Basic Research in Computer Science

BRICS RS-96-30

Fusion Trees can be Implemented with AC^0 Instructions only

Arne Andersson
Peter Bro Miltersen
Mikkel Thorup

BRICS Report Series **RS-96-30**

ISSN 0909-0878 **September 1996**

**Copyright © 1996, BRICS, Department of Computer Science
University of Aarhus. All rights reserved.**

**Reproduction of all or part of this work
is permitted for educational or research use
on condition that this copyright notice is
included in any copy.**

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

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

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

BRICS publications are in general accessible through WWW and anonymous FTP:

`http://www.brics.dk/`
`ftp ftp.brics.dk (cd pub/BRICS)`

Fusion trees can be implemented with AC^0 instructions only

Arne Andersson
Lund University
arne@dna.lth.se

Peter Bro Miltersen*
BRICS†, University of Aarhus
bromille@daimi.aau.dk

Mikkel Thorup
University of Copenhagen
mthorup@diku.dk

Abstract

Addressing a problem of Fredman and Willard, we implement fusion trees in deterministic linear space using AC^0 instructions only.

1 Introduction

Fredman and Willard [FW93], based on earlier ideas of Ajtai, Fredman, and Komlos [AFK84], introduced the *fusion tree*. A fusion tree is a data structure maintaining a subset S of $U = \{0, 1, \ldots, 2^{w-1}\}$ under insertions, deletions, predecessor and successor queries (i.e. for any element x of U, we can find the predecessor and successor of x in S), and rank queries (i.e. for any element x of U, we can find the number of elements of S smaller than or equal to x). The model of computation for the fusion tree is a random access machine whose registers contain w-bit words (i.e. members of U), and with an

*Supported by the ESPRIT Long Term Research Programme of the EU under project number 20244 (ALCOM-IT).

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

instruction set which includes unit-cost addition, subtraction, multiplication, comparison, and bit-wise Boolean `AND`. The fusion tree maintains a set of size n using $O(n)$ space and amortized time $O(\log n / \log\log n)$ per operation. An immediate corollary to the existence of the fusion tree is that n w-bit keys can be sorted in time $O(n \log n / \log\log n)$ and space $O(n)$ on a RAM with word size w.

Fredman and Willard points out that multiplication is not an AC^0 instruction, that is, there is no circuits for multiplication of constant depth and of size (number of gates) polynomial in the word length. Here, the gates are negations, and $\wedge$- and $\vee$-gates with unbounded fan-in. They pose as an open question if the fusion tree can be implemented using AC^0 instructions only. The motivation for this question is obvious: AC^0 is the class of functions which can be computed in constant time in a certain model of hardware, and it is therefore arguably questionable to assume unit-cost for executing operations outside of this class.

In this paper, we solve this problem by showing that, given a small set of non-standard AC^0 instructions in addition to the more standard ones (addition, comparison, bitwise Boolean operations, and shifts), the fusion tree can be implemented, with the same asymptotic space and time bounds as in [FW93].

Our presentation can also be seen as an alternative explanation of the basic mechanisms in fusion trees. We believe that our use of special-purpose instructions in place of the ingenious use of multiplication in [FW93] may make our presentation easier to understand for the casual reader.

2 Model of computation and notation

We use a RAM with word size w and we consider n w-bit keys that can be treated as binary strings or (unsigned) integers. We assume for convenience that the keys are all different, this implies that $w \geq \log n$. We shall also assume that $\sqrt{w}$ is a power of two.

A w-bit word will sometimes be viewed as a concatenation of $\sqrt{w}$ *fields*. Each field is of length $\sqrt{w} - 1$; to the left of each field is the *test bit* of the field. By a *bit pointer* we mean a $(\log w)$-bit key, such a key can be used to specify a bit-position within a word. W.l.o.g we assume that $\log w < \sqrt{w} - 1$ and hence a bit pointer fit in a field. As an example, if $w = 64$ a word

contains 8 fields of length 7, one test bit is stored with each field.

We will use upper-case characters to denote words and lower-case characters to denote fields. For any bit-string α, we use $|\alpha|$ to denote its length, and for $i = 1, \ldots, |\alpha|$, $\alpha[i]$ is the ith bit in α. In particular, $\alpha[1]$ is the leftmost, and $\alpha[|\alpha|]$ is the rightmost bit of α. Also, for $1 \leq i \leq j \leq |\alpha|$, $\alpha[i..j] = \alpha[i] \cdots \alpha[j]$. Finally, $\texttt{int}(\alpha)$ is number represented by α, i.e. $\texttt{int}(\alpha) = \sum_{i=1}^{|\alpha|} 2^i \alpha[|\alpha| - i]$. Note that our indexing of words is slightly non-standard, it is more common to index words from right to left, starting with 0. However, in the main technical part of this paper it is most convenient to think of words as strings, and these are usually indexed from left to right starting with index 1.

Apart from the standard AC^0 instructions (comparison, addition, bitwise Boolean operations and shift), we use the following ones:

$\texttt{LeftmostOne}(X)$: returns a bit pointer to the leftmost 1 in X. A simple depth 2 circuit of quadratic size is indicated by:

$$\forall i \leq w : \texttt{int}(\texttt{LeftmostOne}(X)) = i \iff (\bigwedge_{j=1}^{i-1} \neg x[j]) \wedge x[i].$$

In fact, $\texttt{LeftmostOne}(X)$ can be implemented with standard instructions. Converting X to floating point representation is a standard operation, and afterwards, we just need to return the exponent.

$\texttt{Duplicate}(x, d)$: Returns a word containing copies of the field x in the d rightmost fields.

$\texttt{Select}(X, K)$: The first $\sqrt{w} - 1$ fields in K are viewed as bit pointers; a field is returned, containing the selected bits in X. Not all fields need to be used. The test bit of a used field is 1. A depth 3 circuit of size $O(w^{3/2})$ is indicated by:

$$\forall i \leq \sqrt{w} : \texttt{Select}(X, K)[i] = K[ib] \wedge \bigvee_{j=1}^{\sqrt{w}} (j = \texttt{int}(K[ib{+}1..(i{+}1)b{-}1]) \wedge X[j])$$

Furthermore, we assume that the constants $\texttt{b} = \sqrt{w} - 1$ and $\texttt{k} = \log \sqrt{w}$ are known, $\texttt{b}$ is the length of a field.

3 The AC^0 fusion tree

Lemma 1 *Let Y be a word where the d rightmost fields contain one* `b`*-bit key each. Furthermore, assume that the d keys are sorted right-to-left, the d rightmost test bits are 0, and that all bits to the left of the d used fields (and their test bits) are 1. Then, given a* `b`*-bit key x, we can compute the rank of x among the keys in Y in constant time.*

Proof: The crucial observation is due to Paul and Simon [PauSim80]; they observed that one subtraction can be used to perform comparisons in parallel.

Let M be a key where the d rightmost test bits are 1 and all other bits are 0. In order to compute the rank of x among the keys in Y, we place d copies of x in the d rightmost fields of a word X. We let the test bits of those fields be 1. By the assignment $R \leftarrow (X - Y)$ `AND` M the ith test bit from the right in R will be 1 if and only if $x \geq y_i$. All other test bits (as well as all other bits) in R will be 0. Hence, from the position of the leftmost 1 in R we can compute the rank of x.

We implement this in the function `PackedRank` below. First, we compute d, the number of keys contained in Y. This is the same number as the number of set test bits in `NOT` Y, which can be determined using the function `LeftField` below. Next, we create the word X and the mask M. Finally, we make the subtraction and extract the rank.

For clarity, we introduce two simple AC^0 functions.

`FillTestBits`(d): returns a word where the d rightmost test bits are set and all other bits are zeroes. Can be implemented with shift, bitwise logical operations, and `Duplicate`.

`LeftField`(Y): If the leftmost 1 in Y is a test bit, and if all test bits to the right of this test bit are set, the number of set test bits is returned. This can be computed as $\texttt{b} + 1 - (\texttt{LeftmostOne}(Y) - 1)/(\texttt{b} + 1)$. We don't have a division operation, but since $\texttt{b}+1 = 2^{\texttt{k}}$, we can implement the division by a right shift by `k`.

Algorithm A: `PackedRank` (Y, x)

A.1. $d \leftarrow$ `LeftField(NOT` Y`)`.

A.2. $M \leftarrow$ `FillTestBits`(d).

A.3. $X \leftarrow$ `Duplicate`(x, d) `OR` M.

A.4. $R \leftarrow (X - Y)$ `AND` M.

A.5. `return LeftField`(R).

`PackedRank` clearly runs in constant time. ■

Lemma 2 *Given a word Y and a key x as in Lemma 1, we can generate a new word where x is properly inserted among the keys in Y in constant time.*

Proof: We need a function `InsertField`(Y, x, i): Insert x as the $(i+1)$'st field from the right in Y and set x's test bit to 0. This can easily be implemented with shift and bitwise Boolean operations.

The function `InsertKey` below implements the lemma.

Algorithm B: `InsertKey` (Y, x)

B.1. `return InsertField`$(Y, x,$ `PackedRank`$(Y, x))$.

■

Lemma 3 *Given d sorted w-bit keys, $d \leq \sqrt{w}$, a static data structure can be constructed in $O(d)$ time and space, such that it supports neighbour queries in $O(1)$ worst-case time.*

Proof: The main idea is to make use of significant bit positions. View the set of w-bit sorted keys $Y_1, \ldots, Y_d$ as stored in a binary trie. Each key is represented as a path down the trie, a left edge denotes a 0 and a right edge denotes a 1. We get the significant bit positions by selecting the levels in the trie where there is at least one binary (that is, non-unary) node. These bit positions can be computed by taking the position of the first differing bit between all pairs of neighbouring keys in $Y_1, \ldots, Y_d$. By extracting the significant bits from each key we create a set of compressed keys $y_1, \ldots, y_d$. Since the trie has exactly d leaves, it contains exactly $d-1$ binary nodes. Therefore, the number of significant bit positions, and the length of a compressed key, is at most $d-1$. Since $d - 1 \leq \sqrt{w} - 1 =$ `b`, we can pack these compressed keys in linear time by repeated calls to `InsertKey` in Lemma 2.

We need the following AC^0 functions, which can be implemented in a straightforward way:

`DiffPtr`(X, Y): returns a bit pointer to the leftmost differing bit between X and Y. (Can be implemented as `LeftmostOne`(X `XOR` Y).)

`Fill`(X, p): X is a word and p is a bit pointer. Returns a copy of X where all bits to the right of position p have the same value as the bit in position p. If $p = w + 1$, `Fill`$(X, p) = X$. (Can be implemented by shifting, addition, and bitwise Boolean operations.)

The procedure `Construct` takes as input a set of keys $Y_1, \ldots, Y_d$ and computes the set of significant bit positions; pointers to these positions are concatenated in sorted order in the word K. Next, a set of compressed keys are created by selecting the bits specified in K from $Y_1, \ldots, Y_d$. These compressed keys are packed in the word Y.

Algorithm C: `Construct` $(Y_1, \ldots, Y_d)$

C.1. $K \leftarrow 0$.

C.2. For $i \leftarrow 1$ to $d-1$, $K \leftarrow$ `InsertKey`$(K,$ `DiffPtr`$(Y_i, Y_{i+1}))$.

C.3. $Y \leftarrow 0$.

C.4. For $i \leftarrow 1$ to d, $Y \leftarrow$ `InsertKey`$(Y,$ `Select`$(Y_i, K))$.

When implemented as above, the same bit-pointer may be packed several times in K. This makes no difference.

We can now compute the rank of a query key X in constant time. Let $x =$ `Select`(X, K) and let p_X denote the longest common prefix of X and any key in $Y_1, \ldots, Y_d$. Let y_i denote the compressed version of Y_i. Let $q_1 < q_2 < \cdots < q_{d-1}$ be the significant bit positions and let $q_d = w + 1$. Then $x[j] = X[q_j]$ and $y_i[j] = Y_i[j]$. Note that if $y_{i_1}[1..j] = y_{i_2}[1..j]$, then $Y_{i_1}[1..q_{j+1} - 1] = Y_{i_2}[1..q_{j+1} - 1]$.

Lemma 4 *If x has rank i in $y_1, \ldots, y_d$, then either Y_i or Y_{i+1} start by p_X.*

Proof: Let i' be such that $Y_{i'}$ has the prefix p_X. Let j be the maximal index such that $q_j \leq |p_X|$. Then $y_{i'}[1..j] = x[1..j]$. However, since i is the rank of x, we either have $x = y_i$ or $y_i < x < y_{i+1}$. This means that for i'' equal to either i or $i+1$, we have $y_{i''}[1..j] = y_{i'}[1..j]$. Hence $Y_{i''}[1..q_{j+1} - 1] = Y_{i'}[1..q_{j+1} - 1]$, but $q_{j+1} - 1 \geq |p_X|$, so $Y_{i''}$ must share the prefix p_X with $Y_{i'}$. ■

Lemma 5 *Consider a key Z that starts by p_X and where all remaining bits in Z have the same value as X's first distinguishing bit. Set $z = \texttt{Select}(Z, K)$. Let z^0 be the result of setting the last bit of z to 0, and let z^1 be the result of setting it to 1. Then $z \in \{z^0, z^1\}$, and then the rank of of X among $Y_1, \ldots, Y_d$ is either that of z^0 or that of z^1 among $y_1, \ldots, y_d$.*

Proof: Clearly Z has the same rank as X among $Y_1, \ldots, Y_d$. Let i be the rank of z among $y_1, \ldots, y_d$. If $y_i \neq z$, then i is the correct rank of X. The same holds if $Y_i = Z = X$. However, suppose that $y_i = z$ and $Y_i \neq Z$. Suppose that the first distinguishing bit $X[|p_X| + 1]$ of X is 0. Then there is some $j > |p_X|$, $j \notin \{q_1, \ldots, q_{d-1}\}$, such that $Y_i[j] = 1$. Hence $Y_i > Z$. Since $y_{i'} = y_i$ implies $Y_{i'} = Y_i$, we conclude that the rank of Z among $Y_1, \ldots, Y_d$ is the rank of z^1 among $y_1, \ldots, y_d$.

If $X[|p_X| + 1] = 1$, symmetrically we have that the rank of Z among $Y_1, \ldots, Y_d$ is the rank of z^0 among $y_1, \ldots, y_d$. ■

We can use the first lemma to compute the length of p_X and hence the position of X's first distinguishing bit. Once this position is known, we can apply the second lemma to find the proper rank of X.

We encode this in the function `Rank` below. The variable p is used to store the position of X's first distinguishing bit.

Algorithm D: $\texttt{Rank}(X)$

D.1. $i \leftarrow \texttt{PackedRank}(Y, \texttt{Select}(X, K))$.

D.2. If $i = 0$, $p \leftarrow \texttt{DiffPtr}(X, Y_1)$;

D.3. else $p \leftarrow \max(\texttt{DiffPtr}(X, Y_i), \texttt{DiffPtr}(X, Y_{i+1}))$

D.4. $Z \leftarrow \texttt{Fill}(X, p)$.

D.5. $z \leftarrow \texttt{Select}(Z, K)$.

D.6. $i \leftarrow \texttt{PackedRank}(Y, z)$.

D.7. If $Y_i \leq Z < Y_{i+1}$, return i;

D.8. else $z[\texttt{b}] \leftarrow \neg z[\texttt{b}]$; return $\texttt{PackedRank}(Y, z)$.

■

The original method by Fredman and Willard is slightly different. Instead of filling the query keys with 1s (or 0s) and making a second packed searching, they use a lookup table of size $\Theta(d^2)$ in a node of degree d.

The proof of our main theorem is similar to that of Fredman and Willard. Note that we can allow the B-tree nodes to have higher degree than in the original fusion tree: $\sqrt{w}$ compared to $w^{1/6}$.

Theorem 6 *A set can be maintained using linear space under insertion, deletion, predecessor, successor, and rank queries with* $O(\log n / \log\log n)$ *amortized time per operation on an* AC^0 *RAM.*

Proof: The proof proceeds as in [FW93]:

Lemma 3 allows us to implement a B-tree [BM72] node of degree $d \leq \sqrt{w}$. Searching in such a node takes constant time while splitting, merging, and adding/removing keys take $O(d)$ time. By keeping traditional, comparison-based, weight-balanced trees of size $\Theta(d)$ at the bottom of the B-tree, we can ensure that at most every $\Theta(d)$'th update causes any change in a B-tree node.

The number of B-tree levels is $O(\log n / \log d)$ and the height of a weight-balanced tree is $O(\log d)$. Since $w > \log n / \log\log n$, we can choose $d = \Theta(\sqrt{\log n})$ and the theorem follows. ■

References

[AFK84] M. Ajtai, M.L. Fredman, and J. Komlos, Hash functions for priority queues, *Information and Computation* 63:217–225, 1984.

[BM72] R. Bayer and E.M. McCreight, Organization and maintenance of large ordered indexes, *Acta Informatica* 1(3):173–189, 1972.

[FW93] M.L. Fredman and D.E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47:424–436, 1993.

[PauSim80] W. J. Paul and J. Simon. Decision trees and random access machines. In *Proc. International Symp. on Logic and Algorithmic*, Zürich, pages 331–340, 1980.

Recent Publications in the BRICS Report Series

RS-96-30 **Arne Andersson, Peter Bro Miltersen, and Mikkel Thorup.** ***Fusion Trees can be Implemented with AC^0 Instructions only*****. September 1996. 8 pp.**

RS-96-29 **Lars Arge.** ***The I/O-Complexity of Ordered Binary-Decision Diagram Manipulation*****. August 1996. 35 pp. An extended abstract version appears in Staples, Eades, Kato, and Moffat, editors,** ***Algorithms and Computation: 6th International Symposium*****, ISAAC '95 Proceedings, LNCS 1004, 1995, pages 82–91.**

RS-96-28 **Lars Arge.** ***The Buffer Tree: A New Technique for Optimal I/O Algorithms*****. August 1996. 34 pp. This report is a revised and extended version of the BRICS Report RS-94-16. An extended abstract appears in Akl, Dehne, Sack, and Santoro, editors,** ***Algorithms and Data Structures: 4th Workshop*****, WADS '95 Proceedings, LNCS 955, 1995, pages 334–345.**

RS-96-27 **Devdatt Dubhashi, Volker Priebe, and Desh Ranjan.** ***Negative Dependence Through the FKG Inequality*****. July 1996. 15 pp.**

RS-96-26 **Nils Klarlund and Theis Rauhe.** ***BDD Algortihms and Cache Misses*****. July 1996. 15 pp.**

RS-96-25 **Devdatt Dubhashi and Desh Ranjan.** ***Balls and Bins: A Study in Negative Dependence*****. July 1996. 27 pp.**

RS-96-24 **Henrik Ejersbo Jensen, Kim G. Larsen, and Arne Skou.** ***Modelling and Analysis of a Collision Avoidance Protocol using SPIN and UPPAAL*****. July 1996. 20 pp. Presented at** ***DIMACS Workshop SPIN96 – 2nd International SPIN Verification Workshop on Algorithms, Applications, Tool Use, Theory*** **(Rutgers University, New Jersey, USA, August 5, 1996).**

RS-96-23 **Luca Aceto, Wan J. Fokkink, and Anna Ingólfsdóttir.** ***A Menagerie of Non-Finitely Based Process Semantics over BPA^*: From Ready Simulation Semantics to Completed Tracs*****. July 1996. 38 pp.**