BRICS

Basic Research in Computer Science

Optimal Time-Space Trade-Offs for Sorting

Jakob Pagter
Theis Rauhe

BRICS Report Series RS-98-10

ISSN 0909-0878 May 1998

See back inner page for a list of recent BRICS Report Series publications.
Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark

Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `RS/98/10/`

Optimal Time-Space Trade-Offs for Sorting

(extended abstract)

Jakob Pagter *   Theis Rauhe*†

BRICS‡
Department of Computer Science
University of Aarhus
Denmark

Abstract

We study the fundamental problem of sorting in a sequential model of computation and in particular consider the time-space trade-off (product of time and space) for this problem.

Beame has shown a lower bound of $\Omega(n^2)$ for this product leaving a gap of a logarithmic factor up to the previously best known upper bound of $O(n^2 \log n)$ due to Frederickson. Since then, no progress has been made towards tightening this gap.

The main contribution of this paper is a comparison based sorting algorithm which closes this gap by meeting the lower bound of Beame. The time-space product $O(n^2)$ upper bound holds for the full range of space bounds between $\log n$ and $n/\log n$. Hence in this range our algorithm is optimal for comparison based models as well as for the very powerful general models considered by Beame.

*Supported by the ESPRIT Long Term Research Programme of the EU under project number 20244 (ALCOM-IT). E-mail: {pagter,theis}@brics.dk.

†Part of this work was done while the author was visiting the Fields Institute, Toronto, Canada.

‡Basic Research in Computer Science,
Centre of the Danish National Research Foundation.

1 Introduction

Motivation and results. The complexity of sorting is a classical problem in computer science which has provided a wide scope of both algorithms and lower bounds (see Knuth [1] and Andersson [2] for an overview of classical as well as more recent work in the area). One fruitful line of research has been the investigation of the trade-off between the two most fundamental complexity measures; *time and space* — pioneered by Cobham [3].

Accordingly, time-space trade-offs for sorting is a much studied problem [4, 5, 6, 7, 8, 9]. Despite the successes of this work, a discrepancy of at least a logarithmic factor (referred to in [4, 6, 7, 10, 11]) between the best known upper bound — $\mathrm{O}(n^2 \log n)$ [5], and the best known lower bound — $\Omega(n^2)$ [4] — remained. The main contribution of this paper is an algorithm which closes this gap:

Theorem 1 *There exists positive constants c_1 and c_2 such that for any S in the interval $c_1 \log n \leq S \leq c_2 n/\log n$, there is a comparison based algorithm sorting n keys in time T and space S, with $T \cdot S = \mathrm{O}(n^2)$.*

Hence by Beame's result [4] we obtain as a corollary that the time-space trade-off complexity is $\Theta(n^2)$ for the full range of space bounds between $\log n$ and $n/\log n$ in general sequential models of computation. Clearly this range is maximal for comparison based sorting.

Our result involves a new technique for sorting based on Tree Selection (see [1]). Our algorithm is relatively simple, and involves only small constants, hence it is our opinion that it is of practical as well as theoretical relevance.

Related work. A general survey of time-space trade-offs is given by Borodin in [12]. An introduction to the area is given by Savage [13].

Upper bounds. Classical sorting algorithms like QuickSort and MergeSort, have $T \cdot S = \mathrm{O}(n^2 \log^2 n)$. This stems from the fact that space is measured in bits, and these algorithm use $\mathrm{O}(n)$ log-sized words and $\mathrm{O}(n \log n)$ comparisons. With respect to trade-offs, these algorithms have the weakness that time cannot be traded for space or vice versa; i.e., the time-space product only holds for "fixed" functions T and S of n.

Munro and Paterson [14] gave the first time-space focused algorithm, realising $T \cdot S = \mathrm{O}(n^2 \log n)$, but still the time-space product is only realisable for $S = \Theta(\log n)$. The first fully scalable, and till now the best upper bound for the time-space trade-off was given by Frederickson [5], who showed that for $4\log n \leq S = \mathrm{O}(n)$ there exists a comparison based RAM algorithm sorting n keys in time T, and space S, such that $T \cdot S = \mathrm{O}(n^2 \log n)$.

Of course, these upper bounds also hold in models more general than the comparison based one, but in such stronger models other algorithms are also possible. An example of a non-comparison based algorithm realising $T \cdot S = \mathrm{O}(n^2 \log n)$ is radix-sorting (for keys of size $\mathrm{O}(n^k)$).

With the strong restriction that all keys to be sorted are between 1 and n, Beame [4] exhibits a general branching program realising $T \cdot S = \mathrm{O}(n^2)$. Besides restricting the key-size, this algorithm has the problem that it is not clear how to construct a uniform (e.g. RAM) version. On the other hand, it shows optimality (for restricted keys) of the $\Omega(n^2)$ lower bound.

Lower bounds. The first non-trivial lower bound for non-oblivious sorting was given by Borodin et al. [7], who showed that any comparison based branching program (see Section 2) sorting n keys has $T \cdot S = \Omega(n^2)$. This was followed by a result of Borodin and Cook, showing that any general branching program sorting n keys has $T \cdot S = \Omega(n^2/\log n)$. Their result was improved by Beame [4], who showed that any general branching program sorting n keys in time T, and space S, has $T \cdot S = \Omega(n^2)$.

Set problems. Beame's result is actually obtained by a simple reduction from the unique elements problem (given n keys, output all those that appear exactly once). For keys between 1 and n Beame gives a matching upper bound. Our result implies optimality without this restriction on the range of the keys.

Another set problem is element distinctness (given n keys, output "1" iff they are all distinct, otherwise output "0".) Yao [11] has shown that any comparison based branching program solving the element distinctness problem on n keys, must have $T \cdot S = \Omega(n^{2-\epsilon(n)})$, where $\epsilon(n)$ is decreasing in n.

Patt-Shamir and Peleg [15] studies a number of other set problems, including: set complementation (given a set X from some finite subset of an ordered universe U, output X^C), set subtraction (given X and Y, output $X \setminus Y$), symmetric difference (given X and Y, output $(X \cup Y) \setminus (X \cap Y)$). For these problems they give a lower bound of $\Omega(n^2)$ on general branching programs. For sets containing only elements between 1 and n, they show that these results are optimal. Our result implies optimality without restriction on the range of the keys.

Notice that of all the set problems mentioned, only element distinctness and set disjointness are decision problems, and for these problems trade-off results have only been shown in the comparison based model.

Outline of paper. Section 2 contains some preliminaries including a definition of the computational model used. In Section 3 we state our formal result and describe our algorithm. We finish with some concluding remarks in Section 4.

2 Preliminaries

Computational Models. The model used to show our upper bounds is the comparison based RAM. For completeness, we also comment briefly on branching programs.

Comparison based RAM. One model which can be used for showing upper bounds is a purely combinatorial one like branching programs, described below, but describing algorithms in such an abstract manner can be both troublesome and non-constructive; instead more constructive models like the RAM are normally used for showing upper bounds.

A comparison based RAM is a unit-cost RAM with word-size $\Theta(\log n)$, read-only random access to the input, and write-only access to the output (allowing for a fair space analysis); the registers which are not used for input nor output, are called workspace. The input can only be accessed through unit-cost comparisons of two elements x_i and x_j (this could be regarded as input represented as an $n \times n$-matrix A, such that $A(i, j) = 1$ if and only if $x_i > x_j$).

As usual the time used by a comparison based RAM algorithm is the number of operations executed and the space usage is the maximal number of bits used, i.e., the number of necessary registers in the algorithms workspace times the word size.

Branching programs. Most time-space trade-offs lower bounds for sorting and similar problems are proved for the non-uniform branching program model [4, 6, 7, 8, 9, 10, 11, 15, 16]. Branching programs are mainly used in two variants: 1. Comparison based branching programs (also known as decision branching programs); this is a comparison based model strong enough to real-time simulate (i.e., adding only constant factors on time and space requirements) comparison based RAMs. 2. General branching programs (also known as R-way branching programs); this is a general model of computation with the only restriction that at most one out of the n input values may be accessed per "time-step". General branching programs can real-time simulate a unit-cost RAM with *any* instruction set. A thorough account of branching programs is given by Savage [13].

Notation. Throughout the paper, we assume that n and m are integer powers of two to ease exposition.

Consider a vector $x = (x_1, \ldots, x_n) \in U^n$ for some universe U. We denote the set $\{x_1, \ldots, x_n\}$ by $\{x\}$.

An *m-division* of x is a division of x into m consecutive *sub-vectors* each of length n/m; sub-vector b_i consists of $(x_{in/m}, \ldots, x_{((i+1)n/m)-1})$. An *$m$-tree* related to an m-division of x, is a complete rooted binary tree with m leaves (i.e., of height $\log m + 1$ and with $2m - 1$ nodes).

Each leaf, l_i $(0 \leq i < n/m)$, is associated with sub-vector b_i of the division of x; the root is called r. For a non-leaf node, v, we denote its left and right child v_L and v_R respectively. For each node v define:

$$x(v) = \begin{cases} b_i & \text{if } v = l_i \\ x(v_L) \circ x(v_R) & \text{otherwise} \end{cases}$$

where $x \circ y$ means x concatenated with y. In words, $x(v)$ is the sub-vector "dominated" by v.

An m-tree can be represented by an array, where information in nodes is explicitly represented, whereas the edges are given implicitly (like the classical implementation of a heap).

Consider a vector $x \in U^n$ with associated m-tree, and let Ψ be some subset of $\{x\}$. For each node v define:

$$\Psi(v) = \{x(v)\} \cap \Psi.$$

Notice that $\Psi(r) = \Psi$.

3 Main result

Our main result is derived from a time-space efficient data structure for the following problem (the model is a comparison based RAM). Given $x =$

$(x_1, \ldots, x_n) \in U^n$ from an ordered universe U we want to construct a data structure, D_Ψ, that maintains a set $\Psi \subseteq U$ under the following operations:

INIT(x): $\Psi \leftarrow \{x\}$.

POP: report the index in x of element $y = \min \Psi$ and assign $\Psi \leftarrow \Psi \setminus y$.

Lemma 1 *For $m \geq \log n$ there exists a comparison based RAM algorithm which supports the operation* INIT(x) *in time* $O(n + m \log m)$*, the operation* POP *in time* $O(n/m + \log^2 m)$*, and whose space usage is* $O(m)$ *bits.*

Our data structure consists of a new technique for Tree Selection [1] (similar to HeapSort). The basic idea of Tree Selection is to maintain a binary tree with n leaves (a 1-tree), such that each node, v, contains the smallest key of $x(v)$ not yet reported; instead one can also represent the keys by their indices which means that each element can be represented by $O(\log n)$ bits. The time usage of this approach is $O(\log n)$ to report an element, and the space usage is $O(n \log n)$. Hence, the time-space product for sorting using this Tree Selection becomes $O(n^2 \log^2 n)$.

Proof of Lemma 1. The first step of the proof will be the description of a data structure, D_Ψ, which *almost* does the job. The second step will describe an extension of this data structure, which gives the desired result. Finally a formal analysis of time and space requirements will be given.

We assume without loss of generality that the elements in x are distinct. That is, for a pair of elements where $x_i = x_j$ for some $i < j$, we consider x_i to be "less than" x_j, i.e., our sorting algorithm performs *stable sorting.*

Step 1: The initial data structure. D_Ψ consists of an m-tree, T_Ψ, and an index λ between 1 and n. With each node $v \in T_\Psi$ we associate a bit *state*(v) $\in \{\mathrm{L}, \mathrm{R}\}$ such that the following invariant is satisfied:

$$state(v) = \begin{cases} \mathrm{L} & \text{if } \min \Psi(v) \in \Psi(v_L) \\ \mathrm{R} & \text{otherwise.} \end{cases} \tag{1}$$

Clearly, for any node v, we can find the sub-vector b_i containing $\min \Psi(v)$ by simply following the path from v going left or right depending upon whether *state*(v) is L or R. We call this path the *selection path* from v.

Besides the tree we keep the index λ of most recently reported element in x. Hence λ satisfies the invariant:

$$\Psi(v) = \{\, x_i \in \{x(v)\} \mid x_i > x_\lambda \}. \tag{2}$$

For any node v, the index λ implicitly allows us to distinguish between whether a specified element in $x(v)$ has been reported or not.

Suppose the invariants (1) and (2) are satisfied for D_Ψ. Then the operation Pop can be implemented as follows. Follow the selection path to the sub-vector b_i, and then perform a brute-force (linear) search in b_i in order to find the smallest x_j such that $x_j > x_\lambda$. Report j and assign $\lambda \leftarrow j$. Now (2) holds, but the state information along the processed selection path may no longer be correct i.e., we need to reestablish (1). This is done simply by finding the new minimum of $\Psi(v)$ for each node v along the processed selection path, using the information from the selection paths of the children of v.

Consider the space cost of the above approach. For each node we only need to represent a single bit. Using standard RAM techniques we can represent the tree in an array using $O(m/w)$ machine words of w bits. In addition to this array we only need a constant number of registers for local computation and the representation of λ. Hence the space cost is $O(m + \log n)$, which is the allowed $O(m)$ when $m = \Omega(\log n)$; unfortunately the time cost is $O((n/m)\log m)$ per Pop operation (for each of the $\log m$ nodes on the selection path we perform a brute-force search taking worst-case time $O(n/m)$). Our goal is to improve the data structure, D_Ψ, in order to get rid of the multiplicative $\log m$ factor for time cost, while keeping space cost $O(m)$.

Step 2: The extended data structure. Intuitively speaking the key idea is to associate an additional amount of extra information to each node in the tree. The purpose of this information is to obtain an exponential decrease of the size of the sub-vectors in which the brute-force search is performed. Summing the time costs along the selection path, this exponential speed-up along the path leads to the claimed time bound. The exponential decrease of the search domains, is obtained by using some additional bits within each node. The necessary number of such bits in a node will be proportional to the level of the node, i.e., a constant number of bits for nodes near the leaf level and up to $\log m$ bits for nodes near the root. The final analysis shows that such a scheme still allows total space cost to be bounded by $O(m)$.

Formally we extend D_Ψ as follows to obtain the improvement. For each node v on level t (the root having level 0) maintain an integer, $sb(v)$, between 0 and $m/2^t$ (using $\log m - t$ bits). Suppose $\min \Psi(v)$ is in sub-vector b_i; consider an $m/2^t$-division of b_i into $m/2^t$ sub-vectors, denoted $b_{i,j}$ ($0 \le j < m/2^t$) of length $(n/m^2)2^t$. Then $sb(v)$ satisfies the following invariant

$$\min \Psi(v) \in b_{i,sb(v)}. \tag{3}$$

Indeed given $sb(v)$, a brute-force search for the minimal element in only needs to be concerned with the "small" sub-vector $b_{i,sb(v)}$.

With this additional information in D_Ψ, Pop behaves as follows: first it follows the selection path from the root to the relevant sub-vector b_i of x; it then performs brute-force search in the sub-vector $b_{i,sb(r)}$ (of length n/m^2) and obtain the next minimum to be reported; then λ is updated accordingly to reestablish (2) and finally invariants (1) and (3) are reestablished level by level along the way back to the root. To reestablish (1) and (3) for a given node v, we compute the indices l and r of $\min \Psi(v_L)$ and $\min \Psi(v_R)$ using the information of the sub-trees rooted by v_L and v_R (both with invariants established since we are going bottom up). Let p be the index of $\min(x_l, x_r)$ and set $sb(v) \leftarrow (p - i(n/m))$ div $m/2^t$ assuming x_p is from sub-vector b_i. Finally we set $\mathit{state}(v) \leftarrow$ L if $x_l < x_r$ and $\mathit{state}(v) \leftarrow$ R otherwise.

Init(x) behaves as follows. First (1) is established by assigning $\lambda = 0$ (assume x_0 to be some special value such that $x_i > x_0$ for $1 \le i \le n$). Afterwards T_Ψ is built bottom-up, one level at a time, establishing (2) and (3).

Time and space complexity. As before we note that using standard RAM techniques, we may pack the tree into a compact array such that the total number of bits needed in all nodes of the tree amounts to the space cost of this array (i.e., for a tree consuming m bits, we only use $O(m/w)$ registers with several "small" words with respect to different nodes packed into a single register). With this in mind, all we need is to count the number of bits used.

The total number of bits needed is then the sum of the number of bits used per node plus the small amount of at most $O(\log n)$ additional bits used to keep λ and temporary work space used for computation along the selection path (i.e., keeping information such as current level, constant number of indices in input etc.). The total number of bits used for nodes at level t is bounded by:

$$\sum_{v \text{ at level } t} (\log(m/2^t) + 1) = 2^t(\log m - t + 1)$$

and hence the total number of bits used for maintaining information in all nodes of our tree is bounded by

$$\sum_{t=0}^{\log m} 2^t(\log m - t + 1) = \sum_{t=0}^{\log m} 2^t + \sum_{t=0}^{\log m} 2^t(\log m - t) < 4m.$$

Hence in total, the space cost is $\mathrm{O}(m)$ bits as desired.

In order to bound the time cost we consider the work done at each level of the selection path. At level t the brute-force computations need to make comparisons of a total of $(n/m^2)2^t$ elements in $\{x\}$ together with at most $\mathrm{O}(\log m)$ additional work used to follow the selection path to appropriate sub-vector. The rest of the computation at the level has constant time cost. Hence the total time cost is bounded by

$$\mathrm{O}\left(\sum_{t=1}^{\log m}((n/m^2)2^t + \log m)\right) = \mathrm{O}(n/m + \log^2 m)$$

as desired.

With respect to $\mathrm{INIT}(x)$, we use no more space than the rest of the algorithm, since we only use a constant number of $\mathrm{O}(\log n)$ sized words besides the data structure we are building. The time spent building one level, t, is $\mathrm{O}(2^t(\log m + (n/m^2)2^t))$, yielding a total initialisation time of

$$\mathrm{O}\left(\sum_{t=1}^{\log m} 2^t((n/m^2)2^t + \log m)\right) = \mathrm{O}(n + m\log m)$$

as desired. $\Box$

Calling $\mathrm{INIT}(x)$ followed by n sub-sequent calls to POP, will report the elements of x in sorted order. If $m = \mathrm{O}(n/\log^2 n)$, certainly initialisation time will be $\mathrm{O}(n)$. Thus, if we let $m = S$ we can sort n keys on a comparison based RAM in time T, and space S, with $T \cdot S = \mathrm{O}(n^2)$, for $c_1 \log n \leq S \leq c_2 n/\log^2 n$, for appropriate positive constants c_1 and c_2.

The range in which we can realise this trade-off is not maximal — a factor $\log n$ is missing. Combining our ideas with those of Frederickson [5] we can remove this logarithmic factor, and obtain the trade-off in the maximal range:

Theorem 1 *There exists positive constants c_1 and c_2 such that for any S in the interval $c_1 \log n \leq S \leq c_2 n/\log n$, there is a comparison based algorithm sorting n keys in time T and space S, with $T \cdot S = \mathrm{O}(n^2)$.*

Proof. The idea is to use Lemma 1 as subroutine in Frederickson's algorithm [5]. We will employ an extra space measure S_w, for the number of $\Theta(\log n)$-sized words used, because Frederickson only uses words of this size. This implies $S = \Theta(S_w \log n)$.

The idea behind Frederickson's algorithm is to split the input into S_w sub-vectors, each consisting of n/S_w elements. A heap is maintained based on the current minima of each sub-vector. This algorithm uses time $O(n/S_w)$ to report an element and update the heap with the reported element's successor (from the relevant sub-vector). The heap uses space $O((n/S_w) \cdot \log n)$, since we have n/S_w pointers to elements in the queue and each take $\log n$ bits.

For a suitable size n/S_w of sub-vectors in Frederickson's algorithm, use our algorithm as a sub-routine on each of these sub-vectors with $m = \log n$. This will decrease the asymptotic running time with a factor $O(\log n)$. The asymptotic space usage will be the same, since $O(\log n)$ is exactly the amount of space already used per sub-vector in Frederickson's algorithm. This will work for $\log^2 n \leq n/S_w = O(n)$, i.e., $c_1 \log n \leq S \leq c_2 n/\log n$, for appropriate positive constants c_1 and c_2. $\square$

The following corollary is immediate from Theorem 1 and the result of Beame [4].

Corollary 1 *The time-space trade-off for sorting n keys using time T and space S, in a general sequential model of computation, is $T \cdot S = \Theta(n^2)$, for $c_1 \log n \leq S \leq c_2 n/\log n$, for appropriate positive constants c_1 and c_2.*

4 Conclusions

In this paper we have proved that the sequential time-space complexity of sorting is $\Theta(n^2)$, for time in $\Omega(n \log n)$. It is worth noting that in the specified range, this complexity is optimal for comparison based as well as general models. This means, perhaps surprisingly, that with respect to the time-space product for sorting, comparison based models are as strong as more general models.

5 Acknowledgements

The authors would like to thank Gudmund S. Frandsen, Peter Bro Miltersen, Rasmus Pagh, Erik Meineche Schmidt and Sven Skyum for helpful comments and discussions. Special thanks to Peter for enthusiastic encouragement.

References

[1] D. E. Knuth, *Sorting and Searching*, vol. 3 of *The Art of Computer Programming*. Addison-Wesley, 1973.

[2] A. Andersson, "Sorting and Searching Revisited," in *Algorithm Theory - SWAT'96* (R. Karlsson and A. Lingas, eds.), pp. 185–197, Springer, July 1996.

[3] A. Cobham, "The Recognition Problem for the Set of Perfect Squares," in *Conference Record of 1966 Seventh Annual Symposium on Switching and Automata Theory*, pp. 78–87, IEEE, 1966. Conference also known as FOCS 7.

[4] P. Beame, "A General Sequential Time-Space Tradeoff for Finding Unique Elements," *SIAM Journal on Computing*, vol. 20, pp. 270–277, 1991.

[5] G. N. Frederickson, "Upper Bounds for Time-Space Trade-offs in Sorting and Selection," *Journal of Computer and Systems Sciences*, vol. 34, pp. 19–26, 1987.

[6] A. Borodin and S. Cook, "A Time-Space Tradeoff for Sorting on a General Sequential Model of Computation," *SIAM Journal on Computing*, vol. 11, no. 2, pp. 287–297, 1982.

[7] A. Borodin, M. J. Fischer, D. G. Kirkpatrick, N. A. Lynch, and M. Tompa, "A Time-Space Tradeoff for Sorting on Non-Oblivious Machines," *Journal of Computer and System Sciences*, vol. 22, pp. 351–364, 1981.

[8] S. Reisch and G. Schnitger, "Three Applications of Kolmogorov-Complexity," in *23rd Annual Symposium on Foundations of Computer Science*, pp. 45–52, IEEE, 1982.

[9] A. C.-C. Yao, “On The Time-Space Tradeoff For Sorting With Linear Queries,” *Teoretical Computer Science*, vol. 19, pp. 203–218, 1982.

[10] A. Borodin, F. Fich, F. M. auf der Heide, E. Upfal, and A. Wigderson, “A Time-Space Tradeoff for Element Distinctness,” *SIAM Journal on Computing*, vol. 16, pp. 97–99, 1987.

[11] A. C.-C. Yao, “Near-optimal Time-Space Tradeoff for Element Distinctness,” *SIAM Journal on Computing*, vol. 23, pp. 966–975, 1994.

[12] A. Borodin, “Time Space Tradeoffs (Getting Closer to the Barrier?),” in *Algorithms and Computation; 4th International Symposium, ISAAC'93* (K. Ng, P. Raghavan, N. Balasubramanian, and F. Chin, eds.), Springer-Verlag, LNCS 762, December 1993.

[13] J. E. Savage, *Models of Computation.* Addison Wesley Longman, 1998.

[14] J. I. Munro and M. S. Paterson, “Selection and Sorting with Limited Storage,” *Theoretical Computer Science*, vol. 12, pp. 315–323, 1980.

[15] B. Patt-Shamir and D. Peleg, “Time-Space Tradeoffs for Set Operations,” *Theoretical Computer Science*, vol. 110, pp. 99–129, 1993.

[16] M. Karchmer, “Two Time-Space Tradeoffs for Element Distinctness,” *Theoretical Computer Science*, vol. 47, pp. 237–246, 1986.

Recent BRICS Report Series Publications

RS-98-10 **Jakob Pagter and Theis Rauhe.** ***Optimal Time-Space Trade-Offs for Sorting*****. May 1998. 12 pp.**

RS-98-9 **Zhe Yang.** ***Encoding Types in ML-like Languages (Preliminary Version)*****. April 1998. 32 pp.**

RS-98-8 **P. S. Thiagarajan and Jesper G. Henriksen.** ***Distributed Versions of Linear Time Temporal Logic: A Trace Perspective*****. April 1998. 49 pp. To appear in *3rd Advanced Course on Petri Nets*, ACPN '96 Proceedings, LNCS, 1998.**

RS-98-7 **Stephen Alstrup, Thore Husfeldt, and Theis Rauhe.** ***Marked Ancestor Problems (Preliminary Version)*****. April 1998.**

RS-98-6 **Kim Sunesen.** ***Further Results on Partial Order Equivalences on Infinite Systems*****. March 1998. 48 pp.**

RS-98-5 **Olivier Danvy.** ***Formatting Strings in ML*****. March 1998. 3 pp. This report is superseded by the later report BRICS RS-98-12.**

RS-98-4 **Mogens Nielsen and Thomas S. Hune.** ***Deciding Timed Bisimulation through Open Maps*****. February 1998.**

RS-98-3 **Christian N. S. Pedersen, Rune B. Lyngsø, and Jotun Hein.** ***Comparison of Coding DNA*****. January 1998. 20 pp. To appear in *Combinatorial Pattern Matching: 9th Annual Symposium*, CPM '98 Proceedings, LNCS, 1998.**

RS-98-2 **Olivier Danvy.** ***An Extensional Characterization of Lambda-Lifting and Lambda-Dropping*****. January 1998.**

RS-98-1 **Olivier Danvy.** ***A Simple Solution to Type Specialization (Extended Abstract)*****. January 1998. 7 pp.**

RS-97-53 **Olivier Danvy.** ***Online Type-Directed Partial Evaluation*****. December 1997. 31 pp. Extended version of an article to appear in *Third Fuji International Symposium on Functional and Logic Programming*, FLOPS '98 Proceedings (Kyoto, Japan, April 2–4, 1998), pages 271–295, World Scientific, 1998.**

RS-97-52 **Paola Quaglia.** ***On the Finitary Characterization of π-Congruences*****. December 1997. 59 pp.**

RS-97-51 **James McKinna and Robert Pollack.** ***Some Lambda Calculus and Type Theory Formalized*****. December 1997. 43 pp.**