2006:24

DOCTORAL THESIS

On Data Structures and Memory Models

Johan Karlsson

Luleå University of Technology
Department of Computer Science and Electrical Engineering

2006:24 | ISSN: 1402-1544 | ISRN: LTU-DT -- 06/24 -- SE

On Data Structures and Memory Models

by

Johan Karlsson

Department of Computer Science and Electrical Engineering
Luleå University of Technology
SE-971 87 Luleå, Sweden

May 2006

Supervisor
Andrej Brodnik, Ph.D.,
Luleå University of Technology, Sweden

Abstract

In this thesis we study the limitations of data structures and how they can be overcome through careful consideration of the used memory models. The *word RAM* model represents the memory as a finite set of registers consisting of a constant number of unique bits. From a hardware point of view it is not necessary to arrange the memory as in the word RAM memory model. However, it is the arrangement used in computer hardware today.

Registers may in fact share bits, or overlap their bytes, as in the *RAM with Byte Overlap (RAMBO)* model. This actually means that a physical bit can appear in several registers or even in several positions within one register. The RAMBO model of computation gives us a huge variety of memory topologies/models depending on the appearance sets of the bits.

We show that it is feasible to implement, in hardware, other memory models than the word RAM memory model. We do this by implementing a RAMBO variant on a memory board for the PC100 memory bus. When alternative memory models are allowed, it is possible to solve a number of problems more efficiently than under the word RAM memory model. We look at three priority queue related problems: the *Discrete Extended Priority Queue*, the *Time Queue*, and the *Prefix Sum* problems.

We side-step several lower bounds for the discrete extended priority queue problem and the prefix sum problem by allowing alternative memory models. We suggest two data structures and algorithms, which provide all the operations for the two problems in worst case constant time. It is not possible to achieve this time bound using the word RAM memory model.

We also suggest a data structure for the time queue problem. The algorithms run in expected constant time for the operations that delete the minimum element and worst case constant time for the other operations. The data structure can be maintained by several processes that share a part of the memory. Finally, we also show that it is possible to replace the ALU in a processor with memory while still keeping the ALUs functionality. Hence it is well worth and practical to consider alternative memory models, at least for special purpose processors.

Acknowledgments

First I would like to thank my supervisor Dr. Andrej Brodnik for all his encouragement, guidance, and friendship and for the many discussions over the last years. I would also like to thank Prof. Svante Carlsson and Dr. Jingsen Chen who first introduced the field of theoretical computer science to me and accepted me as a Ph.D. student.

Thanks also to Prof. J. Ian Munro who invited me to spend seven months at University of Waterloo under his guidance. The time there where a great source of inspiration to continue with my studies. It was also an opportunity, for my fiancée and me, to learn more about the life in Canada. Wendy, thank you for helping us with all practical things when we where there.

All colleges and friends, both at the university and outside, thank you for lighting up my days with 'fika', pool, frisbee golf, swimming, long chats about everything and nothing, coffee, single malts, ...

My family has always encouraged me to study, thanks. Finally, I would like to thank my fiancée Christina for all her love, support, and patience and my son Simon who always greets me with a smile on his face when I come home.

Contents

Included Papers

The following papers are included in this thesis:

- Paper 1

  Roni Leben, Marijan Miletić, Marjan Špegel, Andrej Trost, Andrej Brodnik and Johan Karlsson, *Design of a High Performance Memory Module on PC100*. In *Proceedings Electrotechnical and Computer Science Conference*, Slovenia, 1999.

- Paper 2

  Andrej Brodnik and Johan Karlsson. *Bitwise Operations under RAMBO*. Research report LTU - FR - - 06/12 - - SE, Luleå University of Technology, Luleå, Sweden, May 2006. `http://epubl.luth.se/1402-1528/2006/12/index.html`.

- Paper 3

  Andrej Brodnik and Johan Karlsson, *Multiprocess Time Queue*. In Peter Eades and Tadao Takaoka, editors, *Algorithms and Computation ISAAC 2001, 12th International Symposium*, volume 2223 of *Lecture Notes in Computer Science*, pages 599–609. Springer, December 2001.

- Paper 4

  Andrej Brodnik, Svante Carlsson, Michael L. Fredman, Johan Karlsson and J. Ian Munro, *Worst Case Constant Time Priority Queue*. In *Journal of System and Software*, 78(3):249-256, December 2005.

- Paper 5

  Andrej Brodnik, Johan Karlsson, J. Ian Munro and Andreas Nilsson, *An O(1) Solution to the Prefix Sum Problem on a Specialized Memory Architecture*. In *IFIP 19th World Computer Congress, TC1 4th International Conference on Theoretical Computer Science*, Springer, 2006.

To Christina and Simon

To be conscious that you are ignorant is a great step to knowledge.

Sir Benjamin Disraeli

Introduction

In this thesis we study the limitations of data structures and how they can be overcome through careful consideration of the used memory model. The main question is whether one should consider alternative memory models when designing data structures or only work with the already established ordinary RAM memory model.

The thesis consists of two parts, this introduction and five research papers. We start the introduction with a brief discussion about models of computation and memory models. This is followed by a discussion about data structures in general and data structures used to solve priority queue related problems in particular. The five papers are then summarized. We end the introduction with some conclusions and open research questions.

1 Models of Computation and Memory Models

A computer today consists of a CPU, memory, and an I/O subsystem. The CPU contains, at least, a control unit, a cache, and an *Arithmetic Logic Unit (ALU)*. The ALU is used to compute various functions. These functions usually include the bitwise operations (boolean operations and various shifts) and arithmetic operations (addition, subtraction, multiplication, and division). Such a computer is called a *von Neumann computer* [29] and is modeled as a *Random Access Machine (RAM)* [8, 26]. In the RAM model, the CPU can perform four types of operations: *I/O* operations which let the computer communicate with its surrounding, *read* and *write* operations which move data from and to the memory, *jump* operations (conditional or unconditional) which affect the flow of control, and *arithmetic and logic* operations.

Depending on which arithmetic and logic operations one allows, we talk about different variants of the RAM model. The basic RAM model allows addition and subtraction. If multiplication and division are allowed we denote

the model *MRAM* and if bitwise boolean operations are allowed we use an additional B in the name (for example *BRAM* and *MBRAM*) [26].

In the RAM models, the memory is modeled as an infinite set of registers, which store values. This is not feasible since it would require an infinite address size and consequently an infinite register size to address them. Further, if we allow multiplication and an unbounded word size the RAM model becomes as powerful as the parallel model *PRAM* [15, 17, 23]. Hence we restrict the word size and use w to denote it. Such RAMs are denoted *word RAM*s. A restricted word size implies a bounded universe and without loss of generality we let the universe be the integers from 0 to $M-1$ where $M \leq 2^w$. In this thesis we only consider word RAMs and we specify the set of operations on a case-by-case basis.

Moreover, in the RAM models, each register is represented by individual bits. In the word RAM models there are 2^w registers consisting of w bits and hence in total there are $2^w \cdot w$ bits in the memory. We refer to this memory model as the *RAM memory model*. From a hardware point of view it is not necessary to arrange the memory as in the RAM memory model. However, it is the arrangement used in computer hardware today.

Fredman and Saks suggested that registers may share bits, or in their words "words that overlap", and they named the model of computation: *RAM with Byte Overlap (RAMBO)* [11]. This actually means that a physical bit can appear in several registers or even in several positions within one register. Brodnik formalized the definition and introduced the notion of an *implicit* and an *explicit* RAMBO [4]. Each individual bit appears in the registers according to its appearance set. In the implicit RAMBO the appearance sets are static, and in the explicit RAMBO they are dynamic, that is, the appearance sets can be changed during execution of a program. The RAMBO model of computation gives us a huge variety of memory models depending on the appearance sets of the bits. Note that one part of the memory can have bytes that overlap while another does not.

2 Data Structures

Data structures are used by most computer programs to organize their data so that they can be processed efficiently. Several different data structures can often be used to solve one particular problem. For example, to store a set of elements one can use, among other data structures, a linked list, an array, or a hash table. Depending on the properties of the set and how it will be processed, different properties of the data structures are important.

We will look at a few data structures that can be used to solve priority queue related problems. These data structures can be used more efficiently if stored in alternative memory models.

2.1 Priority Queues and Related Data Structures

Priority queues and variants thereof are used to solve a variety of problems: sorting, event driven simulation, time-out managers, dictionaries, union-split-find, graph problems (for example shortest path and minimum spanning tree), closest neighbour searches, scheduling, etc. [7, 9, 18, 22, 25]. These problems appear in many applications, for example: routing of internet traffic (IP address lookup), 3-D games, spread sheet programs, and travel planning.

We look at two variants of priority queues; the *Discrete Extended Priority Queue*, and the *Time Queue*. The time queue is targeted at solving the problem of supporting a time-out manager while the discrete extended priority queue is more general and can be used in solutions to all of the above stated problems. Further, we also look at the related *Prefix Sum* problem introduced by Fredman [10].

The discrete extended priority queue is a rather general extension of the priority queue, which in its most basic variant only supports the operations `insert`, `min`, and `deleteMin`.

Definition 1 *The* Discrete Extended Priority Queue problem *is to maintain a set $\mathcal{N}$ of size N with elements drawn from an ordered bounded universe $\mathcal{M} = [0..M-1]$ and support the following operations:*

`insert`(e)	$\mathcal{N} := \mathcal{N} \cup \{e\}$,
`delete`(e)	$\mathcal{N} := \mathcal{N} \backslash \{e\}$,
`predecessor`(e)	*return the largest element $f \in \mathcal{N}$ such that $f < e$,*
`successor`(e)	*return the smallest element $f \in \mathcal{N}$ such that $f > e$,*
`member`(e)	*return whether $e \in \mathcal{N}$,*
`min`()	*return the smallest element of $\mathcal{N}$,*
`max`()	*return the largest element of $\mathcal{N}$,*
`deleteMin`()	*delete the smallest element of $\mathcal{N}$,*
`deleteMax`()	*delete the largest element of $\mathcal{N}$.*

Note that using the first four operations the last five can be supported. The `min`() operation is a special case of `successor`(e) and `max`() of `predecessor`(e) while `deleteMin`() and `deleteMax`() are special cases of `delete`(e). The `member`(e) operation is equivalent to checking if `successor(predecessor(`e`))` is equal to e.

Under the pointer machine model (cf. [21]), Mehlhorn et al. [18] proved a lower bound of $\Omega(\lg \lg M)$ for the discrete extended priority queue problem. The *stratified tree* by van Emde Boas et al. provides a matching upper bound [25]. Beame and Fich later gave a lower bound for the `predecessor` query under the communication game model (cf. [19, 30]). Under the cell probe model (cf. [32]) the lower bound becomes $\Omega(\min((\lg \lg M / \lg \lg \lg M), \sqrt{\lg N / \lg \lg N}))$ when restricting the memory usage to $N^{O(1)}$ words, which also applies to the RAM models [3]. They also gave a matching upper bound, in the RAM model and hence the communication game and the cell probe models, for the static version of the problem (without `insert, delete, deleteMin` and `deleteMax`). Andersson and Thorup [2] gave a data structure and an algorithm with $O(\sqrt{\lg N / \lg \lg N})$

worst case time for the dynamic version. For the static general case (arbitrary number of dimensions), Brodnik and Munro [6] gave a data structure with $O(1)$ worst case time for any N but using $O(M)$ bits of space. On the other hand, Ajtai et al. [1] presented a solution using only $O(N)$ words when $N = O(M^{1/p})$.

The next problem we look at is to support a time-out manager, which associates items with time stamps. The manager also has a current time, which is increased. If it is increased to the time stamp of some item, this item is considered to have timed out and should be handled specially. Usually the current time is increased according to a real time clock. The operations of the time queue are chosen to support the time-out manager:

Definition 2 *The* Time Queue problem *is the problem of maintaining a set $\mathcal{N}$ of elements and to support the following operations (where t_0 is the time of the min element and C is the maximum duration of any element):*

`insert(`e`, `t`)`	*iff $t_0 < t \leq t_0 + C$ then let $\mathcal{N} := \mathcal{N} \cup \{e\}$ with $e.t = t$,*
`delete(`e`)`	*let $\mathcal{N} := \mathcal{N} \backslash \{e\}$,*
`min():`e	*return the min element,*
`deleteMin()`	*delete the min element,*
`update(`e`,`t`)`	*iff $t_0 < t \leq t_0 + C$ then change the time $e.t$ of e to t,*
`delLessThan(`t`, `$\mathcal{F}$`)`	*delete all elements e with time $e.t$ less than t and call the function '$\mathcal{F}$' for each of the deleted elements.*

Any solution to the discrete extended priority queue problem can be use to solve the time queue problem. In addition, Brown [7] suggested the use of a data structure called *Calendar Queue* and Varghese and Lauck [28] suggested a very similar solution called *Hashed and Hierarchical Timing Wheel*.

We consider a variation of the time queue problem where two processes must be able to use the data structure (cf. [12, 17, 23, 24]). One process performs time critical tasks and must be guaranteed constant time operations. This process, however, only needs to perform the `min` operation and a restricted `update` operation. The other process performs all operations and may spend more time on each operation.

The last problem we study is the prefix sum problem, which was introduced by Fredman [10]:

Definition 3 *The* Prefix Sum problem *is to maintain an array $\mathcal{A}$ of size N and to support the following operations (where $0 \leq j < N$):*

`update(`j`, `Δ`)`	$\mathcal{A}(j) := \mathcal{A}(j) + \Delta$,
`retrieve(`j`)`	*return* $\sum_{i=0}^{j} \mathcal{A}(i)$.

A solution to the prefix sum problem also gives a solution to the Rank problem;

Definition 4 *The* Rank problem *is to maintain a set $\mathcal{N}$ of size N with elements drawn from an ordered bounded universe $\mathcal{M} = [0..M-1]$ and support the following operations:*

`insert(`e`)`	$\mathcal{N} := \mathcal{N} \cup \{e\}$.
`delete(`e`)`	$\mathcal{N} := \mathcal{N} \backslash \{e\}$,
`rank(`e`)`	*return the number of elements $f \in \mathcal{N}$ where $f \leq e$.*

In an array `A` of size M we let the value of `A[i]` be 1 if $i \in \mathcal{N}$ and 0 otherwise. Now the prefix sum for each element in `A` is equal to the rank of that element in the set $\mathcal{N}$.

The inverse problem to the Rank problem is the Select problem:

Definition 5 *The* Select problem *is to maintain a set $\mathcal{N}$ of size N with elements drawn from an ordered bounded universe $\mathcal{M} = [0..M-1]$ and support the following operations:*

`insert(`e`)` $\mathcal{N} := \mathcal{N} \cup \{e\}$,
`delete(`e`)` $\mathcal{N} := \mathcal{N} \backslash \{e\}$,
`select(`i`)` *return the element $e \in \mathcal{N}$ with rank i.*

These problems has been referred to as the *Searchable Partial Sums problem* [16, 20] when combined. Given two data structures, one used to solve the rank problem and one used to solve the select problem, we can solve the discrete extended priority queue problem. We insert and delete the given element in both data structures. To support `successor(`e`)` we first find the rank i of e and then select the element with rank $i+1$. To support `predecessor(`e`)` we first find the rank i of $e-1$ and then select the element with rank i. As noted above the last five operations of the discrete extended priority queue problem are special cases of these four operations.

Several lower bounds has been shown for the prefix sum problem. Fredman showed a $\Omega(\lg N)$ algebraic complexity lower bound and a $\Omega(\lg N / \lg\lg N)$ information-theoretic lower bound [10]. Yao [31] has shown that $\Omega(\lg N / \lg\lg N)$ is an inherent lower bound under the semi-group model of computation and this was improved by Hampapuram and Fredman to $\Omega(\lg N)$ [14].

3 Summary of Included Papers

We now continue with a summary of the five included papers:

- Paper 1

  Design of a High Performance Memory Module on PC100 presents the hardware design of the special memory used in the *Worst Case Constant Time Priority Queue* paper (Paper 4). In this memory the words corresponds to the leaves of a balanced binary tree. Each node of the tree contains a flag bit and each such word contains the flags along the root to leaf path, so, for example, the flag at the root is in all of these words. The specific architecture was called *Yggdrasil* after the giant ash tree linking the worlds in Norse mythology. This particular design and hardware implementation is for the PC100 memory bus used in most Pentium-2 computers.

  This thesis author's contribution to the results in this paper is the software used for the verification of the hardware implementation: a device driver, a user library, and test programs.

- Paper 2

  In the paper *Bitwise Operations under RAMBO* we study the problem of computing w-bit bitwise operations using only $O(1)$ memory probes. When using the RAM memory model there exists a $\Omega(2^w)$ space lower bound while this space bound goes down to $O(w)$ bits if we consider alternative memory models. We present algorithms that use four different memory models to perform bitwise boolean operations and shift operations.

- Paper 3

  In the paper *Multiprocess Time Queue* we implement a time queue for elements with a bounded maximum duration C. This particular time queue supports a time-out manager controlled by two processes. The first process performs all the operations of the time queue while the other only performs `min` and a restricted `update`.

  We use a data structure similar to the Calendar Queue by Brown [7] and consider a memory model where two or more processes can share a part of the memory under mutual exclusion. The operations `deleteMin` and `delLessThan` are supported in expected constant running time under conditions that were met by our application. The other operations are supported in constant worst case running time. The space needed is proportional to the square root of the maximum duration of any element, and the number of elements, that is, $O(\sqrt{C} + N)$.

- Paper 4

  The paper *Worst Case Constant Time Priority Queue* extends work done by Brodnik [4]. It presents a solution to the discrete extended priority queue problem using a data structure called *Split Tagged Tree*. A part of the data structure is stored using the Yggdrasil memory.

  The solution provides all the operations in worst case constant running time using $2M + O(\lg M)$ bits of ordinary memory and M bits of the special Yggdrasil memory. If only support for either `min`, `deleteMin` and `successor` or `max`, `deleteMax` and `predecessor` is needed the amount of ordinary memory used is reduced to $M + O(\lg M)$ bits. The amount of ordinary memory can be reduced even further, to $O(N \lg M)$ bits, at the expense of the worst case time. If reduced, the running time of the update operations (`insert, delete, deleteMin` and `deleteMax`) is expected constant time instead of worst case constant time.

- Paper 5

  In *An O(1) Solution to the Prefix Sum Problem on a Specialized Memory Architecture* we study the prefix sum problem. We show that it is possible to perform both update and retrieval in $O(1)$ time simultaneously under a memory model similar to the Yggdrasil memory model. In this variant we allow each node on the path to store several bits instead of only one bit. To

achieve the $O(1)$ time we must allow $O(\sqrt{M^{(\lg N)}} \cdot \lg M)$ bits of ordinary memory and $O(N \lg M)$ bits of special memory to be used. This is a huge amount of ordinary memory and if we restrict the space requirement to be sub exponential in both N and M ($O(\lg M)$ bits of ordinary memory and $O(N \lg M)$ bits of special memory) we need to used $O(\lg \lg N)$ time. This is still an improvement over the lower bounds that we side-step.

4 Conclusion and Future Research

We showed that it is feasible to implement, in hardware, other memory models than the RAM memory model. We did this by implementing the Yggdrasil variant on a memory board for the PC100 memory bus. Further, we showed that, when we allow alternative memory models, it is possible solve a number of problems more efficiently than under the standard RAM memory model. We looked at two variants of priority queues; the discrete extended priority queue and the time queue; and we studied the prefix sum problem. We provided three data structures, which yield efficient solutions to these three problems. We also showed that it is possible to replace the ALU in a processor with memory while keeping the functionality provided by the ALU.

We conclude that it is possible to enhance data structures by allowing the use of alternative memory models. However, the alternative memory models seem to be different for different problems. They are therefore unlikely be an option in standard computers, which are designed to be general. Thus, we believe that it is well worth to consider alternative memory models, at least for special purpose processors.

Obviously, there is more research needed and there are a number of open research questions. We highlight a few here:

- Searching in Higher Dimensions.

  Is it possible to use the Yggdrasil memory model (or some other novel memory model) to improve the time complexity of solutions to search problems (for example Closest Neighbour Searches) in finite d-dimensional space?

- Non Priority Queue Related Problems

  What other problems, beside priority queue related problems, can be solved more efficiently when alternative memory models are allowed?

  In this thesis we have touched on this briefly in Paper 2 where we studied the bitwise operations and how to compute them under the RAMBO model using only memory probes.

- Decrease Space Requirement.

  – Is it possible to decrease the space requirement for the discrete extended priority queue problem to be $o(M)$ while still retaining the

worst case $O(1)$ time? Brodnik and Iacono have recently started to look at this [5].

– Further, can we decrease the space requirement $O((N^{O(1)} + M^{O(1)}) \cdot \lg M)$ bits for to the prefix sum problem while still retaining a $O(1)$ time solution?

Bibliography

[1] M Ajtai, M Fredman, and J Komlós. Hash functions for priority queues. *Inf. Control*, 63(3):217–225, 1986.

[2] Arne Andersson and Mikkel Thorup. Tight(er) worst-case bounds on dynamic searching and priority queues. In *Proceedings of the 32nd Annual ACM Symposium on Theory of Computing*, pages 335–342. ACM Press, May 21–23 2000.

[3] P. Beame and F. E. Fich. Optimal bounds for the predecessor problem and related problems. *Journal of Computer and System Sciences*, 65(1):38–72, 2002.

[4] Andrej Brodnik. *Searching in Constant Time and Minimum Space (*MINIMÆ RES MAGNI MOMENTI SUNT*)*. PhD thesis, University of Waterloo, Waterloo, Ontario, Canada, 1995. (Also published as technical report CS-95-41.).

[5] Andrej Brodnik and John Iacono. Dynamic predecessor queries. Unpublished manuscript, 2006.

[6] Andrej Brodnik and J. Ian Munro. Neighbours on a grid. In R. Karlsson and A. Lingas, editors, *SWAT '96, 5th Scandinavian Workshop on Algorithm Theory*, volume 1097 of *Lecture Notes in Computer Science*, pages 307–320. Springer, 3–5July 1996.

[7] Randy Brown. Calendar queues: A fast $O(1)$ priority queue implementation for the simulation event set problem. *Communications of the ACM*, 31(10):1220–1227, October 1988.

[8] Stephen A. Cook and Robert A. Reckhow. Time bounded random access machines. *Journal of Computer and System Sciences*, 7(4):354–375, 1973.

[9] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press and McGraw-Hill Book Company, 1990.

[10] Michael L. Fredman. The complexity of maintaining an array and computing its partial sums. *Journal of the ACM*, 29(1):250–260, January 1982.

[11] Michael L. Fredman and Michael E. Saks. The cell probe complexity of dynamic data structures. In *Proceedings of the 21st Annual ACM Symposium on Theory of Computing*, pages 345–354. ACM Press, May 14–17 1989.

[12] Phillip B. Gibbons, Yossi Matias, and Vijaya Ramachandran. Can a shared-memory model serve as a bridging model for parallel computation? *Theory of Computing Systems*, 32(3):327–359, 1999.

[13] Michael D. Godfrey. Introduction to "the first draft report on the EDVAC" by John von Neumann. *IEEE Annals of the History of Computing*, 15(4):27–75, 1993.

[14] Haripriyan Hampapuram and Michael L. Fredman. Optimal biweighted binary trees and the complexity of maintaining partial sums. *SIAM Journal on Computing*, 28(1):1–9, 1998.

[15] J. Hartmanis and J. Simon. On the power of multiplication in random access machines. In *15th Annual Symposium on Switching and Automata Theory*, pages 13–23, 1974.

[16] Wing-Kai Hon, Kunihiko Sadakane, and Wing-Kin Sung. Succinct data structure for searchable partial sums. In Toshihide Ibaraki, Naoki Katoh, and Hirotaka Ono, editors, *Algorithms and Computation – ISAAC 2003, 14th International Symposium*, volume 2906 of *Lecture Notes in Computer Science*, pages 505–516. Springer, December 2003.

[17] Richard M. Karp and Vijaya Ramachandran. Parallel algorithms for shared-memeory machines. In van Leeuwen [27], chapter 17, pages 869–941.

[18] Kurt Mehlhorn, Stefan Näher, and Helmut Alt. A lower bound on the complexity of the union-split-find problem. *SIAM Journal on Computing*, 17(6):1093–1102, 1988.

[19] Peter Bro Miltersen. Lower bounds for Union-Split-Find related problems on random access machines. In *Proceedings of the 26th Annual ACM Symposium on Theory of Computing*, pages 625–634. ACM Press, 23–25 May 1994.

[20] Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct dynamic data structure. In *Algorithms and Data Structures, 7th International Workshop*, volume 2125 of *Lecture Notes in Computer Science*, pages 426–437. Springer, 8–10 August 2001.

[21] Arnold Schönhage. Storage modifications machines. *SIAM Journal on Computing*, 9(3):490–508, August 1980.

[22] J. D. Ullman, A. V. Aho, and J. E. Hopcroft. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, 1974.

[23] L. G. Valiant. General purpose parallel architectures. In van Leeuwen [27], chapter 18, pages 943–971.

[24] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 38(8):103–111, August 1990.

[25] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977.

[26] Peter van Emde Boas. Machine models and simulations. In van Leeuwen [27], chapter 1, pages 3–66.

[27] Jan van Leeuwen, editor. *Handbook of Theoretical Computer Science*, volume A: Algorithms and Complexity. Elsevier/MIT Press, Amsterdam, 1990.

[28] George Varghese and Anthony Lauck. Hashed and hierarchical timing wheels: Efficient data structure for implementing a timer facility. *IEEE/ACM Transaction on Networking*, 5(6):824–834, December 1997.

[29] John von Neumann. First draft of a report on the EDVAC. Technical report, Moore School of Electrical Engineering, University of Pennsylvania, 30 June 1945. Reprinted in [13].

[30] A. Yao. Some complexity questions related to distributive computing. In *Proceedings of the 11th Annual ACM Symposium on Theory of Computing*, pages 209–213, 1979.

[31] Andrew C. Yao. On the complexity of maintaining partial sums. *SIAM Journal on Computing*, 14(2):277–288, May 1985.

[32] Andrew Chi-Chih Yao. Should tables be sorted? *Journal of the ACM*, 28(3):614–628, July 1981.

Papers

Paper 1

Design of a High Performance Memory Module on PC100

Rivers know this: there is no hurry. We shall get there some day.

Winnie the Pooh

Reformatted version of paper published as

Roni Leben, Marijan Miletić, Marjan Špegel, Andrej Trost, Andrej Brodnik and Johan Karlsson, *Design of a High Performance Memory Module on PC100.* In *Proceedings Electrotechnical and Computer Science Conference*, Slovenia, 1999.

Design of a High Performance Memory Module on PC100

Roni Leben [*] Marijan Miletić [†] Marjan Špegel[*]
Andrej Trost [‡] Andrej Brodnik [§¶] Johan Karlsson[¶]

Abstract

In this contribution we present the design of a special-purpose memory board for the PC100 memory bus. The data storage on the module is placed in dynamic RAM chips and in an FPGA chip to provide for faster data access needed in increasingly popular time-critical PC applications such as real-time simulation of systems and time queue handling in general. The complexity of the PCB design for the module is one of the most demanding designs ever accomplished in Slovenia. Actually, there are just a few companies in Europe at the moment capable to design and produce the PC100-compatible boards.

The supporting software was developed under the operating system FreeBSD *ver. 3.2 as a user library and a kernel module.*

1 Introduction and Motivation

Although many computer architectures have been proposed and implemented over the half-century of modern computing, the predominant architecutre is still the von Neumann machine consisting of the processor (e.g. CPU), the memory (e.g. RAM), and the I/O units (e.g. VDU).

The communication between these units is carried out through well defined interfaces – buses. In contemporary off-the-shelf PCs one can find a variety of buses, and the most common among them are the ISA, EISA, and the PCI bus. These buses are designed to connect CPU with I/O devices (e.g. disk-, video-, network-controllers). Though the speed of communication between the CPU and I/O devices is ever increasing, it is typically still significantly lower than the speed of communication between the CPU and the main memory. In fact, because of increasingly higher requirements for the speed of the CPU-to-memory

[*] *Jožef Stefan* Institute, Ljubljana, Slovenia
[†] *Artinian*, Ljubljana, Slovenia
[‡] University of Ljubljana, Faculty of Electrical Engineering, Ljubljana, Slovenia
[§] Institute of Mathematics Physics and Mechanics, Ljubljana, Slovenia
[¶] Luleå University of Technology, Luleå, Sweden

communication, the computer memory is typically placed on a completely different and separate bus. In the beginning of 1998, a new standard for this bus was developed and proposed by Intel called the PC100 bus, permittting speeds up to 6.4 Gbit/sec. In fact, most of creativity in computer design is nowadays spent on the interconnection between the memory and the processor, and the data throughput will ultimately depend on a number of technological and architectural decisions.

In this paper we present the design of an advanced memory module for the PC100 bus. The logical design of the module is based on a special memory topology called *Yggdrasil* described and tested in [4]. Briefly, the memory can be described as a binary tree where each internal node stores one bit of information. The addressable memory registers are actually at the leaves of the tree, whereas their content is stored on the path from this leaf to the root. The design of the memory is under patent protection by *Priqueue AB* from Sweden. Karlsson in his master thesis used two different implementations of the memory: the first one was software emulation of the tree in regular memory, whereas the second one was an ISA-bus hardware implementation (*YGG-I*), where the height of the *YGG-I* was 16 bits.

In the new design called *YGG-II*, we extended the height of *YGG-I* to 25 bits by splitting the address tree into two parts: the first part covering top 13 bits that were placed in an FPGA device, and the second part covering lower 12 bits that were placed in regular memory ICs. In the following section we describe the overall architecture of the *YGG-II*. This is followed by the description of the PCB design, and the FPGA design. We conclude the paper with the discussion of likely further work on the new module.

2 *YGG-II* Architecture

Memory architecture of choice for the design of new PCs at the beginning of 1998 was SDRAM for the 72 bits wide PC100 bus. This is dynamic RAM with synchronous interface and clock running at 100 MHz. Internal DRAM access time of minimum 50 nsec is subdivided into several 10 nsec cycles permitting functions overlap and data bursts. Classic RAS and CAS signals for a multiplexed address and data strobe have been assigned new functions according to the simplified state diagram as shown in Fig. 6.27 in [5]. The basic timing difference is shown in Fig. 6.3 in [5].

The YGG tree-like configuration, shown in Fig. 1.1, requires uniquely decreasing-capacity RAMs for each data bit. Due to the total availability of 24 address and 64 data lines, we can build two separate 25 bits high data trees. Seven MSBs are fixed to the logical 1 level by resistor's pull-up network. Active DQ24/56 flip-flops are enabled by the board-select lines, only without any addressing. The DQ23/55 data are stored in 2-bit memory controlled by the MSB A11 line. The DQ22/54 needs 4-bit memories addressed by the A10 and A11 lines at RAS time. The LSB DQ0/32 uses two SDRAM bits with all address lines active for 16MB access. Data burst length was limited to 1 in order to

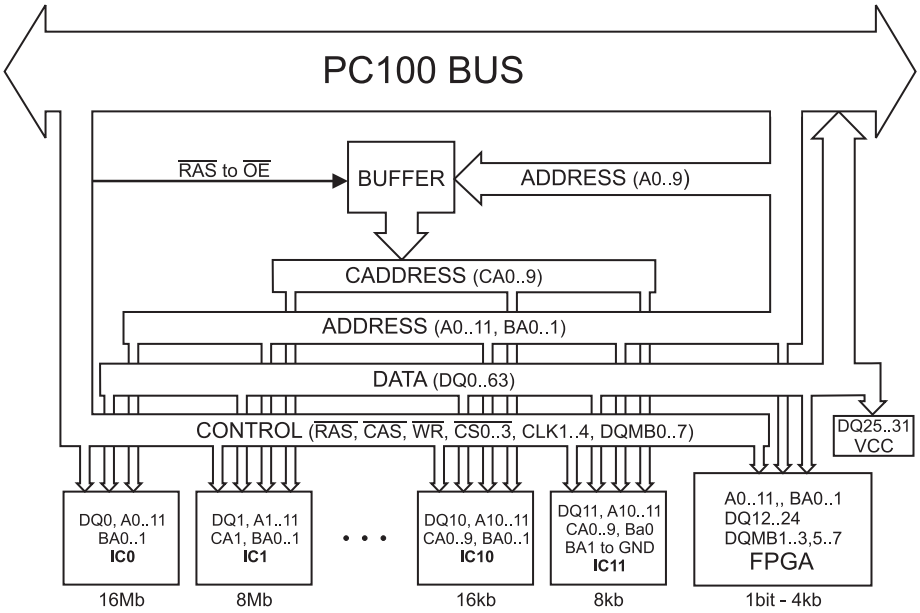


Figure 1.1: Architecture of the YGG-II board.

accommodate the tree-type access rather then the forecasted long sequential order. The lower 12 data bits were realized with a 16M×4 SDRAM IC using only two outputs. Column addresses were sequentially reduced with RAS controlled 10-line buffer. The last IC has the BA1 bank select line grounded rather than connected to A0 as expected. That was discovered to be the way the linear addresses are generated on PC100.

The MSB SDRAM with 2×13 blocks from 4k bits downward were all squeezed into a single FPGA from the Xilinx XC4036XLA series first produced in January 1999. Only the top 12 addresses strobed by RAS are utilized, and only three of all PC100 commands need support: `ACTV`, `READ` and `WRITE`. The `ACTV` command sets FPGA RAM addresses at RAS time. Two clocks later comes CAS with WR signal deciding the remaining data transfer. The `WRITE` data are immediately provided whereas the `READ` output should appear after programmed time of 3 clock cycles. The CS0 and the CS2 signals select 16 bit words whereas the four DQMB signals enable byte operations. The CKE signal can freeze the clock, thereby extending all cycles. All of the input signals are first latched within the fast FPGA input pairs of flip-flops running at 100 MHz with less than 1.5 nsec overall clock delay. Combinatorial logic is hand-placed close to the input signals in order to achieve less than 10 nsec delay. Larger memory blocks made from 32-bit internal RAM ICs were also constrained in their relative placement within the FPGA device in order to achieve the required 100 MHz operation.

3 Printed Circuit Board Design

The aforementioned *YGG-II* architecture was implemented on a six-layer 168-pin DIMM SDRAM format printed circuit board (PCB) module. Since our goal was to design and prototype a PC100-compliant memory module, the PCB complies with JEDEC ([3]) and Intel ([2]) specifications regarding DIMM memory modules. The printed circuit board has four signal layers and two power planes (V_{CC} and V_{SS}). The JEDEC standard ([3]) defines PCB width of 138.5 mm and maximum height of 38.1 mm, of which 4.0 mm are reserved for edge connector pins. Since the Xilinx XC4036XLA FPGA has a footprint of about 32 by 32 mm, there was very little room left for the connections to its pins near the edge connector pins.

But the central PCB design problem was the routing of the signals. The Intel specification ([2]) defines routing topologies, trace lengths, trace widths, clearance between traces, pads, vias, etc., maximum number of vias and layers to be used for each of about two hundred signals. On ordinary DIMM boards, the address and control lines are routed horizontally on the upper part of the PCB (if we define the lower part to be next to the edge connector) and connected to the edge connector vertically on the center of the board. The data lines are routed in vertical direction from edge connector pins via serial resistors to SDRAM IC's. The hardest restrictions, however, are placed on the four clock signals, which are routed diagonally in the two internal layers between the power planes to minimize their radiation. The edge connector pinout is designed with these restrictions in mind.

The routing topology of our design is quite different from the one mentioned above. In addition to the fourteen normal address lines, there are ten more column address lines, generated by the buffer IC. The data lines also are not routed only vertically any more; a lot of them are connected to the XC4036XLA FPGA placed on the center of the board, since the MSB bits of the tree are stored in it. Many address and control signals are also connected to the FPGA. Because of these restrictions and the deviations from the Intel's topology – as well as because of additional lines to be routed – the automatic routing performed by *Specctra* auto-router software package was not an easy task, and the resulting PCB fist had about 750 vias which were subsequently reduced to about 650 by hand. Some trace length restrictions also had to be loosened, because it was physically impossible to fully satisfy them. Furthermore, it did not make sense to follow the design restrictions very tightly because the internal delays of the FPGA were also not known at the time of the PCB design; namely, the FPGA internal routing was completely finished only recently. Finally, there were also many changes made to the schematic design during the design of the PCB, the so-called Engineering Change Orders (ECOs) and – since this is the first prototype – some changes were even made on the finished PCB prototypes. Because of the high design demands – as well as because of its numerous and extremely small vias, this PCB is probably also one of the most demanding designs ever accomplished in Slovenia.

4 FPGA Design

The upper part of the memory tree – as discussed before – is implemented in a Xilinx FPGA *XC4036XLA* consisting of 1296 Configurable Logic Blocks (CLBs). Each CLB can be used for combinatorial and registered logic or as a 32×1-bit Select-RAM static memory ([1]). The external signals are routed through I/O Blocks (IOBs). The IOBs provide input and output latches and flip-flops used to synchronize the external signals with an internal clock. Special global-clock buffers and routing resources are used for low-skew clock distribution to each CLB and IOB.

The circuit consists of 1-bit memory blocks from 4k×1-bit to a single memory cell, of address and data latches, and of read/write control logic shown in Fig. 1.2. The control logic is used for decoding SDRAM instructions provided

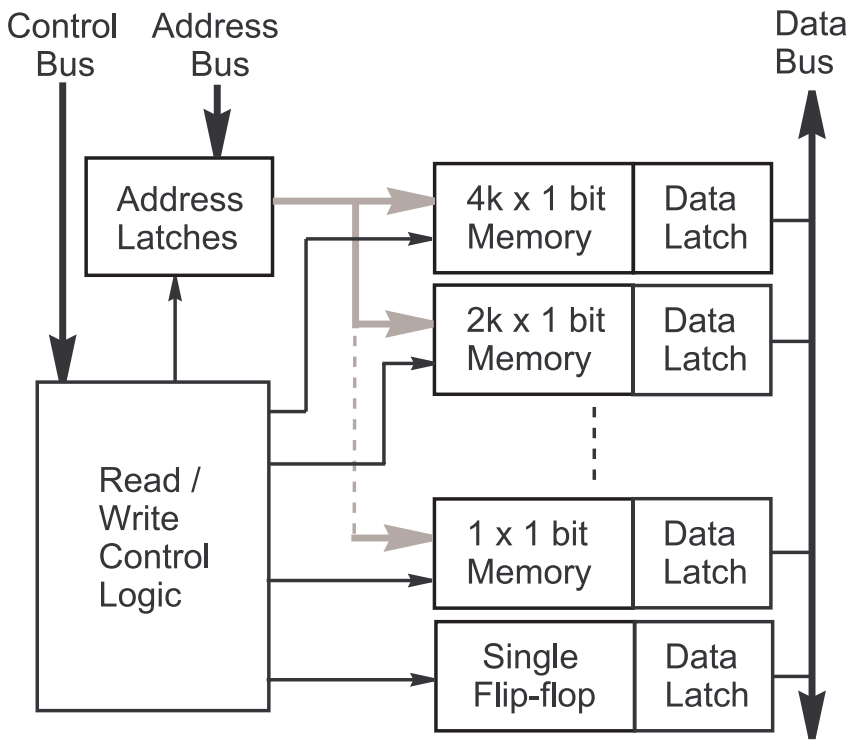


Figure 1.2: The FPGA circuit architecture.

through the control bus signals. The implemented instructions are `ACTV` (RAS cycle), `READ` and `WRITE`. Since the upper memory tree uses only row-address lines, all the internal address signals are latched during `ACTV` command. The

timing diagram of the `READ` and `WRITE` instructions was determined from the SDRAM specifications. The detail structure of one memory block is presented in Fig. 1.3. During the execution of the `READ` instruction, the data is trans-

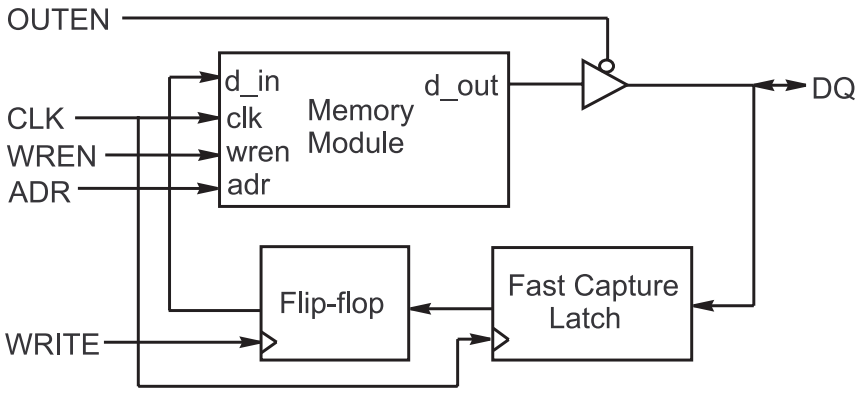


Figure 1.3: A detail of one memory block.

ferred from the output of the memory module through the 3-state output buffer in the third cycle after the instruction. The data and the `WRITE` instruction are inserted simultaneously in the write cycle and are latched in the fast capture latches in the IOBs. The fast capture latches are used to meet SDRAM set-up and hold-time requirements as described in [6]. After decoding of the `WRITE` instruction, each data bit is captured in a single flip-flop for subsequent transfer to the memory module during the following clock cycles.

The circuit was designed schematically with the *Xilinx Foundation* package, which provides many already prepared logic blocks and a hardware macro generator. The macro generator is useful for quick composition of different memory modules but is unfortunately limited to the memory depth of 256 locations only. Larger memory modules were designed hierarchically from the basic 32×1-bit synchronous Select-RAM. We extensively used relationally placed macros for manual mapping of Select-RAM cells and decode logic in 2 column wide CLB matrices. Select-RAM data outputs were combined using 3-state buffers, which are faster than combinatorial multiplexers. The critical part of the design is the read/write control logic required to operate at 100MHz clock frequency. The logic was manually mapped and some parts were placed to absolute CLB locations inside FPGA. The best locations for the control logic were derived experimentally after many automatic placement and routing iterations. The delay-driven placement and routing was used, based on timing specifications for the critical paths in the design. The complete design took several months due to many design iterations, hand mapping and placements, as well as due to very long compilation times permitting only one or two routing iterations per day.

5 Conclusions

In this contribution we presented an architecture and design of a highly advanced memory board for the PC100 memory bus. The board was designed, produced and fully tested in Slovenia. The tests show that the board performs according to the specifications of this bus and hence allows accesses without wait-states, which in other words means, that it supports the same access speed as ordinary DRAM boards.

The successful development and prototyping of our memory module proves two things: first, the feasibility of the tree topology of RAM and consecutively its usefulness in, for example, real-time applications which involve efficient handling of time-queues. Second, our design team has clearly demonstrated sufficient expertise and resources to create this sophisticated piece of electronic hardware and, therefore, seems well qualified for competing on international markets without hesitation.

Acknowledgments

We would like to thank *Jerovšek Computers* for a computer that was used for testing the board (cf. Fig. 1.4) in a real environment.

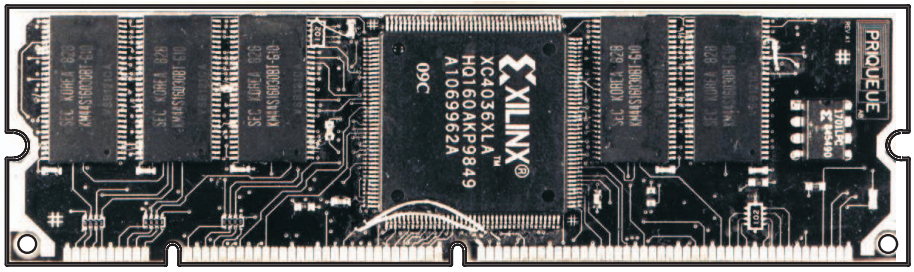


Figure 1.4: Picture of a working prototype.

References

[1] Xilinx Inc. XC4000XLA/XV field programmable gate arrays, product specification, May 1999.

[2] INTEL. Pc sdram unbuffered dimm specification, February 1998. Revision 1.0.

[3] JEDEC. Mo-161 multiple keyways dimm standard, August 1998. Issue C.

[4] J. Karlsson. Evaluation of different solutions to the left-right-neighbour and priority queue problems, June 1998.

[5] B. Price. *High Performance Memories.* John Wiley & Sons, England, 1996.

[6] B. Taylor. XC4000XL FPGAs interface to SDRAMs at 100MHz, Xcell, Q2, 1998.

Paper 2

Bitwise Operations under RAMBO

A little Consideration, a little Thought for Others, makes all the difference.
Winnie the Pooh

Research report

Andrej Brodnik and Johan Karlsson. *Bitwise Operations under RAMBO*. Research report LTU - FR - - 06/12 - - SE, Luleå University of Technology, Luleå, Sweden, May 2006. `http://epubl.luth.se/1402-1528/2006/12/index.html`.

Bitwise Operations under RAMBO

Andrej Brodnik *†‡
andrej.brodnik@upr.si

Johan Karlsson*
johan.karlsson@csee.ltu.se

Abstract

In this paper we study the problem of computing w-bit bitwise operations using only $O(1)$ memory probes. We show that under the RAM model there exists a $\Omega(2^w)$ space lower bound while under the RAMBO model this space bound goes down to $O(w)$ bits. We present algorithms that use four different RAMBO memory topologies to perform bitwise boolean operations and shift operations.

1 Introduction

A computer today consists of a CPU, memory, and an I/O subsystem. The CPU contains, at least, a control unit, a cache, and an *Arithmetic Logic Unit (ALU)*. The ALU is used to, given operands and an operation code (opCode), compute various functions. These functions include the bitwise operations (boolean operations and various shifts) and arithmetic operations (addition, subtraction, multiplication, and division). Although the arithmetic operations are considered atomic operations, they still consist of several micro-steps which, in turn, usually are bitwise operations (for more details see any text book on computer architectures, e.g. [10]).

In theoretical computer science we model such a computer as a RAM (cf. [5, 18]) where the processor is capable of performing functions from some predefined finite subset of NC^1 in $O(1)$ time. The class NC^k of functions is defined by:

Definition 2.1 *[11, p. 135] For each $k \geq 0$ the class NC^k consists of the search problems solvable by log-space uniform classes of boolean circuits having polynomial size and depth $O(\log^k w)$*

NC^1 involves circuits of logarithmic depth that computes, at each step, various bitwise operations (compare micro-steps above). Indeed, these operations could be computed using table lookup – i.e. the circuit elements would be replaced by memory probes only. However, the size of such table would become prohibitively large. In this paper we use a variant of the RAM model called

*Luleå University of Technology, Sweden
†University of Primorska, Slovenia
‡Institute of Mathematics, Physics, and Mechanics, Slovenia

RAMBO (*RAM with bytes/bits overlapping*) and show that all bitwise operations can be performed using writes and reads of memory only, while the space requirement remains at a bearable $O(w)$ bits.

As said, we can divide the bitwise operations into shifts and boolean operations. Further, the shifts come in five flavors: left and right shift, left and right rotation, and arithmetic right shift. In Sect. 2 we show how all these operations can be performed using $4w$ bits under the RAMBO model – a clear gain over the straightforward approach which uses $O(2^w w^2)$ bits under the RAM model. On the other hand, to implement the boolean operations using table lookup, although they all can be computed using NAND (or NOR) only, requires at least $\Omega(2^w)$ bits (see Corollary 2.1) of memory under the RAM model and a straightforward approach uses $O(2^{2w} w)$ bits. This is much more than $O(w)$ bits needed under the RAMBO model as will be seen in Sect. 3. In total, we reduce the size of the data structure (i.e. program) from $O(2^{2w} w)$ under the RAM model to $O(w)$ under the RAMBO model without increasing the time complexity.

Finally in Sect. 4 we show how the presented operations can be combined into a function to perform addition of two words in $O(\lg w)$ steps and $O(w)$ space under the RAMBO model.

1.1 Preliminaries

The RAM model models a computer as a CPU and an infinite set of memory registers [5, 18]. There are several variants of the RAM model (e.g., MBRAM [18], and AC^0 RAM [2]) and they differ in which operations the CPU can perform in unit time. Also they differ in whether the memory registers are of bounded size or not. In this paper we consider a variant of the RAM model where the only operations are read and write from/to memory with registers of bounded size w (cf., cell probe model [20]). We refer to this variant as the *Read-Write RAM* model.

A Read-Write RAM corresponds to a Turing Machine (TM) [18] with an alphabet of size 2^w ($[0, 1, \ldots, 2^w - 1]$), and the possibility to access a random tape location in unit time. The state transition function (the program) of the TM corresponds to precomputed tables in the RAM. The computation of some functions requires a large lookup table:

Theorem 2.1 *The computation of an onto function $f : \mathcal{A} \to \mathcal{B}$ on a Read-Write RAM requires a lookup table of size $\Omega(B)$, where B is the size of $\mathcal{B}$.*

Proof: *To compute f, a TM needs to be able to write all characters in the range $\mathcal{B}$. A TM uses a tuple in its program to decide what to write and hence the TM needs at least B tuples in the program. Hence, the size of the lookup table under the Read-Write RAM is at least B.* *QED*

When considering a character from the alphabet as a binary number the lower bound for performing bitwise boolean operations on such a character under the Read-Write RAM model, is:

Corollary 2.1 *A Read-Write RAM requires a lookup table of size* $\Omega(2^w)$ *to perform bitwise boolean operations.*

Proof: *The range of the boolean operations is the whole alphabet, which is of size* 2^w*. Hence, from Theorem 2.1 we know that the size of the lookup table under the Read-Write RAM is* $\Omega(2^w)$*.* *QED*

We use the term *register* when referring to a specific memory location storing a word and the term *word* to denote a w-bit value. The notation $\mathtt{W}_i$ is used to denote the ith bit of the word/register `W`, where $0 \leq i < w$. The least significant bit of a word/register is bit 0 while the most significant bit is bit $w-1$. When depicting a word/register we place the most significant bit to the left. Hence, in the 8-bit word `abcdefgh`, `a` is the most significant bit 7 while `h` is the least significant bit 0. We let `ONE` denote the w-bit word consisting of w ones (`1..1`), and `ZERO` denote the w-bit word consisting of w zeros (`0..0`).

In the RAM model all the bits are unique for all registers. The RAMBO model is an extended RAM model which also has a part of memory where a bit may occur in several registers or in several positions in one register. The way the bits occur in this part of the memory has to be specified as part of the model. If a bit occurs in more than one position in a register (it is overlapped), and different values are written to the bit, then the bit will store an arbitrary value.

The RAMBO model was suggested by Fredman and Saks [7], and further described by Brodnik [3]. One variant called *Yggdrasil* was used by Brodnik et al. to achieve a worst case constant time priority queue [4]. In this paper we use several variants of the RAMBO model.

The problem of performing boolean operations in different models of computations has been studied extensively. Rennard describes how to perform the boolean operations in the Game of Life model [15]. Shamir describes a paradigm called *visual computation* in which he shows that the boolean operations can be computed [16]. Ogihara et al. show how to simulate `AND` and `OR` circuits on a DNA computer [14], while Ahrabian et al. simulate `NAND` circuits [1]. Tsai et al. derive a systematic algorithm for constructing quantum boolean circuits [17]. The reason to consider the boolean operations is that arithmetic operations, e.g. addition, multiplication and so on, can be computed, using only them, in $O(w^{O(1)})$ time and no additional space.

2 Shift and Rotation Operations

In this section we introduce three new variants of RAMBO (*Line*, *Tail*, and *Circle*) and use them to implement shift and rotation operations. In all examples in this section we assume that $w = 8$ and `x=abcdefgh`.

2.1 Shift

To shift the bits of a word we use the *Line* variant of RAMBO. Line consists of $2w$ bits used to store $w+1$ registers of size w bits. We label the bits λ_i, where

$0 \leq i < 2w$. To store bit j of register `line[`l`]` we use bit λ_i, where $i = j + l$.

As an example, let us write `ZERO` to `line[8]` and afterwards write `x` to `line[0]`. Then $\lambda_{15} = \lambda_{14} = \ldots = \lambda_9 = \lambda_8 =$ `0`, $\lambda_7 =$ `a`, $\lambda_6 =$ `b`, $\ldots$, $\lambda_1 =$ `g`, and $\lambda_0 =$ `h`. Now, since register `line[3]` consists of $\lambda_{10}\lambda_9\lambda_8\lambda_7\lambda_6\lambda_5\lambda_4\lambda_3$, i.e. `000abcde`, reading this register gives the same result as right shift of `x` three steps. In general, reading register `line[`δ`]` gives the result of right shift δ steps. Similarly, after initializing `line[0]` with `ZERO`, writing `x` to register `line[`δ`]` and reading `line[0]` gives us the result as if `x` is shifted left δ steps (cf. Alg. 2.1). Hence,

Lemma 2.1 *We can perform both left and right shifts using 3 probes and* $2w$ *bits.*

2.2 Arithmetic Shift

To perform arithmetic right shift of a word we use the Tail variant of RAMBO. This variant uses w bits to store w registers of size w bits. We label the bits θ_i, where $0 \leq i < w$. To store bit j of register `tail[`l`]` we use bit θ_i, where $i = min(j + l, w - 1)$.

In this example we write `x` to `tail[0]`, then $\theta_7 =$ `a`, $\theta_6 =$ `b`, $\ldots$, $\theta_1 =$ `g`, and $\theta_0 =$ `h`. Now, since register `tail[3]` consists of $\theta_7\theta_7\theta_7\theta_7\theta_6\theta_5\theta_4\theta_3$, i.e. `aaaabcde`, reading this register gives the same result as arithmetic right shift three steps. In general, reading register `tail[`δ`]` gives the result of arithmetic right shift δ steps (cf. Alg. 2.1). Hence,

Lemma 2.2 *We can perform arithmetic right shift using 2 probes and* w *bits.*

2.3 Rotation

Rotation (also known as *barrel shift*) takes the bits which have been shifted out at one end and shifts them in on the other end. To perform rotations we use the Circle variant of RAMBO which uses w bits to store w registers of size w. We label the bits ς_i, where $0 \leq i < w$. To store bit j of register `circle[`l`]` we use bit ς_i, where $i = (j + l) \bmod w$.

Again, when we write `x` to `circle[0]`, then $\varsigma_7 =$ `a`, $\varsigma_6 =$ `b`, $\ldots$, $\varsigma_1 =$ `g`, and $\varsigma_0 =$ `h`. Since register `circle[3]` consists of $\varsigma_2\varsigma_1\varsigma_0\varsigma_7\varsigma_6\varsigma_5\varsigma_4\varsigma_3$ reading it gives the same result as right rotation of x three steps. In general, reading register `circle[`δ`]` gives the result of right rotation δ steps (cf. Alg. 2.1). Further, writing the word to register `circle[`δ`]` and reading `circle[0]` gives the same result as left rotation δ. Hence,

Lemma 2.3 *We can perform both left and right rotations using 2 probes and* w *bits.*

The three lemmata above give us:

Theorem 2.2 *We can perform any of the five shifting operations using* $4w$ *bits in at most 3 probes.*

```
word shiftRight(word a, int δ)
  line[w] = ZERO; line[0] = a; return line[δ];
word shiftLeft(word a, int δ)
  line[0] = ZERO; line[δ] = a; return line[0];
word arithShiftRight(word a, int δ)
  tail[0] = a; return tail[δ];
word rotateRight(word a, int δ)
  circle[0] = a; return circle[δ];
word rotateLeft(word a, int δ)
  circle[δ] = a; return circle[0];
```

Algorithm 2.1: Methods to compute right and left shift, arithmetic right shift and right and left rotation δ steps of `a`.

3 Boolean Operations

We continue with the boolean operations starting with 1-bit values and then generalize to w-bit values. Furthermore, we show how to, simultaneously, perform different boolean operations on the w-bit arguments.

We assume that the reader is familiar with the 16 different boolean operations on two arguments `a` and `b`, where $\mathtt{a}, \mathtt{b} \in \{0, 1\}$ (cf., any textbook on the subject, e.g., "Discrete and Combinatorial Mathematics" [8]).

3.1 Simple Boolean Operations

To describe how to compute the boolean operation we use constants $\mathcal{C} = \{\mathcal{Z}, \mathcal{O}, \mathcal{A}, \mathcal{B}\}$ as indices into a table `val[`$|\mathcal{C}|$`]`. The table is used to store constants `val[`$\mathcal{Z}$`]=ZERO`, `val[`$\mathcal{O}$`]=ONE` and values `val[`$\mathcal{A}$`]=a`, `val[`$\mathcal{B}$`]=b`. We also have an array `r[2]` to store two 1-bit values. The values from `val` are used both as indices into and as values of `r`. At the end `r` also contains the result of our operation. We compute a given boolean operation by three writes (call them steps) into `r` and finally read the result from `r`.

As an example, let us compute `a AND b`. We want to find the result in `r[0]`. The result should be 1 if neither `a` nor `b` are 0. Hence, we initialize `r[0]` with 1,

$$r[0] = 1 \ , \tag{1}$$

then we write 0 to `r[0]` if `a` $==$ 0 or `b` $==$ 0. Instead of checking if `a` $==$ 0 we can just write 0 to `r[val[`$\mathcal{A}$`]]`,

$$r[a] = 0 \ , \tag{2}$$

since if `a` $==$ 1 we will write 0 to `r[1]` which does not affect our result. Similarly for `b`,

$$r[b] = 0 \ . \tag{3}$$

Finally, register `r[0]`, contains a 0 if either or both of `a` and `b` were 0 and 1 otherwise,

$$r[0] \rightarrow res \ . \tag{4}$$

On the other hand, when computing `a NOR b`, we initialize `r[1]` with 1, and write 0 to both `r[val[`$\mathcal{A}$`]]` and `r[val[`$\mathcal{B}$`]]`. Then if neither `a` nor `b` are 1, `r[1]` will still contain 1 and 0 otherwise.

It turns out that all 16 boolean operations can be computed in the same way where the index of `r` and the value stored into `r` at each step depends only on the opCode of the boolean operation. A function $f_{i,j}$`(opCode)` can be used to decide which value to write into which register (cf. Alg. 2.2). The

```
bool boolOp(int opCode, bool a, bool b)
  val[A] = a; val[B] = b;
  r[val[f1,1(opCode)]] = val[f1,2(opCode)];
  r[val[f2,1(opCode)]] = val[f2,2(opCode)];
  r[val[f3,1(opCode)]] = val[f3,2(opCode)];
  return r[val[f1,1(opCode)]];
```

Algorithm 2.2: Method to compute any boolean operation.

function $f_{i,j}$`(opCode)` can be tabulated using a table `F[opCode][i][j]`, where we, for the sake of simplicity, let indices into the table `F` (and its variants we will introduce later) always start at 1 (cf. Alg. 2.3 and Alg. 2.4). The second line in `F`, for example, corresponds to the `AND` operation (i.e. the `opCode` of `AND` is 2), where {$\mathcal{Z}$`,` $\mathcal{O}$} means write `val[`$\mathcal{O}$`]` to `r[val[`$\mathcal{Z}$`]]` (i.e., `r[ZERO]=ONE`) etc., which matches Eq. 1 – 3.

The size of table `F` is $\langle$# of boolean operations$\rangle \cdot \langle$# of steps$\rangle \cdot 2 \cdot \lg |\mathcal{C}| = 192$ bits. Hence, `boolOp` in Alg. 2.4 computes any 1-bit boolean operation, in 34 memory probes (reads or writes), using 198 bits (besides `F`, the arrays `val` and `r` use 6 bits). As we shall see later, we can compress the table `F` to 96 bits which totals to 102 bits over all. Hence, we conclude with:

Lemma 2.4 *We can compute any boolean operation using* 102 *bits of memory using* $O(1)$ *reads and writes only.*

This is worse than the 64 bits straighforward table lookup algorithm under the RAM model. But in the next section we build on this and get a solution for w-bit words.

3.2 w-bit Bitwise Boolean Operations

To compute w-bit bitwise boolean operations we use a variant of the RAMBO model which we refer to as *Twin*. It consists of $2w$ bits labeled $\tau_{i,j}$ where $0 \leq i < w$ and $j \in \{0, 1\}$ (see Fig. 2.1). Although there are only $2w$ bits in Twin, they represent 2^w registers. The register at address $a_{w-1}a_{w-2}\ldots a_0$ (denoted `twin[`$a_{w-1}a_{w-2}\ldots a_0$`]`) is stored using the bits $\tau_{w-1,a_{w-1}}\tau_{w-2,a_{w-2}}\cdots\tau_{0,a_0}$, i.e.,

```
int F[opCode][i][j] = {
  {{Z, Z}, {A, Z}, {B, Z}},/* 0 */
  {{Z, O}, {A, Z}, {B, Z}},/* a AND b */
  {{Z, Z}, {A, Z}, {B, A}},/* NOT (a IMPLIES b) */
  {{Z, O}, {A, Z}, {B, A}},/* a */
  {{Z, O}, {A, O}, {B, Z}},/* NOT (b IMPLIES a) */
  {{Z, O}, {A, O}, {B, Z}},/* b */
  {{Z, Z}, {A, O}, {B, A}},/* a XOR b */
  {{Z, O}, {A, O}, {B, A}},/* a OR b */
  {{O, O}, {A, Z}, {B, Z}},/* a NOR b */
  {{O, O}, {A, Z}, {B, A}},/* a XNOR b */
  {{O, O}, {A, O}, {B, Z}},/* NOT b */
  {{O, O}, {A, O}, {B, A}},/* b IMPLIES a */
  {{O, O}, {A, Z}, {Z, Z}},/* NOT a */
  {{O, O}, {A, Z}, {B, O}},/* a IMPLIES b */
  {{Z, O}, {A, O}, {B, O}},/* a NAND b */
  {{O, O}, {A, O}, {B, O}} /* 1 */ }
```

Algorithm 2.3: Table `F` used by `boolOp` in Alg. 2.4.

```
bool boolOp(int opCode, bool a, bool b)
  val[A] = a; val[B] = b;
  r[val[F[opCode][1][1]]] = val[F[opCode][1][2]];
  r[val[F[opCode][2][1]]] = val[F[opCode][2][2]];
  r[val[F[opCode][3][1]]] = val[F[opCode][3][2]];
  return r[val[F[opCode][1][1]]];
```

Algorithm 2.4: Method to compute any boolean operation using table lookup.

the ith bit of `twin[`$a_{w-1}a_{w-2}\ldots a_0$`]` is τ_{i,a_i} For example, `twin[0011]` consists of the bits $\tau_{3,0}\tau_{2,0}\tau_{1,1}\tau_{0,1}$.

bit:	3	2	1	0
	$\tau_{3,1}$	$\tau_{2,1}$	$\tau_{1,1}$	$\tau_{0,1}$
	$\tau_{3,0}$	$\tau_{2,0}$	$\tau_{1,0}$	$\tau_{0,0}$

Figure 2.1: Twin memory with 4-bit words ($w = 4$)

To get a better feeling for how this memory behaves, let us assume that all bits in the memory are zero (Fig. 2(a)). Then we write `1111` to `twin[0101]` (Fig. 2(b)). Now, if we read `twin[0011]` we get the word 1001, and `twin[1100]` gives 0110.

The `twin` registers behave as w parallel arrays `r`. Hence, similarly to the computation of `AND` above, if we want to compute `a AND b` with w-bit registers, we first write `val[`$\mathcal{O}$`]` to `twin[val[`$\mathcal{Z}$`]]`, then we write `val[`$\mathcal{Z}$`]` to both

3	2	1	0
0	0	0	0
0	0	0	0

(a) All bits equal to zero.

3	2	1	0
0	1	0	1
1	0	1	0

(b) `twin[0101]` ← `1111`.

Figure 2.2: Twin example with $w = 4$.

`twin[val[`$\mathcal{A}$`]]` and `twin[val[`$\mathcal{B}$`]]`. As an example (Fig. 2.3) we use `a=0011`, and `b=0101` and study the content of `twin[val[`$\mathcal{Z}$`]]`. After the three writes, register `twin[val[`$\mathcal{Z}$`]]` (Fig. 3(c)) contains 0001 which is the result of bitwise boolean `AND` of 0011 and 0101.

3	2	1	0
–	–	–	–
1	1	1	1

(a) `twin[ZERO]` ← `ONE`.

3	2	1	0
–	–	0	0
0	0	1	1

(b) `twin[a]` ← `ZERO`.

3	2	1	0
–	0	0	0
0	0	0	1

(c) `twin[b]` ← `ZERO`.

Figure 2.3: Computation of `a AND b` using Twin where `a=0011`, and `b=0101`

The special registers `twin` in the Twin RAMBO variant lets us use a method similar to `boolOp` (Alg. 2.4) to compute bitwise boolean operations on w-bit words. We are still using F from Alg. 2.3 but the parameters `a` and `b` and the array `val` are w bits wide. Further, the array `r` is replaced by the twin registers. Hence using $4w + 192$ bits of regular memory and $2w$ bits of RAMBO memory we can compute any of the boolean operations.

As stated above, we can reduce the amount of memory needed by compressing the table `F`. The first, second, and fourth columns only consist of the values $\mathcal{Z}$ and $\mathcal{O}$ and hence only 1 bit for each position is needed. The third column always contains $\mathcal{A}$ and it can be removed entirely. The value in the fifth column is either $\mathcal{Z}$ or $\mathcal{B}$ and needs only 1 bit. The value in the sixth column is either $\mathcal{Z}$, $\mathcal{O}$, or $\mathcal{A}$ and to store such a value we need 1.5 bits. Hence, each row in the table actually only needs 5.5 bit instead of 12 bit which totals to 88 bits for the table. However, we use 2 bits (actually 2 1-bit values) to store the last column, in order to avoid the gory details needed to use only 1.5 bits, which totals to 96 bits for the table. A new table `Fc` stores in columns 1, 2, and 3 the values from column 1, 2, and 4 of `F` respectively. The fourth column stores $\mathcal{Z}$ if the value of the fifth column of `F` is $\mathcal{Z}$ and $\mathcal{O}$ if it is $\mathcal{B}$. The fifth column stores $\mathcal{O}$ if the sixth column of `F` is $\mathcal{O}$ and $\mathcal{Z}$ otherwise. The sixth columns stores $\mathcal{O}$ if the sixth column of `F` is $\mathcal{A}$ and $\mathcal{Z}$ otherwise.

To be able to use the table `Fc` we need to compute the values stored in

the fifth and sixth column of table `F` based on table `Fc`. To get the value from column five of `F` we first write `ZERO` to `twin[ONE]`, then we write `b` to `twin[val[Fc[opCode][4]]]`. Now if `Fc[opCode][4]` was $\mathcal{O}$, `twin[ONE]` will contain `b` and `ZERO` otherwise. We store this value in a variable, `d`, and use it where column five of `F` was used. We compute the value stored in the sixth column of `F` in a similar way and store it in variable, `e`, for later use. Since `Fc` only stores the values $\mathcal{Z}$ and $\mathcal{O}$, the table `val` only need to store two w-bit values (`ZERO` and `ONE`). However, the two variables `d` and `e` are also w-bit values. This gives us:

Theorem 2.3 *We can compute any bitwise boolean operation on w-bit words in 36 memory probes using $4w+96$ bits of regular memory and $2w$ bits of RAMBO memory in $O(1)$ time.*

This is a huge improvement over the $\Omega(2^w)$ bits needed for table lookup under RAM.

The number of memory probes needed can be reduced by using more memory. Since, `Fc` is storing just the indices $\mathcal{Z}$ and $\mathcal{O}$, we can avoid one level of indirection and the usage of the array `val`, by storing `ZERO` and `ONE` directly into a table `Fw`, e.g., the `AND` row is {`ZERO, ONE, ZERO, ONE, ZERO, ZERO`}. This increases the total usage of regular memory to $96w + 2w$ bits, but we only need 29 memory probes (cf. Alg. 2.5) which gives us the following result:

Corollary 2.2 *We can compute any bitwise boolean operation on w-bit words in 29 memory probes using $96w + 2w$ bits of regular memory and $2w$ bits of RAMBO memory in $O(1)$ time.*

Again, this is still a large improvement over the $\Omega(2^w)$ bits needed for table lookup under RAM.

Note that for 1-bit words the total amount of memory is 100 bits of regular memory and no bits of RAMBO memory (`r` is used instead of `twin`) which is a slight improvment over the result in Sect. 3.1.

```
word boolOp(int opCode, word a, word b)
  twin[ONE] = ZERO; twin[Fw[opCode][4]]= b; d = twin[ONE];
  twin[ONE] = ZERO; twin[Fw[opCode][5]]= ONE;
  twin[Fw[opCode][6]]= a; e = twin[ONE];
  twin[Fw[opCode][1]]=Fw[opCode][2];
  twin[a]=Fw[opCode][3];
  twin[d]=e;
  return twin[Fw[opCode][1]];
```

Algorithm 2.5: Method to compute any combination of bitwise boolean operations for w-bit arguments.

Moreover, when storing w-bit values in the table `Fw` we can actually decide which operation we want to perform on individual bits by storing other values

than ZERO and ONE. For example, we can perform, XOR on the bits at even position and AND on the bits at odd positions (XOR-AND).

As an example, we compute, in 2-bit words, XOR for the least significant bit and AND for the most significant bit. The row for this operation in the table Fw would be {00, 10, 01, 11, 00, 01}. The most significant bit in each word corresponds to the values in the AND line of Fc and the least significant bit to the values in the XOR line. If we let a=11 and b=00 the result should be 01. Following the steps in Alg. 2.5 with these values we get the program trace in Fig. 2.4, which gives the expected result.

Instruction	$\tau_{1,1}\tau_{0,1}/\tau_{1,0}\tau_{0,0}$	d	e
twin[11] = 00	00/- -	-	-
twin[11] = 00	00/- -	-	-
d = twin[11]	00/- -	00	-
twin[11] = 00	00/- -	00	-
twin[00] = 11	00/11	00	-
twin[01] = 11	01/11	00	-
e = twin[11]	01/01	00	01
twin[00] = 10	01/10	00	01
twin[11] = 01	01/11	00	01
twin[d] = e	01/01	00	01
twin[00] → res			

Figure 2.4: Trace of boolOp in Alg. 2.5 with a=11, b=00, and Fw[opcode] = {00,10,01,11,00,01}.

Hence, we can support any combination of bitwise boolean operations on individual bits in w-bit words using $6w$ extra bits per combination. Let c be the number of combinations of different boolean operations we wish to support (note $c \leq 16^w$). Then,

Corollary 2.3 *We can compute, in $O(1)$ time, any of c combinations of bitwise boolean operations on individual bits in w-bit words in 29 memory probes using $c \cdot 6w + 2w$ bits of regular memory and $2w$ bits of RAMBO memory.*

4 Addition Operation

Finally, as an example of how to use these bitwise operations we implement addition of two words within our model of computation. When implementing addition in hardware the depth of the circuit has to be at least $\Omega(\log_d w)$ if the fan-in is restricted to d. Addition is in NC^1 [12] and we match the lower bound using the procedure used by Cormen et al. [6, Sect. 29.2.2].

The basic idea is to use a parallel prefix circuit to compute all the carry bits, c, first and then finally the sum is computed as the parity of a, b and c (boolOp(XOR, c, boolOp(XOR, a, b))).

The carry bit $\mathtt{c}_i$ depends on $\mathtt{a}_{i-1}$, $\mathtt{b}_{i-1}$ and maybe $\mathtt{c}_{i-1}$. If $\mathtt{a}_{i-1} = \mathtt{b}_{i-1} = 0$ then $\mathtt{c}_i = 0$ (we *kill* the carry bit), if $\mathtt{a}_{i-1} = \mathtt{b}_{i-1} = 1$ then $\mathtt{c}_i = 1$ (we *generate* the carry bit), and if $\mathtt{a}_{i-1} \neq \mathtt{b}_{i-1}$ then $\mathtt{c}_i = \mathtt{c}_{i-1}$ (we *propagate* the carry bit).

The notation of *carry status* (kill (k), generate (g), and propagate (p)) is used by Cormen et al. and we can compute combined carry status of two consecutive full adders using the carry status operator $\otimes$. The combined carry status is propagate if both the operands are propagate, it is generate if either the second operand is generate or the first is generate and the second is propagate, and otherwise it is kill.

We encode the three values of the carry status $\mathtt{x}_i$ using two bits ($k = 00$, $p = 01$, $g = 10$). Using this encoding it is easy to compute $\mathtt{x}_i$ since $\mathtt{x}_{i_0} = \mathtt{a}_i \texttt{ XOR } \mathtt{b}_i$ and $\mathtt{x}_{i_1} = \mathtt{a}_i \texttt{ AND } \mathtt{b}_i$. Note, that we can compute both bits, in spite of the fact that we deal with two different boolean operations, simultaneously as shown in Fig. 2.4. Furthermore, we can do this simultaneously for all i.

As shown by Cormen et al. [6] the rest of the algorithm uses $O(\lg w)$ boolean operations and shifts. We leave the details of the implementation to the reader.

5 Conclusion

The computation of the bitwise operations under the RAM model using $O(1)$ table lookups requires a table of size $\Omega(2^w)$ while we have presented a solution under the RAMBO model using only $O(w)$ space still using only $O(1)$ table lookups. To support all the bitwise operations we used $4w + 96$ bits of regular memory and $6w$ bits of special RAMBO memory.

Furthermore, we also showed how to support simultaneous combinations of boolean operations using $6w$ additional bits of ordinary memory per combination. To implement addition we took advantage of the combined boolean operations and got a $O(\lg w)$ time solution.

To perform the bitwise operations we introduced four new variants of the RAMBO model which are straightforward to implement in hardware. For a discussion on how to implement new variants of the RAMBO model we refer the interested reader to "Design of High Performance Memory Module on PC100" by Leben et al. [13].

The address decoding for memory under the RAM model is in NC^1 but not in NC^0. The address decoding for the twin memory is in NC^0 while the address decodings for line, tail and circle are in NC^1.

Content-addressable memory (CAM) (also known as associative memory) [9] is another technique where the memory structure is modified. CAM requires additional hardware to handle processing of all memory cells in parallel. The RAMBO variants, on the other hand, only requires modifications to the address decoding.

Acknowledgment

The authors would like to thanks Prof. Svante Carlsson who participated in initial discussions.

References

[1] H. Ahrabian and A. Nowzari-Dalini. DNA simulation of nand boolean circuits. *The Electronic International Journal Advanced Modeling and Optimization*, 6(2):33–41, 2004.

[2] Arne Andersson, Peter Bro Miltersen, Soren Riis, and Mikkel Thorup. Static dictionaries on AC^0 RAMs: Query time $\Theta(\sqrt{\log n / \log\log n})$ is necessary and sufficient. In *37th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 441–450. IEEE Computer Society, IEEE Computer Society, 14–16 October 1996.

[3] Andrej Brodnik. *Searching in Constant Time and Minimum Space (*Minimæ Res Magni Momenti Sunt*)*. PhD thesis, University of Waterloo, Waterloo, Ontario, Canada, 1995. (Also published as technical report CS-95-41.).

[4] Andrej Brodnik, Svante Carlsson, Michael L. Fredman, Johan Karlsson, and J. Ian Munro. Worst case constant time priority queue. *Journal of System and Software*, 78(3):249–256, December 2005.

[5] Stephen A. Cook and Robert A. Reckhow. Time bounded random access machines. *Journal of Computer and System Sciences*, 7(4):354–375, 1973.

[6] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press and McGraw-Hill Book Company, 1990.

[7] Michael L. Fredman and Michael E. Saks. The cell probe complexity of dynamic data structures. In *Proceedings of the 21st Annual ACM Symposium on Theory of Computing*, pages 345–354. ACM Press, May 14–17 1989.

[8] Ralph P. Grimaldi. *Discrete and Combinatorial Mathematics – An Applied Introduction*. Pearson Education, Inc, 5th edition, 2004.

[9] A.G. Hanlon. Content-addressable and associative memory systems. *IEEE Trans. Electronic Computers*, 15(4):509–521, August 1966.

[10] Kai Hwang. *Advance Computer Architecture*. McGraw-Hill, Inc, 1993.

[11] David S. Johnson. A catalog of complexity classes. In van Leeuwen [19], chapter 2, pages 67–161.

[12] Richard M. Karp and Vijaya Ramachandran. Parallel algorithms for shared-memeory machines. In van Leeuwen [19], chapter 17, pages 869–941.

[13] Roni Leben, Marijan Miletić, Marjan Špegel, Andrej Trost, Andrej Brodnik, and Johan Karlsson. Design of high performance memory module on PC100. In *Proceedings Electrotechnical and Computer Science Conference*, pages 75–78, Slovenia, 1999.

[14] Mitsunbori Ogihara and Animesh Ray. Simulating boolean circuits on a DNA computer. *Algorithmica*, 25(2–3):239–250, 1999.

[15] Jean-Philippe Rennard. Implementation of logical functions in the game of life. In Andrew Adamatzky, editor, *Collision-Based Computing*, chapter 17, pages 491–512. Springer, 2002.

[16] Adi Shamir. Visual cryptanalysis. In Kaisa Nyberg, editor, *Advances in Cryptology - EUROCRYPT '98: International Conference on the Theory and Application of Cryptographic Techniques*, volume 1403 of *Lecture Notes in Computer Science*, pages 201–210. Springer, 1998.

[17] I. M. Tsai and S. Y. Kuo. A systematic algorithm for quantum boolean circuits construction. *ArXiv Quantum Physics e-prints*, (quant-ph/0104037), April 2001. `http://arxiv.org/abs/quant-ph/0104037`.

[18] Peter van Emde Boas. Machine models and simulations. In van Leeuwen [19], chapter 1, pages 3–66.

[19] Jan van Leeuwen, editor. *Handbook of Theoretical Computer Science*, volume A: Algorithms and Complexity. Elsevier/MIT Press, Amsterdam, 1990.

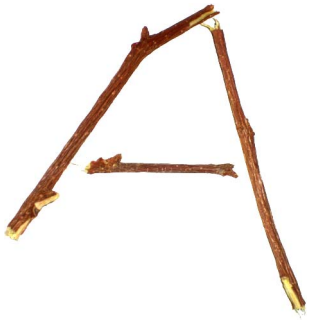
[20] Andrew Chi-Chih Yao. Should tables be sorted? *Journal of the ACM*, 28(3):614–628, July 1981.

Paper 3

Multiprocess Time Queue

To the uneducated, an A is just three sticks.

Eeyore



Extended version of paper published as

Andrej Brodnik and Johan Karlsson, *Multiprocess Time Queue.* In Peter Eades and Tadao Takaoka, editors, *Algorithms and Computation ISAAC 2001, 12th International Symposium*, volume 2223 of *Lecture Notes in Computer Science*, pages 599–609. Springer, December 2001.

Multiprocess Time Queue

Andrej Brodnik * † Johan Karlsson †

Abstract

We show how to implement a bounded time queue for two different processes. The time queue is a variant of a priority queue with elements from a discrete universe. The bounded time queue has elements from a discrete bounded universe. One process has time constraints and may only spend constant worst case time on each operation while the other process may spend more time. The time constrained process only has to be able to perform some of the time queue operations while the other process has to be able to perform all operations. We show how to do a deamortization of the `deleteMin` cost and to provide mutual exclusion for the parts of the data structure that both processes maintain.

1 Introduction

In this paper we look at a special variant of the *Priority Queue* problem which we call the *Time Queue* problem. A time queue is a queue that stores elements together with a time stamp. Newly inserted elements must have a time stamp that lies in the future. The time queue can be used in various ways. One task might be as a time-out manager, where an element has to be processed before some given time otherwise it should be considered to have timed-out and be handled specially. The time queue can also be used for the *simulation event set* problem [3] and other *scheduling* problems.

The time queue supports, given a set $\mathcal{N}$ of N elements, the ordinary operations of a priority queue, `insert`, `min` and `deleteMin`. By convention the highest priority has the lowest numerical value, hence `min`. We refer to the element with the minimum numerical priority value as the *min element* and use t_0 to denote its priority.

Usually a priority queue supports the `decrease-key` operation, which decreases the priority of an element in the queue. The `increase-key` operation is also supported by the time queue and we combine these operations into a general `update` operation, which updates the priority of an element.

Further, a general `delete` operation is also supported in the time queue. Therefore, we let `insert` return a *finger* to the inserted element, which can be

*Department of Theoretical Computer Science, Institute of Mathematics, Physics, and Mechanics, Ljubljana, Slovenia

†Department of Computer Science and Electrical Engineering, Luleå University of Technology, Luleå, Sweden

used by other operation such as `delete` and `update`. It is convenient that the `min` operation also returns a finger. Since we use fingers we need operations to get the priority and element from the finger, `value` and `data` respectively. In this paper we use the terms element and finger to an element interchangeably.

Finally, the time queue supports deletion of all elements with a priority less than a specific value, using the operation `delLessThan`. The `delLessThan` can be augmented with an additional function, $\mathcal{F}$, that is called for each deleted element. Note that this forces the `delLessThan` to take $\Omega(d \cdot F)$ time where d is the number of deleted elements and F is the running time of the function $\mathcal{F}$. Without loss of generality we assume that $F = \Theta(1)$.

In the time queue the priorities are times. We assume that time is a discrete value and hence the time queue is restricted to only support priorities that are discrete (e.g. integers). We require that for the time t_e of the newly inserted or of the updated element e must hold $t_e > t_0$, which means that the min time t_0 is non-decreasing, the time queue is *monotonic* [12]. Moreover, we require that the time for any element in the time queue is less than $t_0 + C$, where C denotes the maximum duration of any element (cf. *maximum event duration* [4]). To sum up: time is drawn from a bounded discrete universe.

The above description gives the following formal definition:

Definition 3.1 *The* Time Queue problem *is the problem of maintaining a set, $\mathcal{N}$, of elements to support the following operations:*

`insert(`e`,`t`):`f	*Iff $t_0 < t \leq t_0 + C$ then let $\mathcal{N} := \mathcal{N} \cup \{e\}$ and return a finger f to the newly inserted element.*
`delete(`f`)`	*let $\mathcal{N} := \mathcal{N} \backslash \{f\}$.*
`min():`f	*Find the min element and return a finger, f, to it.*
`deleteMin()`	*Delete the min element.*
`update(`f`,`t`)`	*Iff $t_0 < t \leq t_0 + C$ then change the time of f to t.*
`delLessThan(`t`, `$\mathcal{F}$`)`	*Delete all elements with time less than t and call the function '$\mathcal{F}$' for each of the deleted elements.*

where t_0 is the priority of the min element and C is the maximum duration of any element.

This research was initiated by a manufacturer of a firewall. In their firewall IP packets are processed in two different paths called *fast* and *slow* path. The fast path must not be delayed when using the time-out manager and this process needs only some of the operations of the time queue. The slow process has to be able to perform all the operations, but it is not that time sensitive.

Hence, in our model, we have two different processes manipulating the data structure. The first process (*fast*) has to be able to perform the `min`, `value`, `data` and a restricted `update` operations. The second process (*slow*) has to be able to perform all the operations on the time queue. The *fast* process is time critical and must not be delayed, i.e. , the operations it uses must run in $O(1)$ worst case time. The fast process `update` only needs to update elements in the near future, i.e. , only elements with current and new time less than $t_0 + \epsilon$ (ϵ is to be defined later).

Our main goal is to implement the operations of the *fast* process to run in $O(1)$ worst case time, hence amortized or expected time is not good enough. To do this, we let only the operations `deleteMin` and the `delLessThan` change the min element. this makes the operations `delete` and `update` more restricted, and, consequently, less complicated than `deleteMin` and the `delLessThan`. We refer to the time queue problem with these restrictions as the *restricted Time Queue problem*.

Furthermore, `delLessThan` is called by the slow, and this at least every c time units for some small value c.

To allow the two processes to share data we need mutual exclusion of the operations. For this we use locks and the interface to the locks has to provide both blocking and non-blocking locking functions. We also assume that the processes can pass messages asynchronously.

To compare different priority queues both theoretically and practically the *hold model* [10] has been used. In this model a priority queue of size N is created and a *hold* operation is performed a number of times. The `hold` operations is a sequence of `min`; `deleteMin`; and `insert` operations, hence N is not changed. The priority of the newly inserted element is $t_0 + d$ for some value d.

In the following section we look at how other solutions can be used to solve the restricted time queue problem, in particular the *Calendar Queue* by Brown [3]. In Sect. 3 we present our solution, a modification of the calendar queue, to support the operations of the fast process while Sect. 4 concludes the paper.

2 Previous work

A number of solutions for the priority queue problem can be used to solve the time queue problem for one process with only small modifications if any. The standard *heap* described by Williams [18] can be modified to use fingers by adding a dictionary that stores the position in the heap for each element. The heap solution (*heap* in Table 3.1) even works if the maximum duration is unbounded and it only needs $O(N)$ space. The model used is the pointer machine model [11].

Van Emde Boas et al. proposed a data structure they call a *stratified tree* which supports the time queue operations in $O(\lg \lg C)$ time (*vEB* in Table 3.1) [13, 14]. However, the stratified tree needs $O(C + N)$ space. The model is the pointer machine model.

Willard shows how perfect hashing (see [6, 8]) can be used to improve the space bound to $O(N)$ for the stratified tree [17] (*vEB-W* in Table 3.1). The model is the RAM model [15] of the stronger cell probe model [19] due to the hashing.

More recently Andersson and Thorup improved their *exponential search* trees to achieve worst case performance of $O(\sqrt{\lg N / \lg \lg N})$ [1] (*EST* in Table 3.1). The model used here is the RAM model.

Brodnik et al. showed how a *split tagged tree* can be used to achieve worst case constant time for all the time queue operations (*SST* in Table 3.1) [2].

They use $O(C + N)$ space in the Yggdrasil implementation [2] of the RAMBO model [9].

So far we have seen the bounds in Table 3.1, with the Calendar queue (CQ) presented below.

Operation	Heap	vEB	vEB-W
`insert`	$O(\lg N)$	$O(\lg\lg C)$	exp $O(\lg\lg C)$
`delete`	$O(\lg N)$	$O(\lg\lg C)$	exp $O(\lg\lg C)$
`min`	$O(1)$	$O(1)$	$O(1)$
`deleteMin`	$O(\lg N)$	$O(\lg\lg C)$	exp $O(\lg\lg C)$
`hold`	$O(\lg N)$	$O(\lg\lg C)$	exp $O(\lg\lg C)$
`update`	$O(\lg N)$	$O(\lg\lg C)$	exp $O(\lg\lg C)$
Space	$O(N)$	$O(C + N)$	$O(N)$

Operation	EST	STT	CQ
`insert`	$O(\sqrt{\lg N/\lg\lg N})$	$O(1)$	am $O(1)$
`delete`	$O(\sqrt{\lg N/\lg\lg N})$	$O(1)$	am $O(1)$
`min`	$O(1)$	$O(1)$	$O(1)$
`deleteMin`	$O(\sqrt{\lg N/\lg\lg N})$	$O(1)$	am exp $O(1)$
`hold`	$O(\sqrt{\lg N/\lg\lg N})$	$O(1)$	exp $O(1)$
`update`	$O(\sqrt{\lg N/\lg\lg N})$	$O(1)$	$O(1)$
Space	$O(N)$	$O(C + N)$	$O(N)$

Table 3.1: Time bounds for different solutions to the Time Queue problem

2.1 The Calendar Queue

The *Calendar Queue* data structure described by Brown [3] and analyzed by Erickson et al. [7] is a priority queue specially designed for the event set problem. Erickson et al. give a short and good description of the calendar queue that we restate here.

"A *calendar queue* has M buckets numbered 0 to $M - 1$, a current bucket with index i_0, a bucket width δ, and a current time t_0. We have the relationship that $i_0 = (t_0 \operatorname{div} \delta) \bmod M$. For each element e in the calendar queue, $t_e \geq t_0$, and element e is located in bucket i if and only if $i \leq (t_e \operatorname{div} \delta) \bmod M < (i+1)$."

The calendar queue is implemented as an array of lists, which we denote `buckets`. Depending on *bucket discipline* the lists in the buckets are either sorted or unsorted. In unsorted buckets `insert` takes constant time and `min` takes time proportional to the number of elements in the bucket. On the other hand, in sorted buckets `min` (`deleteMin`) takes constant time and `insert` time proportional to the log of the number of elements in the bucket. In Brown's and Erickson's descriptions all buckets use the same bucket discipline.

Brown [3] suggests to use a doubling technique to adjust the number of buckets M to be $\Theta(N)$ where N is the number of elements in the queue. Hence, when inserting an element and N becomes greater than M, we allocate $2M$ new buckets, copy all the elements to the new buckets and deallocate the old buckets. When deleting an element and N becomes less than $M/4$, we allocate $M/2$ buckets, copy all the elements and deallocate the old buckets. We see that, if a doubling of the number of buckets occurs when there are N_0 elements, at least N_0 new elements has to be inserted into the queue before the next doubling

will occur. Hence the copying cost of the $2N_0$ elements at the second double can be charged to the insertion of the N_0 elements. Similarly for deletes and the copying cost when halving the number of buckets. The bucket width δ should be adjusted to match the average distance between elements in the queue in order to get an expected constant number of elements in each bucket. Hence, insert and delete can be done in expected $O(1)$ amortized time. Brown gave empirical evidence that the calendar queue achieves expected constant time for the `hold` operation. In other words, if we choose δ and M properly, the number of elements in each bucket will be $O(1)$.

Erickson et al. (see "Optimizing Static Calendar Queues" [7] for details, *Static* here means that the number of elements in the queue is unchanged, not that all events have to be known in advance) analyzed the calender queue with unsorted buckets. They describe how to choose δ and M under the assumption that only the `hold` operation is used (the case for which Brown gave empirical evidence). The value d in the `hold` operations is here defined by a random variable with probability density e. In essence, choose $\delta = \sqrt{2}\frac{\mu}{N}$ where μ is a function of e. Using this bucket width and infinitely many buckets the expected time is constant for the `hold` operation. Given a maximum duration C, choosing $M \geq C \operatorname{div} \delta + 1$ will guarantee no loss of performance over choosing infinitely many buckets. If a small degradation of the performance is acceptable one can choose $M = rN$, where r depends on the allowed degradation.

A variation of the time queue problem has been studied by Varghese and Lauck [16]. They look at the problem of providing a timer facility for an operating system. In the timer facility problem the `delLessThan` operation is called once for each time t (i.e. , $c = 1$). Also even if $t < t_0$. The solution suggested by Varghese and Lauck, called *Hashed and Hierarchical Timing Wheel*, is very similar to the Calendar Queue.

3 Our Solution

We will now modify the calendar queue to achieve $O(1)$ worst case time for the `min`, `update`, `value` and `data` operations, and see under what conditions we can expect `deleteMin` and `delLessThan` to run in $O(1)$ time per deleted element. As Erickson et al. we will use the *unsorted* bucket discipline to achieve $O(1)$ worst case time for insertion into a bucket. We use lists of doubly linked nodes in each bucket and let a finger be a reference to the node that stores the element. Given a finger to the element, this achieves $O(1)$ worst case time for deletion in a bucket.

As pointed out by Thorup [12] we can always, in any monotonic priority queue, make the `min` operation run in $O(1)$ worst case time by remembering the element (and its priority) that was deleted by the last `deleteMin` and consider it part of the priority queue. We implement this by letting `deleteMin` find the element that will be min when the current min is deleted and store a finger to this element.

Since an `update` is a `delete` followed by an `insert`, if we can support `delete`

and `insert` in $O(1)$ worst case time we also have `update` in $O(1)$ worst case time. The reason for the amortization in the calendar queue is the copying of elements when M is changed. If we never need to copy any elements during an `insert` (`delete`) the time for these operations is worst case. Hence, if M and δ are fixed the copying is never needed and we achieve $O(1)$ worst case time for the `min`, `update`, `value` and `data` operations.

Under what conditions can we expect `deleteMin` to run in $O(1)$ time, and can we improve these conditions in some way? The approach with fixed M and δ is what Brown started with in his description of calendar queue. He noted that this will lead to inefficient space use if $N << M$. Moreover, if $N << M$, `deleteMin` may have to search many empty buckets to find the next element, `deleteMin` takes $O(M)$ time in the worst case. On the other hand, if $N >> M$, the current bucket may contain many elements, `deleteMin` takes $O(N)$ time in the worst case. However, on the average $O(M/N + N/M)$ time is needed. From the discussion above we conclude that we can expect $O(1)$ time for `deleteMin` if $N = \Theta(M)$ and the elements are evenly distributed among the buckets.

To improve these conditions we will focus on the case where $N = \Omega(M)$, and the main problem of `deleteMin` is to find the next element in the current bucket. Since the elements in the buckets are unsorted it takes time proportional to the number of elements in the bucket to find the new min.

One way of reducing the time could be to keep the current bucket sorted, then deletion of the min element in the bucket would take $O(1)$ time. Each element would then be involved in one sorting and the amortized cost per element would be $s(k)$ where $s(k)k$ is the cost of sorting k elements (cf. equivalence between sorting and priority queues [12]). This makes `update` of elements within the bucket i_0 too expensive for the *fast* process.

Instead, we use δ buckets of width 1, implemented as an array of doubly linked lists denoted `head`. We store the elements of the current bucket i_0 in `head[`j`]` where $j = t_e \bmod \delta$. Hence, each list in the head only stores elements with the same priority. Now `update`, `insert`, `delete` and `min` are still $O(1)$ in the worst case even though the constant is a bit higher.

In the analysis of `deleteMin` we denote the number of elements in a bucket i by B_i and the number of elements in the `head` by H. Finding the new min in the head is similar to finding the next non empty bucket in the calendar queue, which is done in $O(M/N)$ time on the average. Hence in a head with more than one element it takes $O(\delta/H + 1)$ time on the average. If $H = \Omega(\delta)$ this is $O(1)$. When the last element of the head is deleted, and all the δ buckets are empty, we need to move all the B_{i_0+1} elements of bucket i_0+1 into the head and increase i_0 by one. The cost of the copying is $O(B_{i_0+1})$, which indicates that the worst case cost of `deleteMin` is $O(N)$. However in an amortized analysis the cost of copying an element can be charged to the operation that deletes the element from the head, which makes the amortized time $O(1)$ for `deleteMin`.

Finally, we do a deamortization of the `deleteMin` operation to achieve $O(1)$ expected time instead of amortized time. The deamortization is done by using a second head denoted `head2` and move $\lceil B_{i_0+1}/H \rceil$ elements from bucket $i_0 + 1$ into `head2` in each `deleteMin` operation. When the last element is deleted from

the head the rest of the elements are moved from bucket $i_0 + 1$ into `head2` and the two heads are swapped. If an element should be inserted (updated) into bucket $i_0 + 1$ it will instead be inserted into `head2`. Hence B_{i_0+1} will never increase and therefore the number of elements in bucket $i_0 + 1$ will be $O(1)$ when the last element is deleted from the head. If $H = \Omega(B_{i_0+1})$ the cost is $O(1)$ for copying the elements.

Now if $N = \Omega(M)$, $H = \Omega(\delta)$ and $H = \Omega(B_{i_0+1})$ we have $O(1)$ expected time for `deleteMin`. If not, the time for `deleteMin` and `hold` is exp $O(M/N + B_{i_0+1}/H + \delta/H)$ and $O(M + N + \delta)$ worst case. If we choose $\delta = \Theta(M)$ the above conditions reduce to $H = \Omega(\delta)$ (since $N \geq H$) and $H = \Omega(B_{i_0+1})$ where the second condition depends on the distribution of the elements among the buckets.

Now let us see what conditions are needed if a sequence of `delLessThan` calls are used instead of `deleteMin`. First we note that $O(\delta/c)$ calls are made between two changes of heads. Consequently, if $\lceil B_{i_0+1}/(\delta/c) \rceil$ elements are moved each time `delLessThan` is called, all the elements in bucket $i_0 + 1$ will be moved to the second head before the next head swap. If $\delta = \Omega(B_{i_0+1})$ then only a constant number of elements are moved each time. If the min element is deleted, `delLessThan` needs to find the next element to be min which is done in $O(1)$ time on the average if $H = \Omega(\delta)$. However, assume that p empty buckets have to be scanned to find the next element to be min, when the min element is deleted, then approximately p/c `delLessThan` calls are made before the min is deleted again. Hence on the average c buckets are scanned by `delLessThan` even if $H = o(\delta)$. Note that this is true even if we need to scan the `buckets` array to find the next non empty bucket. The condition we are left with is $\delta = \Omega(B_{i_0+1})$ which is fulfilled if there is only a constant number of elements with equal priority.

Since we know the maximum duration C of all the elements, we can choose δ and M to cover this range, hence $C = M\delta$. We choose $M, \delta = O(\sqrt{C})$ to have $\delta = \Theta(M)$. This gives a space bound of $O(\sqrt{C}+N)$ since the number of buckets and the number of buckets in the heads are $O(\sqrt{C})$. We note that when using $\delta = M = \sqrt{C}$ both mod$\sqrt{C}$ and div$\sqrt{C}$ can be computed fast if $C = 2^h$ and even if $C \neq 2^h$ the approximation $\delta = 2^{\frac{\lg C}{2}}$ is good enough. The above analysis leads to the time and space bounds in Table 3.2.

Operation	Our modified CQ
`insert`	$O(1)$
`delete`	$O(1)$
`min`	$O(1)$
`deleteMin`	exp $O(1)$ if $H = \Omega(\delta), H = \Omega(B_{i_0+1})$
`hold`	exp $O(1)$ if $H = \Omega(\delta), H = \Omega(B_{i_0+1})$
`update`	$O(1)$
`delLessThan`	exp $O(1)$ if $\delta = \Omega(B_{i_0+1})$
Space	$O(\sqrt{C} + N)$

Table 3.2: Time bounds and space for our solution to the Time Queue problem

3.1 Representation and Algorithms

The data structure consists of buckets, two heads of buckets, a finger to the min element and an index into the current bucket (see Alg. 3.1 and Fig. 3.1 for an example). Both the buckets and the two heads of buckets are arrays of lists of doubly linked nodes. We let a finger be a reference to a node.

```
typedef void * Element;
typedef struct node {
  struct node * prev;
  struct node * next;
  Element       e;
  int           t;
  Bool          inHead;
} NODE,
typedef struct list {
  NODE *        first;
  NODE *        last;
  int           nrOfElements;
} LIST;
typedef struct tq {
  LIST          buckets[M];
  LIST          head[δ];          int     H;
  LIST          head2[δ];         int     H2;
  NODE *        min;
  int           i0;
} TQ;
```

Algorithm 3.1: Representation of the Time Queue

We first look at the algorithms for only one process (the slow one) and later see what modifications are needed for the fast process.

- During `insert` (Alg. 3.3) we calculate the bucket index for the new element and check if the element should be in either of the heads. If it is we calculate the head index j and insert the new element at the end of the list, otherwise we insert it at the end of the list of the proper bucket.

- In the `delete` (Alg. 3.4) we have the finger the element and we can easily delete it from the appropriate list. If it is the last element in the list we mark the bucket as empty.

- The `update` (Alg. 3.5) is, as said, a deletion followed by an insertion.

- The `min` returns the stored min finger.

- The `deleteMin` (Alg. 3.6) first deletes the min element. Then it searches for the next element that should be min and updates the min reference.

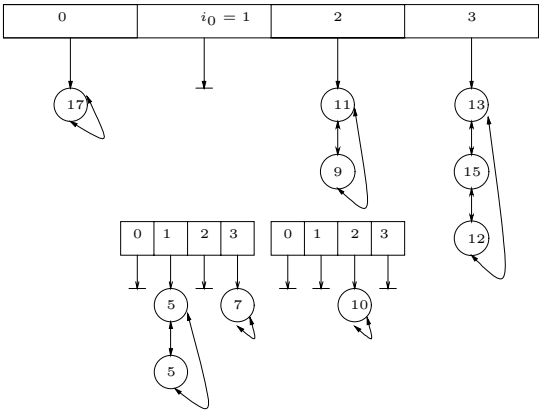


Figure 3.1: Time Queue representation with: $C = 16$, $N = 9$, $M = \delta = 4$, $B_0 = 1$, $B_1 = 0$, $B_2 = 2$, $B_3 = 3$, $H = 2$, $H2 = 1$, $t_0 = 5$ and $i_0 = 1$

Finally, the routine moves some of the elements from the next bucket into `head2`.

If the deleted element was the last element in the head, it first search for the next non empty bucket, moves the remaining elements from that bucket to `head2`, swaps the two heads, and increases i_0. Then it continues with the search for the next element to be min.

- As long as the time t_0 of the min element is less than the specified time, `delLessThan` (Alg. 3.8) gets min, calls the function $\mathcal{F}$ on it, deletes it and finds the new min. Finally, it moves some elements from the next bucket into `head2`.

  If `delLessThan` deletes the last element in the head it search for the next non empty bucket, moves the remaining elements from that bucket to `head2`, swaps the two heads, and increases i_0.

- The operations to get the data and time from a finger, `data` and `value` respectively, only returns the data and time from the linked list node.

3.2 Support for Concurrent Processes

We assume that there are just two processes: one fast process and one slow process. The fast process only has to update elements in the near future ϵ while the remaining updates are sent to the slow process. Without loss of generality we assume that $\epsilon \leq \delta$ and hence only elements in bucket i_0 and bucket i_0+1 may be updated by the fast process. The elements in bucket i_0 are stored in `head` while the elements in bucket i_0+1 may be in both `buckets[`i_0+1`]` and `head2`. We will use three locks to ensure mutual exclusive access to these entities. The

assumption that $\epsilon \leq \delta$ is not really a restriction since if this is not true we only need to add more locks for the buckets that need protection.

Whenever `head`, `H`, i_0 or `min` is read or written we acquire `headLock`. Similarly for `head2Lock` and `bucketLock`. Since only the slow process modifies i_0 and `min` it does not need to acquire the `headLock` in order to read these variables. The fast process always has to acquire the corresponding lock. Alg. 3.9 shows the deletion by the slow process with the proper locks. To avoid deadlocks, we choose to break the circular chain condition by imposing a linear order of the locks [5]. If a process needs more than one lock it has to acquire them in the following order: `headLock`, `head2Lock` and `bucketLock`. The representation of the time queue includes these locks (Alg. 3.2). The `update` (Alg. 3.10)

```
typedef struct tq {
  LIST          buckets[M];
  LIST          head[δ];          int     H;
  LIST          head2[δ];         int     H2;
  NODE *        min;
  int           i0;
  LOCK          headLock;
  LOCK          head2Lock;
  LOCK          bucketLock;
} TQ;
```

Algorithm 3.2: Representation of the Time Queue with locks

for the fast process is only allowed to move elements to/from the heads and corresponding buckets. If the element should be moved to/from another bucket it passes a request to the slow process.

If there are more than one process of each kind special care is needed for the slow processes. We need also to acquire the lock when reading variables in the slow process and have locks for all the different parts of the time queue data structure. More than one fast process can be handled without any special care.

4 Conclusion

We have proposed a solution for two different processes to simultaneously maintain a time queue. One of the processes performs only a subset of the operations in $O(1)$ worst case time, while the other process shall perform all operations. All operations except `deleteMin` and `delLessThan` are performed in $O(1)$ worst case time. `deleteMin` is performed in $O(1)$ expected time and `delLessThan` is performed in $O(1)$ expected time per deleted element.

The main difference from the Hashed and Hierarchical Timing Wheels by Varghese and Lauck [16] is the deamortization of the `deleteMin` and the concurrent solution.

Furthermore, we have shown how to allow one fast and one slow process to maintain our data structure by using locks to provide mutual exclusion.

References

[1] Arne Andersson and Mikkel Thorup. Tight(er) worst-case bounds on dynamic searching and priority queues. In *Proceedings of the 32nd Annual ACM Symposium on Theory of Computing*, pages 335–342. ACM Press, May 21–23 2000.

[2] Andrej Brodnik, Svante Carlsson, Johan Karlsson, and J. Ian Munro. Worst case constant time priority queue. In *Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 523–528, 7–9 January 2001.

[3] Randy Brown. Calendar queues: A fast $O(1)$ priority queue implementation for the simulation event set problem. *Communications of the ACM*, 31(10):1220–1227, October 1988.

[4] Boris V. Cherkassky, Andrew V. Goldberg, and Craig Silverstein. Buckets, heaps, lists, and monotone priority queues. In *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 83–92, 5–7 January 1997.

[5] E. G. Coffman, Jr., M. J. Elphick, and A. Shoshani. System deadlocks. *ACM Computing Surveys*, 3(2):67–78, June 1971.

[6] Martin Dietzfelbinger, Anna Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert, and Robert E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal on Computing*, 23(4):738–791, 1994.

[7] K. Bruce Erickson, Richard E. Ladner, and Anthony LaMarca. Optimizing static calendar queues. In *35th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 732–742. IEEE Computer Society, 20–22 November 1994. Also published as tech. report TR-94-09-02 `ftp://ftp.cs.washington.edu/tr/1994/09/UW-CSE-94-09-02.PS.Z`.

[8] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *Journal of the ACM*, 31(3):538–544, July 1984.

[9] Michael L. Fredman and Michael E. Saks. The cell probe complexity of dynamic data structures. In *Proceedings of the 21st Annual ACM Symposium on Theory of Computing*, pages 345–354. ACM Press, May 14–17 1989.

[10] Douglas W. Jones. An empirical comparison of priority-queue and event-set implementations. *Communications of the ACM*, 29(4):300–311, April 1986.

[11] Arnold Schönhage. Storage modifications machines. *SIAM Journal on Computing*, 9(3):490–508, August 1980.

[12] Mikkel Thorup. On RAM priority queues. In *Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 59–67, 28–30 January 1996.

[13] P. van Emde Boas. Preserving order in a forest in less than logarithmic time. In *16th Symposium on Foundations of Computer Science*, pages 75–84, 1975.

[14] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977.

[15] Peter van Emde Boas. Machine models and simulations. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A: Algorithms and Complexity, chapter 1, pages 3–66. Elsevier/MIT Press, Amsterdam, 1990.

[16] George Varghese and Anthony Lauck. Hashed and hierarchical timing wheels: Efficient data structure for implementing a timer facility. *IEEE/ACM Transaction on Networking*, 5(6):824–834, December 1997.

[17] Dan E. Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Information Processing Letters*, 17(2):81–84, 24 August 1983.

[18] J. W. J. Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7:347–348, 1964.

[19] Andrew Chi-Chih Yao. Should tables be sorted? *Journal of the ACM*, 28(3):614–628, July 1981.

A Pseudo Code

The time queue, `TQ`, (Alg. 3.1 and Alg. 3.2) consists of the arrays, `buckets`,`head`, and `head2`, of lists. These lists are doubly linked lists and each list has a counter, `nrOfElements`, of the number of elements in it. The lists consists of nodes, `NODE`, which stores the element and its time together with an indication, `inHead`, whether the element is in either of the two heads or not. The node also stores references to its neighbours in the list. The `TQ` also stores the number of elements in each of the two heads, `H` and `H2` respectively. Further the current bucket index, i_0, indicates in which of the buckets the min element is. It also has a reference, `min`, to the node storing the min elements and its priority.

```
NODE *
insert(TQ tq, Element e, int t) {
  int    bucket, index;
  NODE  *n;
  Create new node
    n = createNode();
    n->e = e;
    n->t = t;
    n->next = n;
    n->prev = n;
    n->inHead = false;
  bucket = (n->t  div δ) mod M;
  index = n->t mod δ;
  if (bucket == tq.i0) {
    Append n at the end of tq.head[index]
      if (tq.head[index].last != NULL) {
        tq.head[index].last->next = n;
        n->prev = tq.head[index].last;
        n->next = tq.head[index].last->next;
      } else {
        tq.head[index].first = n;
      }
      tq.head[index].last = n;
      tq.head[index].nrOfElements++;
    n->inHead = true;
    tq.H++;
  } else if (bucket == ((tq.i0+1) mod M)) {
    Append n at the end of tq.head2[index]
    n->inHead = true;
    tq.H2++;
  } else {
    Append n at the end of tq.buckets[bucket]
  }
  return n;
}
```

Algorithm 3.3: Insertion into Time Queue

```
void
delete(TQ tq, NODE * finger) {
  int    bucket, index;
  bucket = (finger->t div δ) mod M;
  if (finger->next == finger) {                    /* Last element in list */
    if (!finger->inHead) {
      Clear list tq.buckets[bucket]
        tq.buckets[bucket].first = NULL;
        tq.buckets[bucket].last = NULL;
        tq.buckets[bucket].nrOfElements = 0;
    } else {
      index = finger->t mod δ;
      if (bucket == tq.i0) {
        Clear list tq.head[index]
        tq.H--;
      } else {                          /* (bucket == ((tq.i0+1) mod M)) */
        Clear list tq.head2[index]
        tq.H2--;
      }
    }
  } else {
    Remove node from its list
      finger->next->prev = finger->prev;
      finger->prev->next = finger->next;
      if (!finger->inHead) {
        tq.buckets[bucket].nrOfElements--;
      } else {
        index = finger->t mod δ;
        if (bucket == tq.i0) {
          tq.head[index].nrOfElements--;
          tq.H--;
        } else {                        /* (bucket == ((tq.i0+1) mod M)) */
          tq.head2[index].nrOfElements--;
          tq.H2--;
        }
      }
  }
  free(finger);
}
```

Algorithm 3.4: Deletion from Time Queue

```
void
update(TQ tq, NODE * finger, int t) {
  int    bucket, index;
  bucket = (finger->t div δ) mod M;
  if (finger->next == finger) {                    /* Last element in list */
    if (finger->inHead == false) {
      Clear list tq.buckets[bucket]
    } else {
      index = finger->t mod δ;
      if (bucket == tq.i0) {
        Clear list tq.head[index]
      } else {                              /* (bucket == ((tq.i0+1) mod M)) */
        Clear list tq.head2[index]
      }
    }
  } else {
    Remove node from its list
  }
  finger->t = t;
  finger->inHead = false;
  bucket = (finger->t div δ) mod M;
  index = finger->t mod δ;
  if (bucket == tq.i0) {
    Append finger at the end of tq.head[index]
    finger->inHead = true;
    tq.H++;
  } else if (bucket == ((tq.i0+1) mod M)) {
    Append finger at the end of tq.head2[index]
    finger->inHead = true;
    tq.H2++;
  } else {
    Append finger at the end of tq.buckets[bucket]
  }
}
```

Algorithm 3.5: Updates in Time Queue

```
void
deleteMin(TQ tq) {
  int    bucket, index;
  int    n, j;
  NODE  *finger;
  LIST  *head;
  Delete the min element
    finger = min(tq);
    j = finger->t mod δ
    delete(finger);
    finger = NULL;
  if (tq.H == 0) {                          /* Last element in head deleted */
    Find next non empty bucket
      if (tq.H2 != 0) {
        bucket = (tq.i0 +1) mod M;
      } else {
        bucket = (tq.i0 +1) mod M;
        while ((tq.buckets[bucket].nrOfElements == 0) &&
               (bucket != tq.i0)) {
               bucket = (bucket+1) mod M;
        }
      }
      if (bucket == tq.i0) {                          /* No more elements */
        tq.min = NULL;
        return;
      }
    Move all elements in buckets[bucket] to head2
      while (tq.buckets[bucket].nrOfElements > 0) {
        finger = tq.bucket[bucket].first;
        update(tq, finger, finger->t);
      }
    Swap heads
      head = tq.head;
      n = tq.H;
      tq.head = tq.head2;
      tq.H = tq.H2;
      tq.head2 = head;
      tq.H2 = n;
    Increase i0
      tq.i0 = bucket;
    j = 0;
  }
continued in Alg. 3.7
```

Algorithm 3.6: Deletion of min from Time Queue

```
  Search for next min
    while (tq.head[j].nrOfElements == 0) {
      j++;
    }
  Update the min reference
    tq.min = tq.head[j].first;
  Move B_{i0+1}/H elements from bucket i0 + 1 to head2
    bucket = (tq.i0 +1) mod M;
    for (i = 1;
         i <= ⌈tq.buckets[bucket].nrOfElements/tq.H⌉;
         i++) {
      finger = tq.buckets[bucket].first;
      update(tq, finger, finger->t);
    }
}
```

Algorithm 3.7: Deletion of min from Time Queue (cont.)

```
void
delLessThan(TQ tq, int t, func F) {
  NODE  *finger;
  int    bucket;
  int    index;
  int    i, n;
  LIST  *head;
  while (t < min()->t) {
    F(min());
    Delete the min element
    if (tq.H == 0) {                        /* Last element in head deleted */
      Find next non empty bucket
      Move all elements in buckets[bucket] to head2
      Swap heads
      Increase i0
    }
    Search for next min
    Update the min reference
  }
  Move B_{i0+1}/(δ − (t mod δ)) elements from bucket i0 + 1 to head2
}
```

Algorithm 3.8: Deletion of all elements with time less than t from Time Queue

```
void
delete(TQ tq, NODE * finger) {
  int    bucket;
  LOCK  *l;
  bucket = (finger->t  div δ) mod M;
  if (!finger->inHead) {
    if(bucket == ((tq.i0+1) mod M)) {
      l = bucketLock;
    } else {
      l = NULL;
    }
  } else {
    if (bucket == tq.i0) {
      l = headLock;
    } else {                              /* (bucket == ((tq.i0+1) mod M)) */
      l = head2Lock;
    }
  }
  Acquire(l, BLOCK);
  Delete as in Alg. 3.4
  Release(l);
  free(finger);
}
```

Algorithm 3.9: Deletion from Time Queue with Mutual Exclusion

```
void
update(TQ tq, NODE * finger, int t) {
  int    bucket, bucket2, index;
  bool   gotH2Lock = false;
  bool   gotbucketLock = false;
  if (Acquire(headLock, NOBLOCK)) {
    bucket = (finger->t div δ) mod M;
    bucket2 = (t div δ) mod M;
    if (!(((bucket == i0) || (bucket == (i0 + 1 mod M))) &&
          ((bucket2 == i0) || (bucket2 == (i0 + 1 mod M))))) {
      Release(headLock);
      goto SEND;
    }
    if (((bucket == (i0 + 1 mod M)) && finger->inHead) ||
        (bucket2 == (i0 + 1 mod M))) {
      if (Acquire(head2Lock, NOBLOCK))
        gotH2Lock = true;
      else {
        Release(headLock);
        goto SEND;
      }
    }
    if ((bucket == (i0 + 1 mod M)) && !finger->inHead) {
      if (Acquire(bucketLock, NOBLOCK))
        gotbucketLock = true;
      else {
        if (gotH2Lock)
          Release(head2Lock);
        Release(headLock);
        goto SEND;
      }
    }
    Update as in Alg. 3.5
    if (gotbucketLock)
      Release(bucketLock);
    if (gotH2Lock)
      Release(head2Lock);
    Release(headLock);
    return;
  }
SEND:                                   /* Send request to the slow process */
  SendMesg(update, tq, finger, t);
}
```

Algorithm 3.10: Fast process updates in Time Queue

Paper 4

Worst Case Constant Time Priority Queue

Don't underestimate the value of Doing Nothing, of just going along, listening to all the things you can't hear, and not bothering.

Winnie the Pooh

Reformatted version of paper published as

Andrej Brodnik, Svante Carlsson, Michael L. Fredman, Johan Karlsson and J. Ian Munro, *Worst Case Constant Time Priority Queue.* In *Journal of System and Software*, 78(3):249-256, December 2005.

Worst Case Constant Time Priority Queue

Andrej Brodnik [*†‡] Svante Carlsson [§]
Michael L. Fredman [¶] Johan Karlsson [*] J. Ian Munro [‖]

Abstract

We present a new data structure of size $3M$ bits, where M is the size of the universe at hand, for realizing a *discrete priority queue*. When this data structure is used in combination with a new memory topology it executes all discrete priority queue operations in O(1) worst case time. In doing so we demonstrate how an unconventional, but practically implementable, memory architecture can be employed to sidestep known lower bounds and achieve constant time performance.

1 Introduction

In this paper we reexamine the well known "discrete priority queue" problem of van Emde Boas et al. [18]. Operating over the bounded universe of integers $\mathcal{M} = [0, .., M-1]$, the usual operations of $Insert$ and $ExtractMin$ are supported, as are the additional operations of finding and removing any value, and finding $Predecessor(e)$ and $Successor(e)$. The last two operations determine, respectively, the largest element present that is less than e, and the smallest greater than e. The problem is referred to by Mehlhorn et al. [14] as the union-split-find problem. Under this terminology, one thinks of $[0, .., M-1]$ as being partitioned into subranges that can be further subdivided or merged, and that one can ask for the subrange containing a given value. We revert to the priority queue analogy and terminology of van Emde Boas et al. and more formally define the data type as:

Definition 4.1 *The* discrete extended priority queue problem *is to maintain a set, $\mathcal{N}$ of size N, with elements drawn from an ordered bounded universe*

[*]Department of Computer Science and Electrical Engineering, Luleå University of Technology, SE-97187 Luleå, Sweden

[†]Department of Theoretical Computer Science, Institute of Mathematics, Physics, and Mechanics, 1111 Ljubljana, Slovenia

[‡]Faculty of Education, University of Primorska, Muzejski trg 2, SI-6000 Koper, Slovenia

[§]Blekinge Institute of Technology, SE-37179 Karlskrona, Sweden

[¶]Department of Computer Science, Rutgers University, New Brunswick, NJ

[‖]School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1

$\mathcal{M} = [0..M-1]$, *and support the following operations:*

`Insert`(e)	$\mathcal{N} := \mathcal{N} \cup \{e\}$
`Delete`(e)	$\mathcal{N} := \mathcal{N} \backslash \{e\}$
`Member`(e)	*Return whether* $e \in \mathcal{N}$
`Min`	*Find the smallest element of* $\mathcal{N}$
`Max`	*Find the largest element of* $\mathcal{N}$
`DeleteMin`	*Delete the smallest element of* $\mathcal{N}$
`DeleteMax`	*Delete the largest element of* $\mathcal{N}$
`Predecessor`(e)	*Find the largest element of* $\mathcal{N}$ *less than* e
`Successor`(e)	*Find the smallest element of* $\mathcal{N}$ *greater than* e

We will refer to the predecessor of an element e as its *left neighbor* and the successor as its *right neighbor*. When talking about the *neighbors* of e we mean the left and the right neighbor. The *static* version of the problem does not include the `Insert, Delete, DeleteMin`, and `DeleteMax` operations, instead the set may be preprocessed. We let m denote $\lg M$.

1.1 Lower Bounds and some Matching Upper Bounds

Under the pointer machine model (cf. [17]) Mehlhorn et al. [14] proved a lower bound of $\Omega(\lg \lg M)$ for the discrete priority queue problem. The stratified tree by van Emde Boas et al. provides a matching upper bound [18]. More recently, Beame and Fich [3] gave a lower bound, for the static version, of $\Omega(\min((\lg \lg M / \lg \lg \lg M), \sqrt{\lg N / \lg \lg N}))$, when restricting the memory usage to $N^{O(1)}$, under the communication game model (cf. [15, 20]) which also applies to the cell probe (cf. [21]) and RAM (cf. [19]) models. This lower bound implies the same lower bound for the dynamic version. They also gave a matching upper bound, in the RAM model and hence the communication game and the cell probe models, for the static version of the problem. Andersson and Thorup [2] gave a data structure and an algorithm with $O(\sqrt{\lg N / \lg \lg N})$ worst case time for the dynamic version. For the static case, Brodnik and Munro [6] gave a data structure with $O(1)$ worst case time for any N but using $O(M)$ space. On the other hand, Ajtai et al. [1] presented a solution using only $O(N)$ space when $N = O(M^{1/p})$.

1.2 Model of Computation

Our goal is to sidestep these lower bounds, but to retain a practically implementable model of computation. Our model is based on the RAM model of computation which includes branching and the arithmetic operations addition and subtraction. We will also need bitwise boolean operations and multiplication (cf. MBRAM [19]). However, we do not want the model to be unrealistic and therefore we restrict the model to only use bounded registers. The registers we use are at least m bits wide; i.e. a given memory location can store at least m-bit values and all operations are defined for arguments with at least m bits. For fixed m, this model of computation is implemented by any standard computer today.

Operations to search for *Least (Most) Significant Bit (LSB, MSB)* in a register can be implemented to run in $O(1)$ time in our model, using a technique called *Word-Size-Parallelism* [4] (also, see [11]). Hence, we let these operations be defined in the model as well. Alternatively, the use of $O(M^\epsilon)$ extra bits permits such queries to be answered in constant time by simple table lookup.

The final aspect, and the crucial twist, of our model of computation is the notion of a word of memory. Under the standard model, a word is a sequence of bits and each bit is in one word only. We will consider a model in which a single bit may be in several different words. The notion of a "random access machine with byte overlap", *RAMBO*, was introduced by Fredman and Saks [10]. The way the bits occur, in the part of the memory where bytes overlap, has to be specified as part of the model variant.

We consider a variant which we refer to as *Yggdrasil* (see Sect. 2.3). The part of the memory where bytes overlap has been developed in hardware by *Priqueue AB*, as a SDRAM memory module according to the PC100 standard [13].

2 The Split Tagged Tree

In this section we introduce an abstract data structure *Split Tagged Tree (STT)* used to solve the discrete extended priority queue problem. We first define the STT and describe its properties, and later use these properties to implement the operations from Definition 4.1.

2.1 The Split Tagged Tree and its properties

In a complete binary tree that has leaves for every element in a universe $\mathcal{M}$ (cf. *trie* with leaves $0..M-1$ numbered from left to right, and leaves corresponding to elements of $\mathcal{N}$ are "tagged") we define:

Definition 4.2 *An internal node is a* splitting node *if there is at least one tagged leaf in each of its subtrees. The splitting node ν is a* left splitting node *of e if e is a leaf in the left subtree of ν. The first left splitting node on the path from e to the root is the* lowest left splitting node *of e.* Right splitting nodes *and the* lowest right splitting node *of e are defined symmetrically.*

Note that the splitting nodes are the nodes that appear in a path compressed trie (cf. PATRICIA trie [16]).

The splitting nodes (black nodes in Fig. 4.1) are the only internal nodes of the STT that store additional information. In detail:

Definition 4.3 *A* Split Tagged Tree *is a complete binary tree on $\mathcal{M}$ in which:*

- *Each leaf representing an element of $\mathcal{N}$ is tagged.*
- *A splitting node is tagged and has additional references to the leaves representing the largest element x in its left subtree and the smallest element y in the right subtree, where $x, y \in \mathcal{N}$.*

- *There is a special supernode, placed on top of the tree. This supernode is tagged if the set contains at least one element. If the supernode is tagged, it contains references to the leaves representing the maximum and minimum elements.*

There are two key aspects of this definition. First, note that a splitting node has references to its "inside" tagged leaves — the leftmost leaf in the right subtree and rightmost leaf in the left subtree. These two leaves are actually neighbors. Second, we consider the supernode as both a left and a right splitting node for all the elements in the universe. Hence, all elements, even minimum and maximum, have both left and right splitting nodes. This actually simplifies the operations on the STT because the supernode has a role similar to the sentinel in a linked list.

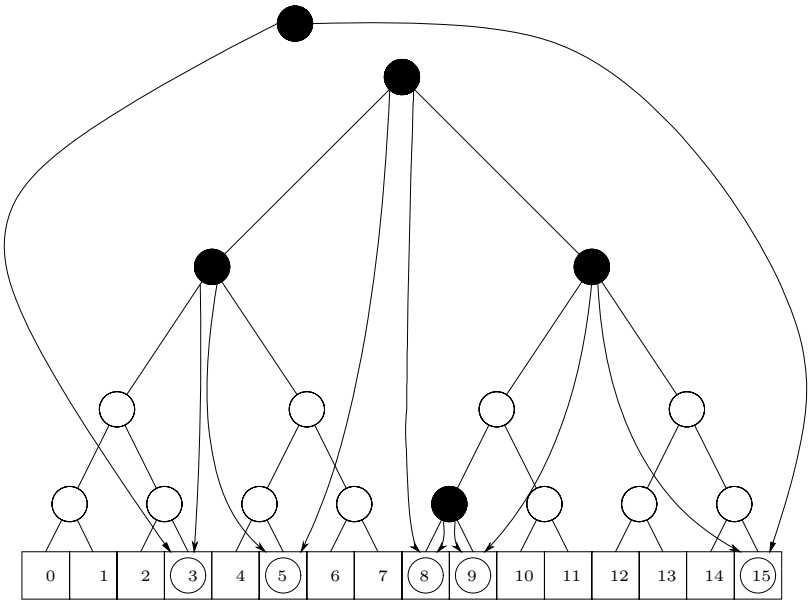


Figure 4.1: A Split Tagged Tree for $\mathcal{N} = \{3, 5, 8, 9, 15\}$ and $M = 16$.

Since leaves represent elements of $\mathcal{M}$, we will refer to the leaves and the corresponding elements interchangeably.

The only information required in a leaf is its tag, hence a Boolean array suffices for the leaves. As a consequence, references to leaves are simple indices into the array, or simply the element value. On the other hand, the internal nodes and the supernode consist of two references and a tag. The nodes can be stored in an array, in standard heap order (the root maps to location 1 and the children of the node mapping to location i are mapped to locations $2i$ and $2i + 1$). The supernode is stored at the location 0.

The split tagged tree is somewhat similar to the *stratified tree* presented by van Emde Boas et al. [18]. A key feature of the stratified tree is that nodes are tagged in a manner such that the lowest tagged ancestor of a tagged leaf can be found by a binary search. This property does not need to hold for the STT as we will use a novel memory architecture to permit constant time search and still find the analogous nodes. The STT does, however, have a number of crucial properties.

Lemma 4.1 *Let $e \in \mathcal{N}$ and let ν_l and ν_r be its lowest left and lowest right splitting nodes respectively. Then, ν_l is the only left splitting node of e that refers to e, and ν_r is the only right splitting node of e that refers to e.*

Proof: The proof is given in terms of the left splitting node. The proof for the right splitting node is symmetric.

First, we show that ν_l refers to e. From Definition 4.3 we know that an element is referred to by a splitting node if it is the largest element in the left subtree of the node. Thus, it suffices to demonstrate that e is the largest element of $\mathcal{N}$ in the left subtree of ν_l. Assume to the contrary that an element $z \in \mathcal{N}$ exists such that $z \neq e$ is the largest element in the left subtree of ν_l. Since both e and z are in the left subtree of ν_l, the lowest common ancestor ν of e and z, is also in the left subtree of ν_l (see Fig. 4.2). By Definition 4.2, ν is a splitting node. Moreover, since $e < z$, ν is a left splitting node of e which contradicts our initial assumption that ν_l is the lowest left splitting node of e.

Finally, no other left splitting node ν' of e (necessarily higher) can refer to e since the right subtree of ν_l contains an element $z' \in \mathcal{N}$ (see Fig. 4.2). The element z' has prior claim to being referred to by ν' since $e < z'$. $\mathcal{QED}$

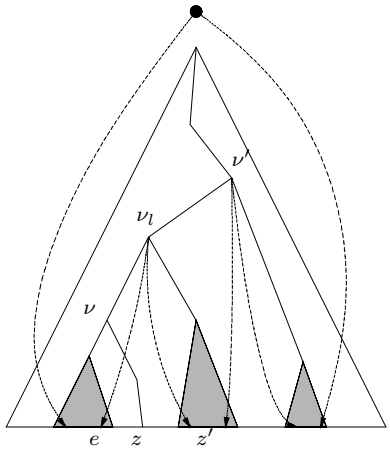


Figure 4.2: The element e is only referenced by the lowest left splitting node ν_l of its left splitting nodes ($e, z' \in \mathcal{N}, z \notin \mathcal{N}$).

Lemma 4.1 states that in the STT there are exactly two nodes that have references to an element in $\mathcal{N}$, namely, its lowest left and lowest right splitting nodes. We proceed by showing how to find the neighbors in $\mathcal{N}$ of an arbitrary leaf e of $\mathcal{M}$.

Lemma 4.2 *Let ν_l and ν_r be the lowest left and lowest right splitting nodes of $e \in \mathcal{M}$ respectively. Further, let $x \in \mathcal{N}$ be the left neighbor of e, hence $x < e$. Then, if $e \in \mathcal{N}$ a reference in ν_r refers to x, and if $e \notin \mathcal{N}$ then either a reference in ν_l or in ν_r refers to x.*

If we instead let $x \in \mathcal{N}$ be the right neighbor of e, then if $e \in \mathcal{N}$ a reference in ν_l refers to x, and if $e \notin \mathcal{N}$ then either a reference in ν_l or in ν_r refers to x.

Proof: Again we focus on the left neighbor and the proof for the right neighbor is symmetric.

If $e \in \mathcal{N}$, the lowest common ancestor ν of x and e is a splitting node and has references to both x and e, since x and e are neighbors. The node ν is a right splitting node of e since $x < e$. Since ν_r cannot be above ν, ν_r must also refer to x. But then Lemma 4.1 implies that ν and ν_r are the same.

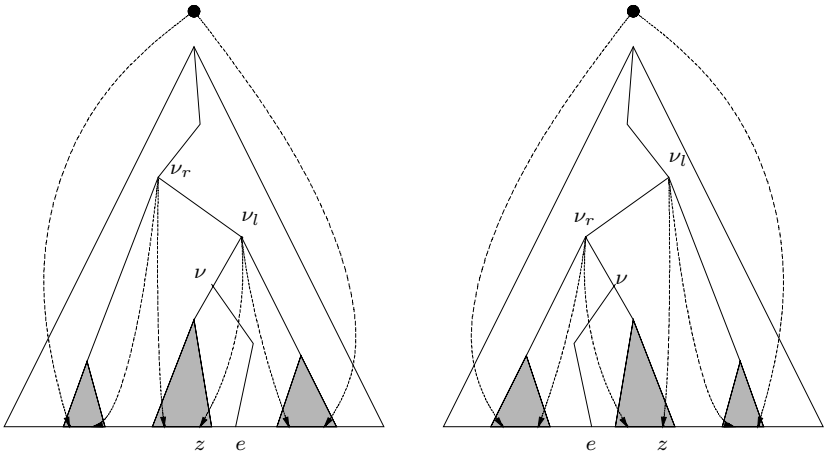
If $e \notin \mathcal{N}$, let $y \in \mathcal{N}$ be the right neighbor of e, hence $x < e < y$, and let ν be the lowest common ancestor of x and y. By definition, ν is also a splitting node, and since $x < e < y$ it is a splitting node on the path from e to the root. Since x and y are neighbors in $\mathcal{N}$, x is the largest element of $\mathcal{N}$ in the left subtree of ν and y is the smallest element of $\mathcal{N}$ in the right subtree of ν. Consequently, ν has a reference to x, which is the left neighbor of e. Now assume that ν is a right splitting node of e. Since ν_r cannot be above ν, ν_r must also refer to x. But then Lemma 4.1 implies that ν and ν_r are the same. Similarly, if ν is a left splitting node it has to be ν_l. $\mathcal{QED}$

Finally, we show that for the insertion of an element e, it is sufficient to find either the lowest left or the lowest right splitting node of e, to decide where the new splitting node shall be (see Fig. 4.3). We start with the lemma in terms of the left splitting node:

Lemma 4.3 *Let ν_l be the lowest left splitting node of $e \notin \mathcal{N}$ and let $z \in \mathcal{N}$ be the largest element in the left subtree of ν_l. Let ν be the lowest common ancestor of z and e. If $z < e$ then the right subtree of ν is empty, and if $e < z$ then the left subtree of ν is empty.*

Proof: For the case $z < e$ (see Fig. 3(a)), assume there is an element $g \in \mathcal{N}$ in the right subtree of ν. Then $z < g$ since z is in the left subtree of ν. Since z and e are in the left subtree of ν_l, so is ν, and therefore g is likewise. But since $g > z$, this contradicts the assumption that z is the largest element of $\mathcal{N}$ in the left subtree of ν_l and hence there is no element in the right subtree of ν.

For the case $e < z$ (see Fig. 3(b)), assume an element $g \in \mathcal{N}$ exists in the left subtree of ν. Then, by Definition 4.2, ν is a splitting node. Since e and z are in the left subtree of ν_l, ν has to be in the left subtree of ν_l. Hence ν is lower than ν_l and ν is a left splitting node of e contradicting the assumption about ν_l. Consequently, there is no element in the left subtree of ν. $\mathcal{QED}$



(a) case $z < e$ (b) case $e < z$

Figure 4.3: The two scenarios before insertion of e.

Lemma 4.4 *Let ν_r be the lowest right splitting node of $e \notin \mathcal{N}$ and let $z \in \mathcal{N}$ be the smallest element in the right subtree of ν_r. Let ν be the lowest common ancestor of z and e. If $z < e$ then the right subtree of ν is empty, and if $e < z$ then the left subtree of ν is empty.*

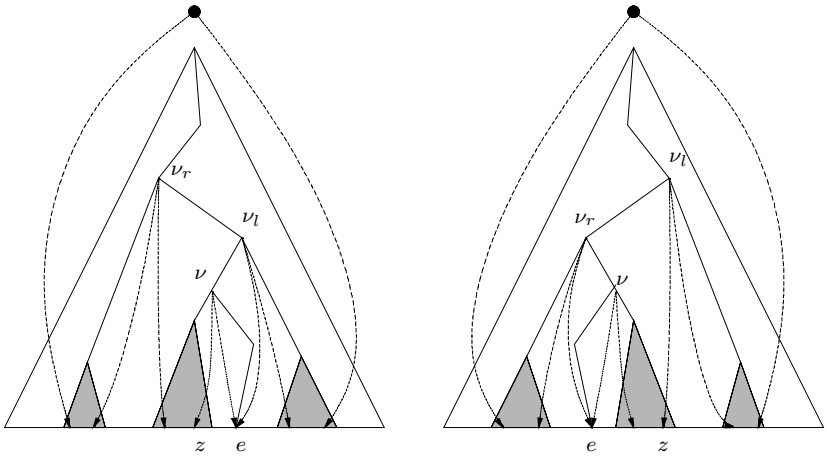
Proof: This proof is symmetric to the proof of Lemma 4.3. $\mathcal{QED}$

2.2 Operations on the Split Tagged Tree

We proceed to describe how to answer queries and perform updates in the STT. Predecessor and successor queries can be answered using Lemma 4.2. In a binary tree there are always $n-1$ internal nodes when there are n leaves. In the STT there are as many internal splitting nodes as there are internal nodes in a binary tree and since we have the supernode as an additional splitting node we have n splitting nodes when there are n leaves. Hence, at an *insertion* of element $e \notin \mathcal{N}$, exactly one internal node becomes a splitting node and one leaf gets tagged.

The leaf is that corresponding to e. To find the new splitting node, we get the largest element z in the left subtree of the lowest left splitting node ν_l of e. Let ν be the lowest common ancestor of e and z. If $z < e$ (see Fig. 3(a)) then, by Lemma 4.3, ν should be the new lowest right splitting node of e with one reference to z and one to e, and ν_l should be updated to refer to e instead of z. If $e < z$ (see Fig. 3(b)) then, by Lemma 4.3, ν should be the new lowest left splitting node of e with one reference to e and one to the right neighbor of e. The node ν_r should be updated to refer to e instead of e's right neighbor (see

Fig. 4.4).



(a) case $z < e$ (b) case $e < z$

Figure 4.4: The result after insertion of e in the two scenarios in Fig. 4.3.

By a similar reasoning we see that *deletion* is done by removing the tags in the leaf and in the lower of its lowest left and right splitting nodes. The references in the other splitting node should be updated to refer to the neighbors of e.

Since only a constant number of nodes need to be updated (and hence only a constant number of references), the time to update (and also to search) the STT is asymptotically bounded by the time to traverse the tree. The tree is of height $\lg M + 1$, which implies that the time to update and search the tree is $O(\lg M)$ in our model.

2.3 The Yggdrasil variant of RAMBO

We have worked our way to a situation that would appear to require a sequential scan of the nodes up a path. However, if we can find the lowest splitting nodes and the lowest common ancestor in constant time, then we can perform all the operations from Definition 4.1 in constant time. At this point we resort to a change in the model model of computation and consequently an improvement in the hardware to achieve a constant time solution.

The RAMBO model of computation is a RAM model in which, in one part of the memory, registers can share bits, i.e. bytes overlap (cf. RAMBO introduced by Fredman and Saks [10] and further described by Brodnik [5]). One particular variant of the RAMBO which we have called *Yggdrasil* can help us to solve our problem. (Yggdrasil according to Norse mythology is the great ash tree that

holds together heaven, hell and earth with its roots and branches — a task clearly requiring a nonstandard architecture, see [8].)

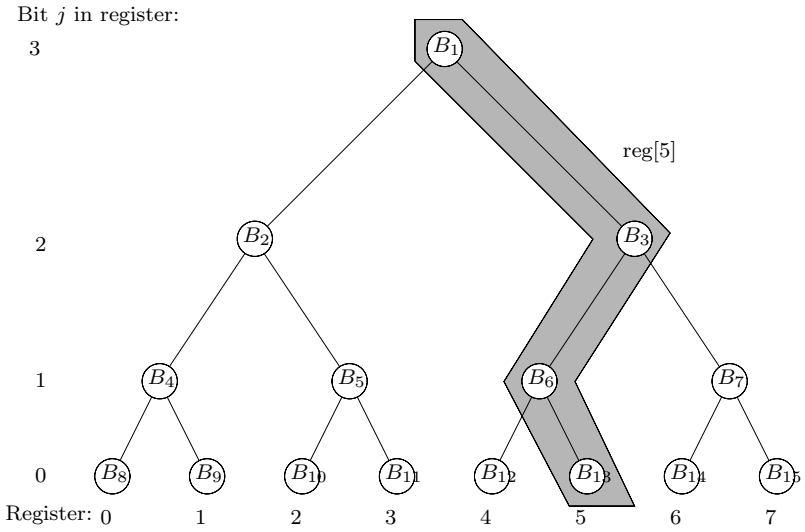


Figure 4.5: Overlapped memory *Yggdrasil*.

In the Yggdrasil memory layout, registers overlap as paths from leaf to root in a complete binary tree (see Fig. 4.5). In particular, we think of the bits B_k where $k = 1, .., M-1$ as being enumerated in standard heap order. (The root is B_1, the children of B_k are B_{2k} and B_{2k+1}, and the leaves are bits $B_{M/2}$ through B_{M-1}.) The most significant bit of any register is the root bit, B_1. (By convention, we call this bit $m-1$, and so, bit 0 is the least significant of a register.) The bits of register i correspond to those along the path from ith leaf (i.e. bit $M/2+i$) to the root. This means:

$$\begin{aligned} reg[i].bit[j] &= B_k \qquad \text{where} \\ k &= \left\lfloor \frac{i}{2^j} \right\rfloor + 2^{m-j-1} \ . \end{aligned} \tag{1}$$

Note that the path for element e corresponds to register `reg[i]`, where $i = \lfloor e/2 \rfloor$. We now store the tags of the internal nodes in the Yggdrasil memory, while an array of heap ordered internal nodes, storing the references, and a boolean array of tags of the leaves remain in regular memory.

To find the lowest splitting node ν_k of the element e we read the register `reg[i]`, where $i = \lfloor e/2 \rfloor$. We then find the least significant set bit j of `reg[i]` and compute k using Eq. 1. The element e is in the right subtree of a node ν_k if the corresponding bit j of the binary representation of e is 1. Therefore, we can use e and mask the value of register `reg[i]` to get the right splitting nodes of e only. Hence, we can find the lowest right splitting node in constant time.

Symmetrically, we can use the bitwise binary negation, $\overline{e}$, of e to find the lowest left splitting node.

The lowest common ancestor ν_k of two elements e and $f \in \mathcal{M}$ is the last common node on the paths from the root to e and f respectively. In the binary representation of e and f, this corresponds to the most significant bit j that differs between e and f. Eq. 1 then gives

$$\begin{aligned} k &= \left\lfloor \frac{i}{2^j} \right\rfloor + 2^{m-j-1} \\ &= \left\lfloor \frac{\lfloor e/2 \rfloor}{2^j} \right\rfloor + 2^{m-j-1} \\ &= \left\lfloor \frac{e}{2^{j+1}} \right\rfloor + 2^{m-j-1} \text{ where} \qquad (2) \\ j &= \text{MSB}(e \text{ XOR } f) \ . \qquad (3) \end{aligned}$$

Now we can find both the lowest splitting nodes of an element and the lowest common ancestor of two elements in constant time. Following references and updating those and the tags can also be done in constant time. Hence, given Yggdrasil memory, updates and queries can be performed in constant time.

2.4 Saving memory

The STT contains a substantial amount of redundant information. First, the leaves can be removed since a leaf is tagged if and only if it is referred to by an internal node. Hence, membership can be determined by finding the lowest splitting node ν of the element e. The element e is a member of $\mathcal{N}$ if and only if ν contains a reference to e. Next, the information in the internal nodes can be stored using references of variable length. A reference in a node ν_k on level j (levels are counted, starting at 0 from the lowest level of internal nodes, increasing towards the root, i.e., the root is at level $m-1$) needs only j bits instead of m bits. The node ν_k covers 2^{j+1} leaves, but only 2^j leaves are in each of the subtrees, and hence a leaf can be identified using j bits. The $j+1$st bit is 1 if the reference is to the right subtree and 0 otherwise. The remaining $m-j-1$ most significant bits are identical to the $m-j-1$ most significant bits of e, where e is the element used to find the internal node ν_k. The space needed to store the internal nodes (excluding the supernode) is $\sum_{j=0}^{m-1} 2^{m-1-j} \cdot 2 \cdot j$. If we substitute with $j_1 = m-1-j$, we get

$$\begin{aligned} \sum_{j_1=0}^{m-1} 2^{j_1} \cdot 2 \cdot (m-1-j_1) &= 2(\sum_{j_1=0}^{m-1} (m-1)2^{j_1} - \sum_{j_1=0}^{m-1} j_1 2^{j_1} \\ &= 2((m-1)(2^m-1) - \sum_{j_1=0}^{m-1} j_1 2^{j_1}) \ . \end{aligned}$$

Now we can use $\sum_{j=0}^{n} j2^j = (n-1)2^{n+1}+2$ (see [12, page 33]) and finally get

$$\begin{aligned} 2((m-1)(2^m-1) - \sum_{j_1=0}^{m-1} j_1 2^{j_1}) &= 2((m-1)(2^m-1) - ((m-2)2^m+2)) \\ &= 2\cdot 2^m - 2m - 2 \ . \end{aligned}$$

Hence, our data structure is using $2\cdot 2^m - 2m - 2$ bits instead of $2(2^m-1)m$ bits to store the references in the internal nodes. We also need to store the references in the supernode and use $2m$ bits for that. These two improvements reduce the space requirements from $2M \lg M + M$ bits to $2M$ bits of conventional memory for the discrete extended priority queue problem. We must admit, however, that reducing the size does increase the, still constant, runtime.

The above discussion leads to our main result:

Theorem 4.1 *Using the* Split Tagged Tree *together with the* Yggdrasil *memory we can solve the discrete extended priority queue problem in* $O(1)$ *worst case time per operation using* $2M$ *bits of ordinary memory and* M *bits of Yggdrasil memory.*

To solve the problem with support for either predecessor or successor, but not both, we remove the reference to one or the other element in the nodes. This saves half of the ordinary memory and reduces the constant for update times.

More precisely, if we want to support predecessor queries only, we keep, in internal nodes, the reference to the largest element in the left subtree and remove the reference to the smallest element in the right subtree. Then the condition on the lowest left splitting node of Lemma 4.1 still holds since the proof does not use the reference to the smallest element in the right subtree. Further, the condition on the left neighbor of Lemma 4.2 also holds for the same reason. Similarly, Lemma 4.3 holds as well. Hence, the discussion of how to find predecessor and perform updates in Sect. 2.2 holds if we simply omit the parts about the reference to the smallest element in the right subtree.

Similarly, if we want to support successor queries only, we keep, in the internal nodes, the reference to the smallest element in the right subtree and remove the reference to the largest element in the left subtree. Instead of Lemma 4.3 we use Lemma 4.4 when looking for the new splitting node. Hence:

Corollary 4.1 *Using the* Split Tagged Tree *together with the* Yggdrasil *memory we can solve a variant of the discrete extended priority queue problem, with support for either* `Min`, `DeleteMin` *and* `Successor` *or* `Max`, `DeleteMax` *and* `Predecessor`, *in* $O(1)$ *worst case time per operation using* M *bits of ordinary memory and* M *bits of Yggdrasil memory.*

When $N = o(M/\lg M)$, we can reduce the space even further by using perfect hashing (see [7, 9]) to store tagged internal nodes. Each tagged node contains two references of size $\lg M$ and we store N of M nodes using a hash table. This gives a space requirement of $O(N \lg M)$ bits of ordinary memory.

Queries can still be performed in $O(1)$ worst case time while updates can be performed in $O(1)$ amortized expected time. This gives us an improved result from Theorem 4.1:

Theorem 4.2 *Using the* Split Tagged Tree *together with the* Yggdrasil *memory we can solve the discrete extended priority queue problem using $O(N \lg M)$ bits of ordinary memory and M bits of Yggdrasil memory. Query operations can be performed in $O(1)$ worst case time and update operations can be performed in $O(1)$ amortized expected time.*

From Theorem 4.2, using the same reasoning as for Corollary 4.1 we also get:

Corollary 4.2 *Using the* Split Tagged Tree *together with the* Yggdrasil *memory we can solve a variant of the discrete extended priority queue problem, with support for either* `Min`, `DeleteMin` *and* `Successor` *or* `Max`, `DeleteMax` *and* `Predecessor` *using $O(N \lg M)$ bits of ordinary memory and M bits of Yggdrasil memory where query operations can be performed in $O(1)$ worst case time and update operations can be performed in $O(1)$ amortized expected time.*

Acknowledgment

We thank the anonymous reviewers for helpful comments.

References

[1] M Ajtai, M Fredman, and J Komlós. Hash functions for priority queues. *Inf. Control*, 63(3):217–225, 1986.

[2] Arne Andersson and Mikkel Thorup. Tight(er) worst-case bounds on dynamic searching and priority queues. In *Proceedings of the 32nd Annual ACM Symposium on Theory of Computing*, pages 335–342. ACM Press, May 21–23 2000.

[3] P. Beame and F. E. Fich. Optimal bounds for the predecessor problem and related problems. *Journal of Computer and System Sciences*, 65(1):38–72, 2002.

[4] A. Brodnik. Computation of the least significant set bit. In *Proceedings Electrotechnical and Computer Science Conference*, volume B, pages 7–10, Portorož, Slovenia, 1993.

[5] Andrej Brodnik. *Searching in Constant Time and Minimum Space (*Minimæ Res Magni Momenti Sunt*)*. PhD thesis, University of Waterloo, Waterloo, Ontario, Canada, 1995. (Also published as technical report CS-95-41.).

[6] Andrej Brodnik and J. Ian Munro. Neighbours on a grid. In R. Karlsson and A. Lingas, editors, *SWAT '96, 5th Scandinavian Workshop on Algorithm Theory*, volume 1097 of *Lecture Notes in Computer Science*, pages 307–320. Springer, 3–5July 1996.

[7] Martin Dietzfelbinger, Anna Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert, and Robert E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal on Computing*, 23(4):738–791, 1994.

[8] Encyclopedia Mythica. Yggdrasil. `http://www.pantheon.org/articles/y/yggdrasil.html`.

[9] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *Journal of the ACM*, 31(3):538–544, July 1984.

[10] Michael L. Fredman and Michael E. Saks. The cell probe complexity of dynamic data structures. In *Proceedings of the 21st Annual ACM Symposium on Theory of Computing*, pages 345–354. ACM Press, May 14–17 1989.

[11] Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47(3):424–436, December 1993.

[12] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics – A Foundation for Computer Science*. Addison–Wesley, Reading, MA, 1988.

[13] Roni Leben, Marijan Miletić, Marjan Špegel, Andrej Trost, Andrej Brodnik, and Johan Karlsson. Design of high performance memory module on PC100. In *Proceedings Electrotechnical and Computer Science Conference*, pages 75–78, Slovenia, 1999.

[14] Kurt Mehlhorn, Stefan Näher, and Helmut Alt. A lower bound on the complexity of the union-split-find problem. *SIAM Journal on Computing*, 17(6):1093–1102, 1988.

[15] Peter Bro Miltersen. Lower bounds for Union-Split-Find related problems on random access machines. In *Proceedings of the 26th Annual ACM Symposium on Theory of Computing*, pages 625–634. ACM Press, 23–25 May 1994.

[16] D. R. Morrison. PATRICIA - practical algorithm to retrieve information coded in alphanumeric. *Jrnl. A.C.M.*, 15(4):514–534, October 1968.

[17] Arnold Schönhage. Storage modifications machines. *SIAM Journal on Computing*, 9(3):490–508, August 1980.

[18] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977.

[19] Peter van Emde Boas. Machine models and simulations. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A: Algorithms and Complexity, chapter 1, pages 3–66. Elsevier/MIT Press, Amsterdam, 1990.

[20] A. Yao. Some complexity questions related to distributive computing. In *Proceedings of the 11th Annual ACM Symposium on Theory of Computing*, pages 209–213, 1979.

[21] Andrew Chi-Chih Yao. Should tables be sorted? *Journal of the ACM*, 28(3):614–628, July 1981.

Paper 5

An O(1) Solution to the Prefix Sum Problem on a Specialized Memory Architecture

When you are a Bear of Very Little Brain, and you Think of Things, you find sometimes that a Thing which seemed very Thingish inside you is quite different when it gets out into the open and has other people looking at it.

Winnie the Pooh

Reformatted version of paper to appear as

Andrej Brodnik, Johan Karlsson, J. Ian Munro and Andreas Nilsson, *An O(1) Solution to the Prefix Sum Problem on a Specialized Memory Architecture.* In *IFIP 19th World Computer Congress, TC1 4th International Conference on Theoretical Computer Science*, Springer, 2006.

An O(1) Solution to the Prefix Sum Problem on a Specialized Memory Architecture

Andrej Brodnik [*†‡] Johan Karlsson [*] J. Ian Munro [§]
Andreas Nilsson [*]

Abstract

In this paper we study the Prefix Sum problem introduced by Fredman. We show that it is possible to perform both update and retrieval in $O(1)$ time simultaneously under a memory model in which individual bits may be shared by several words. We also show that two variants (generalizations) of the problem can be solved optimally in $\Theta(\lg N)$ time under the comparison based model of computation.

1 Introduction

Models of computation play a fundamental role in theoretical Computer Science, and indeed, in the subject as a whole. Even in modeling a standard computer, the random access machine (RAM) model has been subject to refinements which more realistically model cost or, as in this paper, suggest feasible extensions to the model that permit more efficient computation, at least for some problems. Work taking into account a memory hierarchy, either when memory and page sizes are known (cf. [2]) or not (cf. [11]) is an example of the former. Taking into account parallelism, as in the PRAM model (cf. [17, 26]), is an obvious example of the latter. More subtle examples include the recent result that the operations of an arbitrary finite Abelian group can be carried out in constant time (We assume a word of memory is adequate to hold the size of the group.) provided one can reverse the bits of a word in constant time [8]. This argues for a more robust set of operations. Here we deal with the way a single level memory is organized and demonstrate that the power of a machine can be increased if we permit individual bits to occur in several words simultaneously. This Random Access Machine with Byte Overlap (RAMBO) was first suggested by Fredman and Saks [10] and subsequently used by Brodnik et al. [6] and Brodnik and Iacono [7]. Indeed it is shown in the latter two papers that a priority queue of word sized objects can be maintained in constant time under a particular form

[*]Luleå University of Technology, Sweden
[†]University of Primorska, Slovenia
[‡]Institute of Mathematics, Physics, and Mechanics, Slovenia
[§]Cheriton School of Computer Science, University of Waterloo, Ontario, Canada

of the RAMBO model, whereas Beame and Fich [3] and Brodnik and Iacono [7] have both shown lower bounds on the problem under various forms of the RAM model.

Here we discuss solutions to variants of the *Prefix Sum problem* (i.e. finding the sum of the first j elements in an array and also updating these values) which was introduced by Fredman [9]. Various lower bounds have been proven for the problem. We, however, focus on the problem under a nonstandard, though very feasible, model to achieve a constant time solution.

Fredman and Saks actually suggested the RAMBO model in connection with the Prefix Sum problem. They claim, with no hint of how it may be done, that Prefix Sum mod 2 can be solved in constant time under the model. We show how this can be done not only for Prefix Sum mod 2 but for Prefix Sum modulo an arbitrary universe size $M \leq 2^{\Theta(b/n)}$ where b is the word size, $n = \lceil \lg N \rceil$ and N is the size of the array.

The RAMBO model, besides the usual RAM operations (cf. [27]), also has a part of memory where a bit may occur in several registers or in several positions in one register. The way the bits occur in this part of the memory has to be specified as part of the model. One example of such a memory variant is a square of bits with b rows and b columns. A b-bit word can be fetched either as a row or a column. In such a memory each bit can be accessed either by the row word or the column word.

The form of RAMBO used by Brodnik et al. [6] to solve the priority queue problem in O(1) worst case time makes use of words corresponding to the leaves of a balanced binary tree. Each node of the tree contains a flag bit and each such word contains the flags along the root to leaf path, so, for example, the flag at the root is in all of these words. The specific architecture was called *Yggdrasil* after the giant ash tree linking the worlds in Norse mythology. That variant has been implemented in hardware [18] and the actual rerouting of the bits on a word fetch is not difficult. In this paper we modify the Yggdrasil variant slightly and solve the Prefix Sum problem. This gives further evidence of the value of such an architecture, at least for a special purpose processor.

Now let us formally define the Prefix Sum problem:

Definition 5.1 *The* Prefix Sum problem *is to maintain an array, $\mathcal{A}$, of size N, and to support the following operations:*

`Update(`j`, Δ)` $\mathcal{A}(j) := \mathcal{A}(j) + \Delta$

`Retrieve(`j`)` *return* $\sum_{i=0}^{j} \mathcal{A}(i)$

where $0 \leq j < N$.

Fredman showed that, under the comparison based model of computation, an $O(\lg N)$ solution exists for the Prefix Sum problem [9].

The problem can be generalized in several ways and we start by adding another parameter, k to the `Retrieve` operation. This parameter is used to tell the starting point of the array interval to sum over. Hence, `Retrieve(k,j)` returns $\sum_{i=k}^{j} \mathcal{A}(i)$, where $0 \leq k \leq j < N$. This variant is usually referred to as the *Partial Sum* or *Range Sum problem*. The Partial Sum problem can be solved using a solution to the Prefix Sum problem (`Retrieve(k,j) =`

`Retrieve(j) - Retrieve(k-1)`). In fact, the two problems are often used interchangeably.

Furthermore, there is no obvious reason to only allow addition in the `Update` and `Retrieve` operations. We can allow any binary function, $\oplus$, to be used. In fact we can allow the `Update` operation to use one function, $\oplus_u$, and the `Retrieve` operation to use another function, $\oplus_r$. We will refer to this variant of the problem as the *General Prefix Sum problem.*

Moreover, one can allow array position to be inserted at or deleted from arbitrary places. Hence, we can have sparse arrays, e.g. an array where only $\mathcal{A}(5)$ and $\mathcal{A}(500)$ are present. Positions which have not yet been added or have been deleted have the value 0. We refer to this variant as the *Dynamic Prefix Sum problem.* Brodnik and Nilsson [21, pp 65-80] describe a data structure they call a BinSeT tree which can be modified slightly to support all operation of the Dynamic Prefix Sum problem in $O(\lg N)$ time.

The *Searchable Partial Sum* problem extends the set of operations with a `select(j)` operation which finds the smallest i such that $\sum_{k=0}^{i} \mathcal{A}(k) \geq j$ [23]. Hon et al. consider the Dynamic version of the Searchable Partial Sum problem [16]. Another generalization is to use multidimensional arrays and this variant has been studied by the data base community [4, 12, 13, 15, 24, 25].

Several lower bounds have been presented for the Prefix Sum problem: Fredman showed a $\Omega(\lg N)$ algebraic complexity lower bound and a $\Omega(\lg N/\lg\lg N)$ information-theoretic lower bound [9]. Yao [29] has shown that $\Omega(\lg N/\lg\lg N)$ is an inherent lower bound under the semi-group model of computation and this was improved by Hampapuram and Fredman to $\Omega(\lg N)$ [14]. We side step these lower bounds by considering the RAMBO model of computation [5, 10].

As with all RAM based model we need to restrict the size of a word which can be stored and operated on. We denote the word size with b and assume that b is an integer power of 2 which is true for most computers today. A bounded word size also implies a bounded universe of elements that we store in the array. We use M to denote the universe size. Hence all operations $\oplus$ have to be computed modulo M and we require that each of the operands and the result are stored in one word.

We will use n and m to denote $\lceil \lg N \rceil$ and $\lceil \lg M \rceil$ respectively. Hence, $N \leq 2^n$ and $M \leq 2^m$. Both n and m are less than or equal to b, $(n, m \leq b)$. In one of the solutions we actually require that $nm \leq b$.

In Sect. 2 we show a $O(1)$ solution to the Prefix Sum problem under the RAMBO model using a modified Yggdrasil variant. In Sect. 3 we discuss a $O(\lg N)$ solution to the General and Dynamic Prefix Sum problems and finally conclude the paper with some open questions in Sect. 4.

2 An $O(1)$ Solution to the Prefix Sum Problem

In our O(1) solution to the Prefix Sum problem we use a complete binary tree on top of the array (Fig. 5.1). We label the nodes in standard heap order, i.e., the root is node ν_1 and the left and right children of a node ν_i are ν_{2i} and ν_{2i+1}

respectively. In each node we store m bits representing the sum of the leaves in the left subtree. Since we build a complete binary tree on top of the array we assume that $N = 2^n$ (if this is not true we still build the complete tree and in worst case waste space proportional to $N/2 - 1$). We do not store the original array $\mathcal{A}$ since its values are stored implicitly in the tree. The only value not stored in the tree (if $N = 2^n$ only) is $\mathcal{A}(N-1)$ and we store this value explicitly (`vn1`). Formally we define:

Definition 5.2 *A* N-m-tree *is a complete binary tree with N leaves in which the internal nodes (ν_i) store a m-bit value. In addition, a m-bit value is stored separately (*`vn1`*).*

To update $\mathcal{A}(j)$ (Alg. 5.1) in this structure we have to update all the nodes on the path from leaf j to the root in which j belongs to the left subtree. To `Retrieve(`j`)` (Alg. 5.2) we need to sum the values of all the nodes on the path from leaf $j+1$ to the root in which $j+1$ belongs to right subtree. Note that the path corresponding to array position j starts at node $\nu_{N/2+j/2}$.

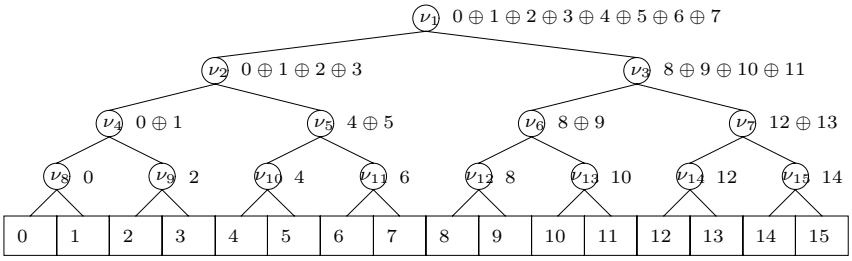


Figure 5.1: Complete binary tree on top of $\mathcal{A}$. Nodes are storing the sum of the values in the leaves covered by the left subtree.

```
update(j, Δ)
  if (j == N-1)
    vn1 = vn1 + Δ;
  else
    i = N + j;
    while (i > 1)
      next = i div 2;
      if (i mod 2 == 0)
        ν_next = ν_next + Δ mod M);
      i = next;
```

Algorithm 5.1: Updating of a N-m-tree in $O(\lg N)$ time.

The method described above implies a $O(\lg N)$ update and retrieval time in the RAM model. To achieve constant time update and retrieval we use a

```
retrieve(j)
  if (j == N-1)
    sum = vn1;
    i = N + j;
  else
    sum = 0;
    i = N + j + 1;
  while (i > 1)
    next = i div 2;
    if (i mod 2 == 1)
      sum = sum + ν_next mod M;
    i = next;
  return sum;
```

Algorithm 5.2: Retrieve in a N-m-tree in $O(\lg N)$ time.

variant of the RAMBO model similar to the Yggdrasil variant. In the Yggdrasil variant, registers overlap as paths from leaf to root in a complete binary tree with one bit stored in each internal node [6]. We generalize the Yggdrasil variant and let it store m bits in each node and call this variant *m-Yggdrasil*. In any m-Yggdrasil, register `reg[i]` corresponds to the path from node $\nu_{N/2+i}$ to the root of the tree. Each register consists of $nm \leq b$ bits. In total the m-Yggdrasil registers need $(N-1) \cdot m$ bits.

Now, we use the registers from m-Yggdrasil to store the nodes of our tree. The path corresponding to array position j is stored in `reg[j/2]` and hence all nodes along the path can be accessed at once.

We let levels of the tree be counted from the internal nodes above the leaves starting at 0 and ending with $n-1$ at the root. If the ith bit of j is 1 then j is in the right subtree of the node on level i of the path and in the left otherwise. Hence j can be used to determine which nodes along the path should be updated (nodes corresponding to bits of j that are 0) and which nodes should be used when retrieving a sum (nodes corresponding to bits of j that are 1).

When updating the m-Yggdrasil registers (Alg. 5.3), for all bits of j, if the ith bit of j is 0 we add Δ to the value of the ith node along the path from j to the root. To do this we shift Δ to the corresponding position ($\Delta << (im)$) and add to `reg[j/2]`. Instead of checking whether the ith bit of j is 0 we can mask the shifted Δ with a value based on NOT j. The value consists of, if the ith bit of NOT j is 1, m 1s shifted to the correct position and m 0s otherwise.

Actually, as long as the binary operation only affects the m bits that should be updated we can use word-size parallelism (cf. [5]) and perform the update of all nodes in parallel. In Sect. 2.1 we show that addition modulo M can be implemented affecting only m bits.

We use two functions (`dist(i)` and `mask(i)`) to simplify the description of the update and retrieve methods. The function `dist(i)`, $(0 \leq i < 2^m)$ computes nm-bit values. The values are n copies of the m bits in i. For example, given

```
update(j, Δ)
  if (j == N-1)
    vn1 = vn1 + Δ;
  else
    for (i=0; 0 < n; i++)
      if (((j >> i) AND 1) == 0)
        reg[j/2] = reg[j/2] + (Δ << (i*m));
```

Algorithm 5.3: Updating of a N-m-tree stored in m-Yggdrasil memory ($O(\lg N)$ time).

$m = 3, n = 4$ `dist(010)` is 010010010010. The function `mask(i)`, $(0 \leq i < 2^n)$ also computes nm-bit values. These values are computed as follow: bit j $(0 \leq j < n)$ of i is copied to bits $jm..(j+1)m-1$. For example, given $m = 3, n = 4$, `mask(1001)` is 111000000111. Both these functions can be implemented by using word-size parallelism [5].

We can update the tree in constant time using the procedure in Alg. 5.4. First we make n copies of Δ and then mask out the copies we need. Then finally we add the value in `reg[j/2]` and the masked distributed Δ and store the result in `reg[j/2]`. For the case when $j = N-1$ we simply add `vn1` and Δ and store it in `vn1`. This gives us the following lemma:

Lemma 5.1 *The update operation of the Prefix Sum problem can be supported in $O(1)$ when part of the N-m-tree is stored in a m-Yggdrasil memory.*

```
update(j, Δ)
  if (j == N-1)
    vn1 = vn1 + Δ;
  else
    reg[j/2] = reg[j/2] + (dist(Δ) AND mask(NOT j));
```

Algorithm 5.4: Updating of a N-m-tree stored in m-Yggdrasil memory using word size parallelism ($O(1)$ time).

To support the retrieve method in constant time we use a table `SUM[i]`, $(0 \leq i < 2^{nm})$ with m-bit values that are the sum modulo M of the n m-bit values in i.

To retrieve the sum (Alg. 5.5) we read the register `reg` corresponding to j and mask out the parts we need. Then we use the table `SUM` to calculate the sum. Finally, we add `vn1` to the sum if $j = N-1$.

The space needed by the table `SUM` is $2^{nm} \cdot m = N^{\lg M} \cdot m = M^{\lg N} \cdot m$, which is rather large. In order to reduce the space requirement we can reduce, by half, the number of bits used as index into the table. This gives us a space requirement of $\sqrt{M^{\lg N}} \cdot m$. We do this by shifting the top $n/2$ m-bit values from

```
retrieve(j)
  if (j == N-1)
    v = reg[j/2] AND mask(j);
  else
    v = reg[(j+1)/2] AND mask(j+1);
  sum = SUM[v];
  if (j == N-1)
    sum = vn1 + sum;
  return sum;
```

Algorithm 5.5: Retrieve in a N-m-tree stored in m-Yggdrasil memory using word size parallelism ($O(1)$ time).

`reg` down and computing the sum modulo M of these values and the bottom $n/2$ values. Then this new $(n/2)m$-bit value is used as index into `SUM` instead.

We can actually repeat this process until we get the m-bit we desire, and hence we do not need the table `SUM` (Alg. 5.6). However, this does increase the time complexity to $O(\lg n) = O(\lg \lg N)$. This gives us a trade off between space and time. By allowing $O(\iota)$ steps for the retrieve method we need $M^{\lg N/2^\iota} \cdot m$ bits for the table.

```
retrieve(j)
  if (j == N-1)
    v = reg[j/2] AND mask(j);
  else
    v = reg[(j+1)/2] AND mask(j+1);
  ι = ⌈lg n⌉;
  do
    ι = ι-1;
    vnew = (v>>((2^ι)m)) + (v AND ((1<<((2^ι)m))-1));
    v = vnew;
  while (ι > 0)
  sum = v;
  if (j == N-1)
    sum = vn1 + sum;
  return sum;
```

Algorithm 5.6: Retrieve in a N-m-tree stored in m-Yggdrasil memory using no additional memory ($O(\lg \lg N)$ time).

Lemma 5.2 *The retrieve operation of the Prefix Sum problem can be supported in $O(\iota + 1)$ time using $O(M^{\lg N/2^\iota} \cdot m + m)$ bits of memory in addition to the N-m-tree. Part of the N-m-tree is stored in m-Yggdrasil memory.*

By adjusting ι we can achieve the following result:

Corollary 5.1 *The retrieve operation of the Prefix Sum problem can be supported in:*

- $O(1)$ *time using* $O(M^{(\lceil \lg N \rceil)/2} \cdot m)$ *bits of memory in addition to the N-m-tree, with* $\iota = 1$.
- $O(\lg \lg N)$ *time using* $O(m)$ *bits of memory in addition to the N-m-tree, with* $\iota = \lceil \lg \lg N \rceil$.

2.1 Addition modulo M

Let us consider the two m-bit operands a and b which are split into two pieces each (a_{lo}, a_{hi}, b_{lo} and b_{hi}). The two pieces a_{lo} and a_{hi} contain the $m/2$ least and most significant bits of a respectively (similarly for b_{lo} and b_{hi}). Note that a_{lo} and the other pieces are stored in m-bit but only the $m/2$ least significant bits are used.

We can now add the the two operands

$$c1_{lo} = a_{lo} + b_{lo} \quad (1)$$
$$c1_{hi} = a_{hi} + b_{hi} \ . \quad (2)$$

However, both $c1_{lo}$ and $c1_{hi}$ might need $m/2+1$ bits for its result. The $m/2+1$ bit of $c1_{lo}$ should be added to $c1_{hi}$ and we split $c1_{lo}$ into two pieces ($c1_{lo,lo}$ and $c1_{lo,hi}$) and add the most significant bits to $c1_{hi}$,

$$c_{hi} = c_{hi} + c_{lo,hi} \quad (3)$$
$$c_{lo} = c_{lo,lo} \ . \quad (4)$$

The result of $a+b$ is now stored in c_{lo} and c_{hi} and we have not used more than m bits in any word. However, in total $m+1$ might be needed for the value.

To compute $c \bmod M$ we can check whether or not $c - M >= 0$, if so $c \bmod M = c - M$ and otherwise $c \bmod M = c$. However, we do not want to produce a negative value since that would affect all the bits in the word. Instead we add an additional 2^m to the value and compare to 2^m, i.e. $c + 2^m - M \geq 2^m$. Since $2^m - M \geq 0$ this will never produce a negative value. Note that $c + 2^m - M < M - 1 + M - 1 + 2^m - M = M + 2^m - 2 <= 2^{m+1} - 2$ which only needs $m+1$ bits to be represented. Hence, if we calculate this value using the strategy above we will not use more than m bits of any word.

Furthermore, a straight forward less than comparison can not be performed using word-size parallelism since all bits of the words are considered. Instead we view the comparison as a check whether the $m+1$st bit is set or not. If it is set the value is larger than or equal to 2^m (cf. [19, 22]). We can actually create a bit mask which consists of m 1s if the $m+1$st bit is set and m 0s otherwise

$$d = (c + 2^m - M \text{ AND } 2^m) - ((c + 2^m - M \text{ AND } 2^m) >> m) \ . \quad (5)$$

This bit mask d can then be used to calculate $res = c \bmod M$. Since res is equal to $c - M$ if the $m+1$st bit of c is set and c otherwise we get

$$res = ((c - M) \text{ AND } d) \text{ OR } (c \text{ AND NOT } d) \ . \quad (6)$$

When computing $c - M$ we must make sure that we do not produce a negative value. This is done by using a similar strategy as for addition above, but we also set any of the bits in $c_{hi,hi}$ to 1 during the computation. If $c - M$ is greater than 0 this will not affect the result and otherwise the result will not be used.

We have a procedure which can be used to compute $(a + b) \bmod M$ without using more than m bits in any word. Hence, word-size parallelism can be used and we get our main result from this section:

Theorem 5.1 *Using the N-m-tree together with the m-Yggdrasil memory we can support the operations of the Prefix Sum problem in $O(\iota + 1)$ time using $(N - 1)m$ bits of m-Yggdrasil memory and $O(M^{n/2^\iota} \cdot m + m)$ bits of ordinary memory.*

3 An $O(\lg N)$ Solution to the General and Dynamic Prefix Sum Problem

We can actually partially solve the General Prefix Sum problem using the N-m-tree data structure and the m-Yggdrasil variant of RAMBO. All binary operations such that all elements in the universe have a unique inverse element (i.e. binary operations which form a *Group* with the set of elements in the universe) and only affect the m bits involved in the operation can be supported. This includes for example addition and subtraction but not the maximum function.

To solve the General and Dynamic Prefix Sum problem for semi-group operations we modify the Binary Segment Tree (BinSeT) data structure suggested by Brodnik and Nilsson. It was designed to handle in-advance resource reservation [21, pp 65-80] and if it is slightly modified it can solve both the General and Dynamic Prefix Sum problems efficiently. The original BinSeT stores, in each internal node, μ, the maximum value over the interval, and δ, the change of the value over the interval. Further, it also stores τ, the time of the left most event in the right subtree.

Instead of storing times as interval dividers we store array indices. To solve the Dynamic Prefix Sum problem with addition as operation and we only need to store δ. When solving the General and Prefix Sum problem one need to store information depending on the two binary operations $\oplus_u$ and $\oplus_r$.

When adding a new array position or deleting an array position the tree is rebalanced (cf. [1, 20]) and hence the height is always $O(\lg N)$. When updating a value in an array position we start at the root and search for the proper leaf using the interval dividers. During the back tracking of the recursion we update the information stored in each affected node.

At retrieval we process the information of the proper nodes when traversing the tree. Since the height of the tree is $O(\lg N)$ all the operations can be performed in $O(\lg N)$ time. This matches the lower bound by Hampapuram and Fredman [14]

BinSeT consists of $O(N)$ nodes when we use it to solve the General Prefix

Sum. Each node contains $O(1)$ m-bit values and hence the total space requirement is $O(Nm)$ bits.

4 Conclusion

The Dynamic and General Prefix Sum problems can both be solved optimally in $\Theta(\lg N)$ using $O(Nm)$ space under the comparison based model with semi-group operations.

The Prefix Sum problem can be solved in $O(1)$ time under the RAMBO model when we allow $O(\sqrt{M^{(\lceil \lg N \rceil)}} \cdot m)$ bits of ordinary memory and $O(Nm)$ bits of m-Yggdrasil memory to be used. This is a huge amount of ordinary memory and if we restrict the space requirement to be sub exponential in both N and M ($O(m)$ bits of ordinary memory and $O(Nm)$ bits of m-Yggdrasil memory) we need to used $O(\lg \lg N)$ time. We know of no better lower bound under RAMBO than the trivial $\Omega(1)$ when only allowing $O((N^{O(1)} + M^{O(1)})m)$ space.

Further, it is currently unknown if one can achieve a $O(1)$ solution to the Dynamic and General Prefix Sum problems using the RAMBO model. Another open question is whether or not it is possible achieve a $o(\lg N)$ solution to the multidimensional variant.

Acknowledgment

We thank the anonymous reviewers for helpful comments and additional references.

References

[1] G. M. Adelson-Velskii and E. M. Landis. An algorithm for the organization of information. In *Soviet Math. Doclady 3*, pages 1259–1263, 1962.

[2] Alok Aggarwal and Ashok K. Chandra. Virtual memory algorithms (preliminary version). In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing*, pages 173–185. ACM Press, May 2–4 1988.

[3] P. Beame and F. E. Fich. Optimal bounds for the predecessor problem and related problems. *Journal of Computer and System Sciences*, 65(1):38–72, 2002.

[4] Fredrik Bengtsson and Jingsen Chen. Space-efficient range-sum queries in OLAP. In Yahiko Kambayashi, Mukesh Mohania, and Wolfram Wöß, editors, *Data Warehousing and Knowledge Discovery: 6th International Conference DaWaK*, volume 3181 of *Lecture Notes in Computer Science*, pages 87–96. Springer, September 2004.

[5] Andrej Brodnik. *Searching in Constant Time and Minimum Space (*Minimæ Res Magni Momenti Sunt*)*. PhD thesis, University of Waterloo, Waterloo, Ontario, Canada, 1995. (Also published as technical report CS-95-41.).

[6] Andrej Brodnik, Svante Carlsson, Michael L. Fredman, Johan Karlsson, and J. Ian Munro. Worst case constant time priority queue. *Journal of System and Software*, 78(3):249–256, December 2005.

[7] Andrej Brodnik and John Iacono. Dynamic predecessor queries. Unpublished manuscript, 2006.

[8] Arash Farzan and J. Ian Munro. Succinct representation of finite abelian groups. In *Proceedings of the 2006 International Symposium on Symbolic and Algebraic Computation*, Lecture Notes in Computer Science. Springer, 2006. To appear.

[9] Michael L. Fredman. The complexity of maintaining an array and computing its partial sums. *Journal of the ACM*, 29(1):250–260, January 1982.

[10] Michael L. Fredman and Michael E. Saks. The cell probe complexity of dynamic data structures. In *Proceedings of the 21st Annual ACM Symposium on Theory of Computing*, pages 345–354. ACM Press, May 14–17 1989.

[11] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In IEEE, editor, *40th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 285–297. IEEE Computer Society, IEEE Computer Society, October 17–19 1999.

[12] Steven P. Geffner, Divyakant Agrawal, Amr El Abbadi, and T. Smith. Relatve prefix sums: An efficient approach for querying dynamic OLAP data cubes. In *Proceedings of the 15th International Conference on Data Engineering*, pages 328–335, 1999.

[13] Steven P. Geffner, Mirek Riedewald, Divyakant Agrawal, and Amr El Abbadi. Data cubes in dynamic environments. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, pages 31–40, 1999.

[14] Haripriyan Hampapuram and Michael L. Fredman. Optimal biweighted binary trees and the complexity of maintaining partial sums. *SIAM Journal on Computing*, 28(1):1–9, 1998.

[15] C. Ho, R. Agrawal, N. Megiddo, and R. Srikant. Range queries in OLAP data cubes. In *Proceedings ACM SIGMOD International Conference on Management of Data*, pages 73–88, 1997.

[16] Wing-Kai Hon, Kunihiko Sadakane, and Wing-Kin Sung. Succinct data structure for searchable partial sums. In Toshihide Ibaraki, Naoki Katoh, and Hirotaka Ono, editors, *Algorithms and Computation – ISAAC 2003, 14th International Symposium*, volume 2906 of *Lecture Notes in Computer Science*, pages 505–516. Springer, December 2003.

[17] Richard M. Karp and Vijaya Ramachandran. Parallel algorithms for shared-memeory machines. In van Leeuwen [28], chapter 17, pages 869–941.

[18] Roni Leben, Marijan Miletić, Marjan Špegel, Andrej Trost, Andrej Brodnik, and Johan Karlsson. Design of high performance memory module on PC100. In *Proceedings Electrotechnical and Computer Science Conference*, pages 75–78, Slovenia, 1999.

[19] Kjell Lemström, Gonzalo Navarro, and Yoan Pinzon. Practical algorithms for transposition-invariant string-matching. *Journal of Discrete Algorithms*, 3(2–4):267–292, 2005.

[20] Anany Levitin. *Introduction to The Design & Analysis of Algorithms*. Pearson Education Inc., Addison-Wesley, 2003.

[21] Andreas Nilsson. *Data Structures for Bandwidth Reservation and Quiality of Service on the Internet*. Lic. thesis, Department of Computer Science and Electrical Engineering, Luleå University of Technology, Luleå, Sweden, April 2004.

[22] W. Paul and J. Simon. Decision trees and random access machines. In *Proc. Int'l. Symp. on Logic and Algorithmic*, pages 331–340, Zurich, 1980.

[23] Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct dynamic data structure. In *Algorithms and Data Structures, 7th International Workshop*, volume 2125 of *Lecture Notes in Computer Science*, pages 426–437. Springer, 8–10 August 2001.

[24] Mirek Riedewald, Divyakant Agrawal, and Amr El Abbadi. Flexible data cubes for online aggregation. In *Database Theory - ICDT 2001, 8th International Conference, London , UK, January 4-6, 2001, Proceedings*, volume 1973 of *Lecture Notes in Computer Science*, pages 159–173, 2001.

[25] Mirek Riedewald, Divyakant Agrawal, Amr El Abbadi, and Renato Pajarola. Space-efficient data cubes for dynamic environments. In *Proceedings of the International Conference on Data Warehousing and Knowledge Discovery (DaWak)*, pages 24–33, 2000.

[26] L. G. Valiant. General purpose parallel architectures. In van Leeuwen [28], chapter 18, pages 943–971.

[27] Peter van Emde Boas. Machine models and simulations. In van Leeuwen [28], chapter 1, pages 3–66.

[28] Jan van Leeuwen, editor. *Handbook of Theoretical Computer Science*, volume A: Algorithms and Complexity. Elsevier/MIT Press, Amsterdam, 1990.

[29] Andrew C. Yao. On the complexity of maintaining partial sums. *SIAM Journal on Computing*, 14(2):277–288, May 1985.