arrive at equation (2) in the following way (see also Figure 4). When 2^j processes start working on a tree of height $i+1$, they will first all call procedure *traverse* on the root of the tree. This accounts for the summand 2^j in the definition of $W_{i+1,j}$. Next, the processes split up in two groups of 2^{j-1} processes, according to their *pids*. One group will process the subtrees from left to right and the other group vice versa. The collision-principle assures us that these two groups can interfere in only *one* of the x subtrees, the one where they collide (the grey sub-tree in Figure 4). The work load associated with the subtrees can therefore be split in $x-1$ times the work load of 2^{j-1} processes on a tree of height i (the summand $(x-1)W_{i,j-1}$), and the work load of 2^j processes on a single tree of height i (the summand $W_{i,j}$).

We will now transform the recurrence relation W into a simpler one, taking advantage of the fact that we are primarily interested in $W_{k,k}$ (where $N = x^k$ and $P = 2^k$). We first need to prove the following conjecture which states that doubling the number of processes doubles the work load:

$$2 \cdot W_{i,j-1} = W_{i,j} \qquad (\text{for } j > i) \tag{3}$$

We prove this by induction on i. For $i = 0$ we have $2 \cdot W_{0,j-1} = 2 \cdot 2^{j-1} = 2^j = W_{0,j}$. Assuming that the conjecture holds for i, we derive for $i+1$: $2 \cdot W_{i+1,j-1} = 2 \cdot (2^{j-1} + (x-1) \cdot W_{i,j-2} + W_{i,j-1}) = 2^j + (x-1) \cdot 2 \cdot W_{i,j-2} + 2 \cdot W_{i,j-1} = 2^j + (x-1)W_{i,j-1} + W_{i,j} = W_{i+1,j}$.

Property (3) can also be explained in terms of *pids*. When the number of processes is doubled, they will have to share *pids* (because $j > i$). Each process will have a doppelganger that follows the exact same route through the tree. This imitative behaviour explains the doubling of the work load.

Because of property (3) we can rewrite the second equation of the definition of W as follows:

$$\begin{aligned} W_{i+1,j} &= 2^j + (x-1)W_{i,j-1} + 2 \cdot W_{i,j-1} \\ &= 2^j + (x+1)W_{i,j-1} \end{aligned}$$

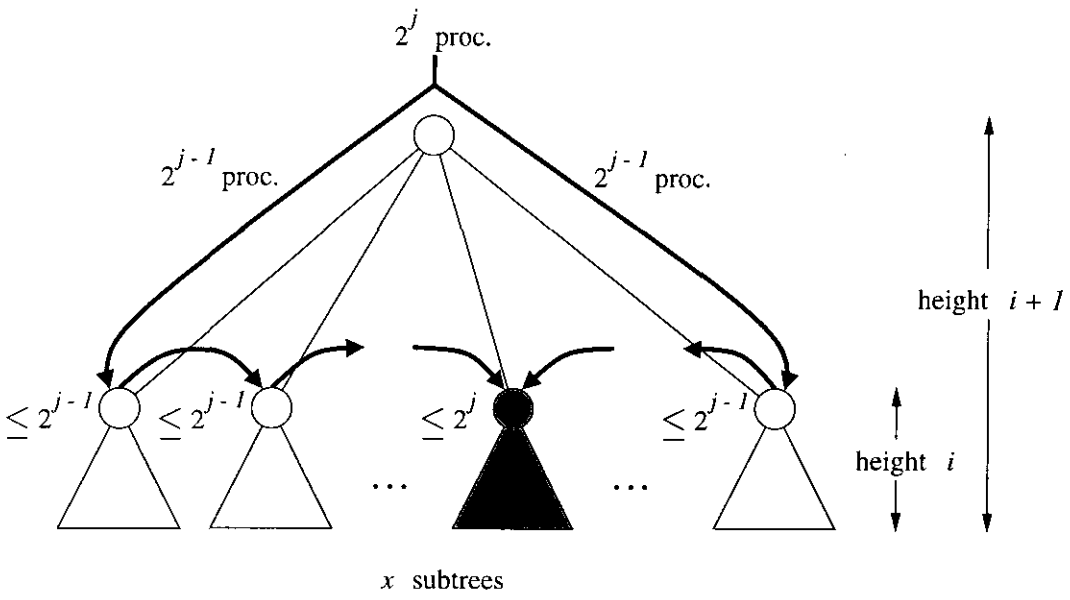

Figure 4: Worst case distribution of processes over subtrees.

This equation exhibits a nice correlation between arguments i and j that equation (2) did not. This observation and the fact that we are interested in $W_{k,k}$ lead to the introduction of w_i, which, for $i \geq 0$, denotes the work load of 2^i processes on a tree of height i. Therefore we define $w_k = W_{k,k}$ and it follows that w_k satisfies the recurrence relations (4) and (5).

$$w_0 = 1 \tag{4}$$

$$w_{i+1} = 2^{i+1} + (x+1)w_i \tag{5}$$

We now calculate an appropriate estimate for w_k. Although these calculations are straightforward we provide them in full.

$$\begin{aligned}
& w_k \\
&= \{ \text{ solve the recurrence relation } \} \\
& \sum_{i=0}^{k} 2^{k-i}(x+1)^i \\
&= \{ \text{ simple math } \} \\
& \frac{(x+1)^{k+1} - 2^{k+1}}{x-1} \\
&= \{ \text{ even more simple math } \} \\
& \frac{x+1}{x-1}(x+1)^k - \frac{2^{k+1}}{x-1} \\
&\leq \{ \ x \geq 2, \text{ hence } (x+1)/(x-1) \leq 3 \text{ and } 2^{k+1}/(x-1) > 0 \ \} \\
& 3 \cdot (x+1)^k
\end{aligned}$$

Hence, we have $w_k = \mathcal{O}((x+1)^k)$. We would like to express the work load in terms of N and P, so we do some more calculations on $(x+1)^k$, using the equalities $N = x^k$ and $P = 2^k$:

$$\begin{aligned}(x+1)^k &= x^k \frac{(x+1)^k}{x^k} \\ &= x^k \left(\frac{x+1}{x}\right)^k \\ &= N \cdot \left(\frac{x+1}{x}\right)^{\log(P)} \\ &= N \cdot P^{\log(\frac{x+1}{x})}\end{aligned}$$

We have now proven that the work load of our algorithm is $\mathcal{O}(N \cdot P^{\log(\frac{x+1}{x})})$, where $x = N^{1/\log(P)}$.

4 Non-uniform fan-out

Although we were able to prove correctness of the algorithm for trees with non-uniform fan-out, we did assume uniform fan-out for our work load calculations. This assumption proved very useful for obtaining a result which can easily be compared with work load calculations for other algorithms.

Nonetheless, we claim that this assumption is not critical for the performance of our algorithm. Calculations and experimentation support this claim. Examples show that, strictly speaking, an optimal work load is almost never achieved with a uniform fan-out. In almost all cases the work load can be slightly improved by rebalancing the tree, while still keeping it quasi-uniform. By quasi-uniform we mean that the nodes at the same level of a tree have equal fan-out.

In the case of quasi-uniform fan-out the work load can be given as a closed expression that contains sum and product quantifiers. One cannot expect otherwise, since the fan-out of the levels is now given by a vector (the sequence of fan-outs of various levels). If one now treats the fan-outs as real numbers, one can obtain a recursive formula for the optimal fan-outs. This can be used as the starting point for a search for an optimal integer solution under the side condition that the quasi-uniform tree has at least N leaves.

These calculations show, e.g., that in the case that $N = 12,000$ optimal work load is obtained if the fan-out for each level is 5, 5, 5, 6, 16, from the root level to the level above the leaves. In the general case we see that an optimal work load is obtained if the fan-out for all levels are approximately the same, except for the fan-out at the level above the leaves, which should be three times larger.

We expect that in practice rebalancing the tree will yield at most a constant speed up in performance.

5 Observations and conclusions

We have presented an algorithm for the asynchronous *Write-All* problem. This algorithm is suitable for a multiprocess environment, as due to the lack of explicit synchronization, it has good performance. In particular this is the case when the task of setting a variable to one is replaced by a more time consuming operation. Moreover, the algorithm is fault tolerant in the sense that it works correctly even if individual processes can fail or can stop and resume arbitrarily, assuming that not all processes die. Finally, our algorithm performs a kind of dynamic load balancing. Every process checks in a specific order all the tasks that must be executed and if it finds one that has not been performed, it carries it out. Due to the data structures involved, this can be done with minimal duplication of work. This guarantees a distribution of tasks over processes, where no process will idle when work can be done.

Our algorithm improves upon existing asynchronous algorithms in several ways. In comparison with most published algorithms it has a better order of performance. This does not hold for algorithms AW and AW^T, which are based on a rather different algorithmic concept than our algorithm. Algorithm AW 'only' improves upon our algorithm with high probability, although we expect that in practice this algorithm has a good performance. From a theoretical perspective AW^T performs better than our algorithm, but due to a high initial constant amount of work AW^T is not suitable for any practical purposes.

To ascertain these findings, we have implemented our algorithm and ran it for different numbers of processes, where we compared the number of process steps with the worst case estimate of the amount of work that needs to be done. Without going into detail, as we believe that it is very hard to draw universal conclusions from experiments, we found that the overhead always remained far below our worst case estimate.

Finally, we make some observations concerning the restrictions on the values for N and P. In the case that we use a tree with uniform fan-out as the shared data structure, an array of size $N = x^k$ can be accommodated. However, such uniform fan-out is not needed for obtaining an optimal work load. By adjusting the fan-out of the nodes in the tree, it is possible to accommodate an array with arbitrary size N. Furthermore, since processes need not execute, we can take $P \leq 2^k$, provided all process identifiers differ and have a length at least equal to the depth of the tree. The work load remains essentially the same.

References

[1] R.J. Anderson and H. Woll. Algorithms for the certified write-all problem. *Siam Journal of Computing*, 26(5):1277-1283, 1997.

[2] J.F. Buss, P.C. Kanellakis, P.L. Ragde and A.A. Shvartsman. Parallel Algorithms with Processor Failures and Delays. *Journal of Algorithms*, 20:45-86, 1996.

[3] J.F. Groote and W.H. Hesselink. Synchronization-free parallel accessible hash-tables. In preparation, 1999

[4] Herlihy, M.P.: Wait–free synchronization. ACM Trans. on Program. Languages and Systems **13** (1991) 124–149.

[5] W.H. Hesselink and J.F. Groote. Waitfree Distributed Memory Management by Create, and Read Until Deletion (CRUD). Technical report SEN-R9811, CWI, Amsterdam, 1998.

[6] P.C. Kanellakis and A.A. Shvartsman. Efficient parallel algorithms can be made robust. *Distributed Computing*, 5(4):201–217, 1992. A preliminary version appeared in *Proceedings of the 8th ACM PODC*, pages 211–222, 1989.

[7] P.C. Kanellakis and A.A. Shvartsman. Fault-tolerance and efficiency in massively parallel algorithms. In G.M. Koob and C.G. Lau, editors,*Foundations of Dependable Computing – Paradigms for Dependable Applications*, pages 125–154, Kluwer Academic, 1994.

[8] P.C. Kanellakis and A.A. Shvartsman. *Fault-tolerant parallel computation.* Kluwer Academic Publishers, 1997.

[9] Z.M. Kedem, K.V. Palem, A. Raghunathan, and P. Spirakis. Combining tentative and definite executions for dependable parallel computing in *Proceedings of the 23d ACM Symposium on Theory of Computing*, 1991.

Computing Science Reports

Department of Mathematics and Computing Science
Eindhoven University of Technology

In this series appeared:

96/01	M. Voorhoeve and T. Basten	Process Algebra with Autonomous Actions, p. 12.
96/02	P. de Bra and A. Aerts	Multi-User Publishing in the Web: DreSS, A Document Repository Service Station, p. 12
96/03	W.M.P. van der Aalst	Parallel Computation of Reachable Dead States in a Free-choice Petri Net, p. 26.
96/05	T. Basten and W.M.P. v.d. Aalst	A Process-Algebraic Approach to Life-Cycle Inheritance Inheritance = Encapsulation + Abstraction, p. 15.
96/06	W.M.P. van der Aalst and T. Basten	Life-Cycle Inheritance A Petri-Net-Based Approach, p. 18.
96/07	M. Voorhoeve	Structural Petri Net Equivalence, p. 16.
96/08	A.T.M. Aerts, P.M.E. De Bra, J.T. de Munk	OODB Support for WWW Applications: Disclosing the internal structure of Hyperdocuments, p. 14.
96/09	F. Dignum, H. Weigand, E. Verharen	A Formal Specification of Deadlines using Dynamic Deontic Logic, p. 18.
96/10	R. Bloo, H. Geuvers	Explicit Substitution: on the Edge of Strong Normalisation, p. 13.
96/11	T. Laan	AUTOMATH and Pure Type Systems, p. 30.
96/12	F. Kamareddine and T. Laan	A Correspondence between Nuprl and the Ramified Theory of Types, p. 12.
96/13	T. Borghuis	Priorean Tense Logics in Modal Pure Type Systems, p. 61
96/14	S.H.J. Bos and M.A. Reniers	The I^2C-bus in Discrete-Time Process Algebra, p. 25.
96/15	M.A. Reniers and J.J. Vereijken	Completeness in Discrete-Time Process Algebra, p. 139.
96/17	E. Boiten and P. Hoogendijk	Nested collections and polytypism, p. 11.
96/18	P.D.V. van der Stok	Real-Time Distributed Concurrency Control Algorithms with mixed time constraints, p. 71.
96/19	M.A. Reniers	Static Semantics of Message Sequence Charts, p. 71
96/20	L. Feijs	Algebraic Specification and Simulation of Lazy Functional Programs in a concurrent Environment, p. 27.
96/21	L. Bijlsma and R. Nederpelt	Predicate calculus: concepts and misconceptions, p. 26.
96/22	M.C.A. van de Graaf and G.J. Houben	Designing Effective Workflow Management Processes, p. 22.
96/23	W.M.P. van der Aalst	Structural Characterizations of sound workflow nets, p. 22.
96/24	M. Voorhoeve and W. van der Aalst	Conservative Adaption of Workflow, p.22
96/25	M. Vaccari and R.C. Backhouse	Deriving a systolic regular language recognizer, p. 28
97/02	J. Hooman and O. v. Roosmalen	A Programming-Language Extension for Distributed Real-Time Systems, p. 50.
97/03	J. Blanco and A. v. Deursen	Basic Conditional Process Algebra, p. 20.
97/04	J.C.M. Baeten and J.A. Bergstra	Discrete Time Process Algebra: Absolute Time, Relative Time and Parametric Time, p. 26.
97/05	J.C.M. Baeten and J.J. Vereijken	Discrete-Time Process Algebra with Empty Process, p. 51.
97/06	M. Franssen	Tools for the Construction of Correct Programs: an Overview, p. 33.
97/07	J.C.M. Baeten and J.A. Bergstra	Bounded Stacks, Bags and Queues, p. 15.
97/08	P. Hoogendijk and R.C. Backhouse	When do datatypes commute? p. 35.

97/09	Proceedings of the Second International Workshop on Communication Modeling, Veldhoven, The Netherlands, 9-10 June, 1997.	Communication Modeling- The Language/Action Perspective, p. 147.
97/10	P.C.N. v. Gorp, E.J. Luit, D.K. Hammer E.H.L. Aarts	Distributed real-time systems: a survey of applications and a general design model, p. 31.
97/11	A. Engels, S. Mauw and M.A. Reniers	A Hierarchy of Communication Models for Message Sequence Charts, p. 30.
97/12	D. Hauschildt, E. Verbeek and W. van der Aalst	WOFLAN: A Petri-net-based Workflow Analyzer, p. 30.
97/13	W.M.P. van der Aalst	Exploring the Process Dimension of Workflow Management, p. 56.
97/14	J.F. Groote, F. Monin and J. Springintveld	A computer checked algebraic verification of a distributed summation algorithm, p. 28
97/15	M. Franssen	λP-: A Pure Type System for First Order Loginc with Automated Theorem Proving, p.35.
97/16	W.M.P. van der Aalst	On the verification of Inter-organizational workflows, p. 23
97/17	M. Vaccari and R.C. Backhouse	Calculating a Round-Robin Scheduler, p. 23.
97/18	Werkgemeenschap Informatiewetenschap redactie: P.M.E. De Bra	Informatiewetenschap 1997 Wetenschappelijke bijdragen aan de Vijfde Interdisciplinaire Conferentie Informatiewetenschap, p. 60.
98/01	W. Van der Aalst	Formalization and Verification of Event-driven Process Chains, p. 26.
98/02	M. Voorhoeve	State / Event Net Equivalence, p. 25
98/03	J.C.M. Baeten and J.A. Bergstra	Deadlock Behaviour in Split and ST Bisimulation Semantics, p. 15.
98/04	R.C. Backhouse	Pair Algebras and Galois Connections, p. 14
98/05	D. Dams	Flat Fragments of CTL and CTL*: Separating the Expressive and Distinguishing Powers. P. 22.
98/06	G. v.d. Bergen, A. Kaldewaij V.J. Dielissen	Maintenance of the Union of Intervals on a Line Revisited, p. 10.
98/07	Proceedings of the workhop on Workflow Management: Net-based Concepts, Models, Techniques and Tools (WFM'98) June 22, 1998 Lisbon, Portugal	edited by W. v.d. Aalst, p. 209
98/08	Informal proceedings of the Workshop on User Interfaces for Theorem Provers. Eindhoven University of Technology ,13-15 July 1998	edited by R.C. Backhouse, p. 180
98/09	K.M. van Hee and H.A. Reijers	An analytical method for assessing business processes, p. 29.
98/10	T. Basten and J. Hooman	Process Algebra in PVS
98/11	J. Zwanenburg	The Proof-assistemt Yarrow, p. 15
98/12	Ninth ACM Conference on Hypertext and Hypermedia Hypertext '98 Pittsburgh, USA, June 20-24, 1998 Proceedings of the second workshop on Adaptive Hypertext and Hypermedia.	Edited by P. Brusilovsky and P. De Bra, p. 95.
98/13	J.F. Groote, F. Monin and J. v.d. Pol	Checking verifications of protocols and distributed systems by computer. Extended version of a tutorial at CONCUR'98, p. 27.
98/14	T. Verhoeff (artikel volgt)	
99/01	V. Bos and J.J.T. Kleijn	Structured Operational Semantics of χ , p. 27
99/02	H.M.W. Verbeek, T. Basten and W.M.P. van der Aalst	Diagnosing Workflow Processes using Woflan, p. 44
99/03	R.C. Backhouse and P: Hoogendijk	Final Dialgebras: From Categories to Allegories, p. 26
99/04	S. Andova	Process Algebra with Interleaving Probabilistic Parallel Composition, p. 81

99/05	M. Franssen, R.C. Veltkamp and W. Wesselink	Efficient Evaluation of Triangular B-splines, p. 13
99/06	T. Basten and W. v.d. Aalst	Inheritance of Workflows: An Approach to tackling problems related to change, p. 66
99/07	P. Brusilovsky and P. De Bra	Second Workshop on Adaptive Systems and User Modeling on the World Wide Web, p. 119.
99/08	D. Bosnacki, S. Mauw, and T. Willemse	Proceedings of the first international syposium on Visual Formal Methods - VFM'99
99/09	J. v.d. Pol, J. Hooman and E. de Jong	Requirements Specification and Analysis of Command and Control Systems
99/10	T.A.C. Willemse	The Analysis of a Conveyor Belt System, a case study in Hybrid Systems and timed μ CRL, p. 44.
99/11	J.C.M. Baeten and C.A. Middelburg	Process Algebra with Timing: Real Time and Discrete Time, p. 50.
99/12	S. Andova	Process Algebra with Probabilistic Choice, p. 38.
99/13	K.M. van Hee, R.A. van der Toorn, J. van der Woude and P.A.C. Verkoulen	A Framework for Component Based Software Architectures, p. 19
99/14	A. Engels and S. Mauw	Why men (and octopuses) cannot juggle a four ball cascade, p. 10

Computing Science Reports

Department of Mathematics and Computing Science
Eindhoven University of Technology

In this series appeared:

96/01	M. Voorhoeve and T. Basten	Process Algebra with Autonomous Actions, p. 12.
96/02	P. de Bra and A. Aerts	Multi-User Publishing in the Web: DreSS, A Document Repository Service Station, p. 12
96/03	W.M.P. van der Aalst	Parallel Computation of Reachable Dead States in a Free-choice Petri Net, p. 26.
96/05	T. Basten and W.M.P. v.d. Aalst	A Process-Algebraic Approach to Life-Cycle Inheritance Inheritance = Encapsulation + Abstraction, p. 15.
96/06	W.M.P. van der Aalst and T. Basten	Life-Cycle Inheritance A Petri-Net-Based Approach, p. 18.
96/07	M. Voorhoeve	Structural Petri Net Equivalence, p. 16.
96/08	A.T.M. Aerts, P.M.E. De Bra, J.T. de Munk	OODB Support for WWW Applications: Disclosing the internal structure of Hyperdocuments, p. 14.
96/09	F. Dignum, H. Weigand, E. Verharen	A Formal Specification of Deadlines using Dynamic Deontic Logic, p. 18.
96/10	R. Bloo, H. Geuvers	Explicit Substitution: on the Edge of Strong Normalisation, p. 13.
96/11	T. Laan	AUTOMATH and Pure Type Systems, p. 30.
96/12	F. Kamareddine and T. Laan	A Correspondence between Nuprl and the Ramified Theory of Types, p. 12.
96/13	T. Borghuis	Priorean Tense Logics in Modal Pure Type Systems, p. 61
96/14	S.H.J. Bos and M.A. Reniers	The I^2C-bus in Discrete-Time Process Algebra, p. 25.
96/15	M.A. Reniers and J.J. Vereijken	Completeness in Discrete-Time Process Algebra, p. 139.
96/17	E. Boiten and P. Hoogendijk	Nested collections and polytypism, p. 11.
96/18	P.D.V. van der Stok	Real-Time Distributed Concurrency Control Algorithms with mixed time constraints, p. 71.
96/19	M.A. Reniers	Static Semantics of Message Sequence Charts, p. 71
96/20	L. Feijs	Algebraic Specification and Simulation of Lazy Functional Programs in a concurrent Environment, p. 27.
96/21	L. Bijlsma and R. Nederpelt	Predicate calculus: concepts and misconceptions, p. 26.
96/22	M.C.A. van de Graaf and G.J. Houben	Designing Effective Workflow Management Processes, p. 22.
96/23	W.M.P. van der Aalst	Structural Characterizations of sound workflow nets, p. 22.
96/24	M. Voorhoeve and W. van der Aalst	Conservative Adaption of Workflow, p.22
96/25	M. Vaccari and R.C. Backhouse	Deriving a systolic regular language recognizer, p. 28
97/02	J. Hooman and O. v. Roosmalen	A Programming-Language Extension for Distributed Real-Time Systems, p. 50.
97/03	J. Blanco and A. v. Deursen	Basic Conditional Process Algebra, p. 20.
97/04	J.C.M. Baeten and J.A. Bergstra	Discrete Time Process Algebra: Absolute Time, Relative Time and Parametric Time, p. 26.
97/05	J.C.M. Baeten and J.J. Vereijken	Discrete-Time Process Algebra with Empty Process, p. 51.
97/06	M. Franssen	Tools for the Construction of Correct Programs: an Overview, p. 33.
97/07	J.C.M. Baeten and J.A. Bergstra	Bounded Stacks, Bags and Queues, p. 15.
97/08	P. Hoogendijk and R.C. Backhouse	When do datatypes commute? p. 35.

97/09	Proceedings of the Second International Workshop on Communication Modeling, Veldhoven, The Netherlands, 9-10 June, 1997.	Communication Modeling- The Language/Action Perspective, p. 147.
97/10	P.C.N. v. Gorp, E.J. Luit, D.K. Hammer E.H.L. Aarts	Distributed real-time systems: a survey of applications and a general design model, p. 31.
97/11	A. Engels, S. Mauw and M.A. Reniers	A Hierarchy of Communication Models for Message Sequence Charts, p. 30.
97/12	D. Hauschildt, E. Verbeek and W. van der Aalst	WOFLAN: A Petri-net-based Workflow Analyzer, p. 30.
97/13	W.M.P. van der Aalst	Exploring the Process Dimension of Workflow Management, p. 56.
97/14	J.F. Groote, F. Monin and J. Springintveld	A computer checked algebraic verification of a distributed summation algorithm, p. 28
97/15	M. Franssen	λP-: A Pure Type System for First Order Loginc with Automated Theorem Proving, p.35.
97/16	W.M.P. van der Aalst	On the verification of Inter-organizational workflows, p. 23
97/17	M. Vaccari and R.C. Backhouse	Calculating a Round-Robin Scheduler, p. 23.
97/18	Werkgemeenschap Informatiewetenschap redactie: P.M.E. De Bra	Informatiewetenschap 1997 Wetenschappelijke bijdragen aan de Vijfde Interdisciplinaire Conferentie Informatiewetenschap, p. 60.
98/01	W. Van der Aalst	Formalization and Verification of Event-driven Process Chains, p. 26.
98/02	M. Voorhoeve	State / Event Net Equivalence, p. 25
98/03	J.C.M. Baeten and J.A. Bergstra	Deadlock Behaviour in Split and ST Bisimulation Semantics, p. 15.
98/04	R.C. Backhouse	Pair Algebras and Galois Connections, p. 14
98/05	D. Dams	Flat Fragments of CTL and CTL*: Separating the Expressive and Distinguishing Powers. P. 22.
98/06	G. v.d. Bergen, A. Kaldewaij V.J. Dielissen	Maintenance of the Union of Intervals on a Line Revisited, p. 10.
98/07	Proceedings of the workhop on Workflow Management: Net-based Concepts, Models, Techniques and Tools (WFM'98) June 22, 1998 Lisbon, Portugal	edited by W. v.d. Aalst, p. 209
98/08	Informal proceedings of the Workshop on User Interfaces for Theorem Provers. Eindhoven University of Technology ,13-15 July 1998	edited by R.C. Backhouse, p. 180
98/09	K.M. van Hee and H.A. Reijers	An analytical method for assessing business processes, p. 29.
98/10	T. Basten and J. Hooman	Process Algebra in PVS
98/11	J. Zwanenburg	The Proof-assistemt Yarrow, p. 15
98/12	Ninth ACM Conference on Hypertext and Hypermedia Hypertext '98 Pittsburgh, USA, June 20-24, 1998 Proceedings of the second workshop on Adaptive Hypertext and Hypermedia.	Edited by P. Brusilovsky and P. De Bra, p. 95.
98/13	J.F. Groote, F. Monin and J. v.d. Pol	Checking verifications of protocols and distributed systems by computer. Extended version of a tutorial at CONCUR'98, p. 27.
98/14	T. Verhoeff (artikel volgt)	
99/01	V. Bos and J.J.T. Kleijn	Structured Operational Semantics of χ , p. 27
99/02	H.M.W. Verbeek, T. Basten and W.M.P. van der Aalst	Diagnosing Workflow Processes using Woflan, p. 44
99/03	R.C. Backhouse and P. Hoogendijk	Final Dialgebras: From Categories to Allegories, p. 26
99/04	S. Andova	Process Algebra with Interleaving Probabilistic Parallel Composition, p. 81

99/05	M. Franssen, R.C. Veltkamp and W. Wesselink	Efficient Evaluation of Triangular B-splines, p. 13
99/06	T. Basten and W. v.d. Aalst	Inheritance of Workflows: An Approach to tackling problems related to change, p. 66
99/07	P. Brusilovsky and P. De Bra	Second Workshop on Adaptive Systems and User Modeling on the World Wide Web, p. 119.
99/08	D. Bosnacki, S. Mauw, and T. Willemse	Proceedings of the first international syposium on Visual Formal Methods - VFM'99
99/09	J. v.d. Pol, J. Hooman and E. de Jong	Requirements Specification and Analysis of Command and Control Systems
99/10	T.A.C. Willemse	The Analysis of a Conveyor Belt System, a case study in Hybrid Systems and timed μ CRL, p. 44.
99/11	J.C.M. Baeten and C.A. Middelburg	Process Algebra with Timing: Real Time and Discrete Time, p. 50.
99/12	S. Andova	Process Algebra with Probabilistic Choice, p. 38.
99/13	K.M. van Hee, R.A. van der Toorn, J. van der Woude and P.A.C. Verkoulen	A Framework for Component Based Software Architectures, p. 19
99/14	A. Engels and S. Mauw	Why men (and octopuses) cannot juggle a four ball cascade, p. 10