Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

Corso di Laurea Magistrale in Ingegneria Informatica

Tesi di Laurea

Parallel Stack Distance Computation for the Least Recently Used Replacement Policy

Advisor: Prof. Gianfranco Bilardi

Student: Alberto Chimetto

9 Ottobre 2017
Anno Accademico 2016/2017

In God we trust.
All others must bring data.

W. EDWARDS DEMING

Abstract

Storage hierarchies have been a staple in the organization of the memory in a computer system. Faster levels of memory are placed closer to the CPU, while further devices have higher storage capacity, but take considerably more time to access. In this work we focus on caches, and specifically on the stack distance computation problem. This problem involves the calculation of the minimum cache capacity containing a given page reference encountered while processing the address trace of a computer program.

Stack distance computation has been widely investigated for a single processor; our objective is to design a parallel algorithm in order to speedup this operation. To this end, we present a theoretical framework which involves Finite State Machines and prefix computation on semigroups. With this framework, we have focused on the LRU policy and developed a parallel algorithm capable of analyzing a trace of L references to V distinct addresses in time $O((L \log V)/P)$, when executed by $P \leq L/V$ processors. This represents a linear speedup with respect to the best known sequential algorithm. We have implemented our algorithm in C++ and evaluated its performances on an IBM Power 770 machine.

Contents

Chapter 1

Introduction

As of today, it has been recognized that storage systems cannot achieve the required performances of speed and capacity within a single technology at an acceptable cost-performance balance. For this reason, the memory in a state-of-the-art computer system is organized in several levels. This has led to the development of *storage hierarchies*, where faster pieces of hardware are closer to the CPU than slower ones. The goal is to maximize the number of times items of interest are in the fast memory levels: most references to those items will then be directed to the fast devices, while the majority of data is still stored in the slow, large stores. The system will then acquire almost the same speed of the fast levels, although maintaining the cost-per-bit of the less expensive ones. This trade-off between cost and performances is the main justification for storage hierarchies. In Figure 1.1 is depicted a typical storage hierarchy:

- **CPU registers:** $\sim 0.3ns$ access time;
- **L1 cache:** $\sim 1ns$ access time;
- **L2 cache:** $\sim 4ns$ access time;
- **L3 cache:** $\sim 30ns$ access time;
- **Main memory/RAM:** $\sim 100ns$ access time;
- **Secondary memory/HDD:** $\sim 10ms$ access time.

At the top of the pyramid are located the fast and expensive devices, while at the bottom we encounter levels of memory with higher access times, but capable of storing great quantities of data at a reasonable price. Modern architectures usually contain three levels of cache memory, while Solid State

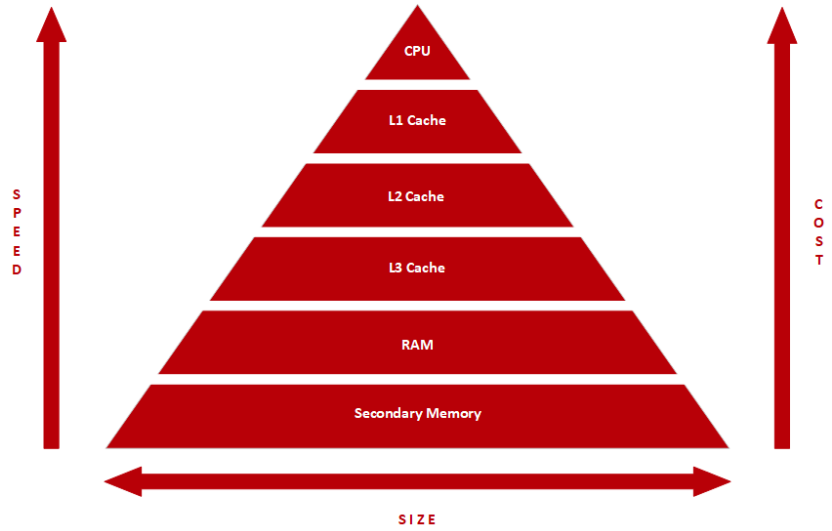


Figure 1.1: Schematic of a typical storage hierarchy

Drives (SSD) can replace traditional Hard Disk Drives (HDD), granting lower access times ($\sim 100\mu s$).

Following the path taken in [1], in this work we will focus on two levels of memory called the *buffer* and the *backing storage*. The task of the CPU is to process the instructions composing a computer program; when one of these instructions contains an access request for a stored item, this request is called a *hit* if said item is contained in the buffer, and a *miss* if it is in the backing storage. Upon a miss, the item will be searched in the backing storage and brought into the buffer. If the latter is full, a *replacement policy* will choose one item in the buffer to discard to make place for the requested one. A *replacement algorithm* is a procedure which implements one of the policies. These replacement operations are transparent (except for timing) to the remainder of the computer, so the work of the storage hierarchy can be considered indistinguishable from that of a single-device system.

As a final remark, it is important to note that the hierarchy model will probably remain the standard paradigm in future computational systems. As mentioned in [2], even if we were able to reach the minimum chip-size of silicon components, so that we could have CPU registers in large quantities at a reasonable price, the principle of maximum information speed states that a bit of information can not travel faster that the speed of light. This means that the physical distance of memory blocks from the CPU core will inherently introduce a hierarchy, since the access times will linearly grow as the distances increase.

1.1 Replacement policies for the memory hierarchy

As described in Chapter 1, the goal of a replacement policy is usually to minimize the number of misses encountered while processing the sequence of access requests. With the term *address trace* we indicate the sequence of addresses where each item is represented by its address in a virtual space. In today's systems, the address space is partitioned into equal sized *pages*, which are the block of information being moved between the various levels of the hierarchy. We will then use the terms 'page' and 'item' interchangeably. Below is a list of some of the most used replacement policies:

- **Least Recently Used (LRU):** the item (page) to be discarded when the buffer is full is the least recently used one;
- **Least Frequently Used (LFU):** this policy counts how often each page is referenced, and the least used ones will be discarded first. It is very similar to LRU, but it stores the number of accesses for every page instead of how recently it has been accessed;
- **First In First Out (FIFO):** the buffer behaves as a FIFO queue and the page to be discarded is the first accessed, without any regard to how many times or how often it has been accessed before;
- **Last In First Out (LIFO):** the buffer behaves in the exact opposite way than a FIFO queue;
- **Random Replacement (RR):** the page to be discarded is selected randomly between those in the buffer. This policy is extremely simple, since it does not need to keep track of any information about the history of the buffer;
- **Belady's algorithm (MIN):** the purpose of this optimal replacement policy is to minimize the number of misses. It is an off-line algorithm, since it needs a certain amount of lookahead to determine the page to be replaced in the buffer;
- **Optimal algorithm (OPT):** similar to MIN, this policy replaces the page whose reference is farthest in the future. This optimal algorithm as well cannot be realized on-line in modern computers, since it requires the knowledge of the entire address trace to determine future page references.

As underlined in [3], it is important to note that both OPT and MIN are optimal policies, since they achieve the minimum number of misses in any address trace. There are however several differences between them: for instance, MIN and OPT incur in a miss at exactly the same accesses, but they may choose to replace different pages. Moreover, the look-ahead needed to determine the item to evict may be smaller for MIN than for OPT.

1.2 The LRU policy

In this section we will briefly review the main aspects of the Least Recently Used replacement policy. As mentioned before, under LRU policy the page to be replaced is the one that has not been referenced for the longest time. In Figure 1.2 is shown an example of LRU replacement; the buffers have capacity up to four elements and the asterisks below them indicate a hit, namely the access requests reference objects already in the buffer.

The figure also highlights one of the key features of the LRU replacement, that is the *inclusion property*: the set of the C most recently used pages is always contained in the set of the $C+1$ most recently used items. This means that the success function $F(C)$, the number of hits over the entire address trace, will always be greater for buffers with greater capacity.

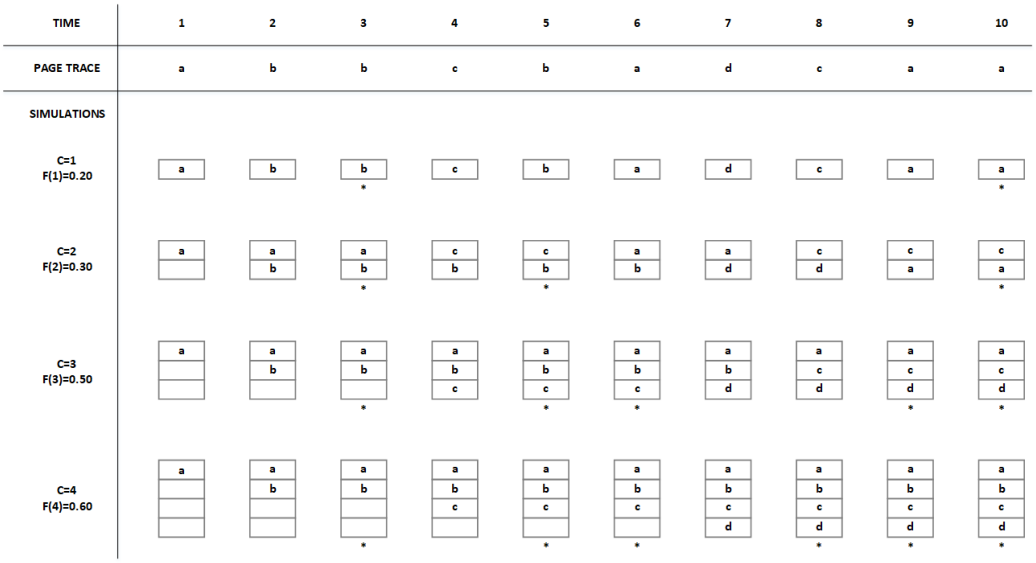


Figure 1.2: LRU replacement for buffers of different capacities.

We introduce the concept of *stack distance*, namely the distance of the next page to be referenced from the top of the stack. If the next item to be

processed is not in the buffer (i.e. it has been discarded or it has yet to be seen for the first time), the distance is set to $+\infty$.

Figure 1.3 illustrates how the LRU policy works in a buffer of four elements capacity. The stack distance is $+\infty$ when the element is first inserted in the buffer, while later it depends on its position in the stack. The distance counters in the last row keep track of how many times each stack distance is observed.

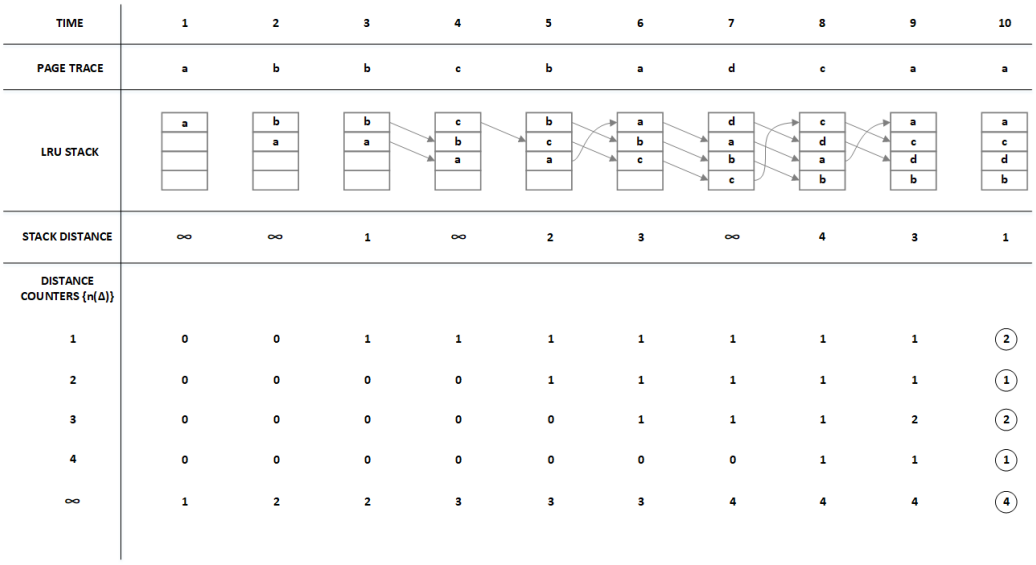


Figure 1.3: LRU stack for a buffer with four pages.

The stack distance strategy allows us to directly evaluate the success function: by summing all the times a stack distance has been observed over all the address trace, we are able to estimate the minimum buffer capacity needed to reduce the number of misses encountered while processing the sequence of access requests. Mathematically, let $n(\Delta)$ be the number of times the stack distance Δ is observed, C the buffer capacity and L the length of the address trace. The number of times that the next page is in the buffer is

$$N(C) = \sum_{\Delta=1}^{C} n(\Delta) \tag{1.1}$$

and the success function is hence given by the expression

$$F(C) = N(C)/L \tag{1.2}$$

The set $\{n(\Delta)\}$ can be determined from a set of distance counters, as mentioned before. These counters are initialized to zero, and the counter

for a given distance Δ is incremented when that distance occurs. For page numbers of k-bit we need at most $2^k + 1$ counters, since $1 \leq \Delta \leq 2^k$ and 1 counter is reserved for the infinity distance $\Delta = +\infty$. At the end of the trace, the set $\{n(\Delta)\}$ is determined by the final values of each distance counter, while the success function $F(C)$ can be obtained from Equations 1.1 and 1.2.

In the remainder of this section we will show a numerical example of the evaluation of the success function. In Figure 1.3 is shown the content of the buffer in each moment of the address trace processing. The final values for Δ's of 1,2,3,4 and ∞ are, respectively, 2,1,2,1 and 4. This distribution is shown in Figure 1.4A while the success function $F(C)$ is computed using Equations 1.1 and 1.2 and is shown in Figure 1.4B. The results are the same as those obtained in the simulation in Figure 1.2.

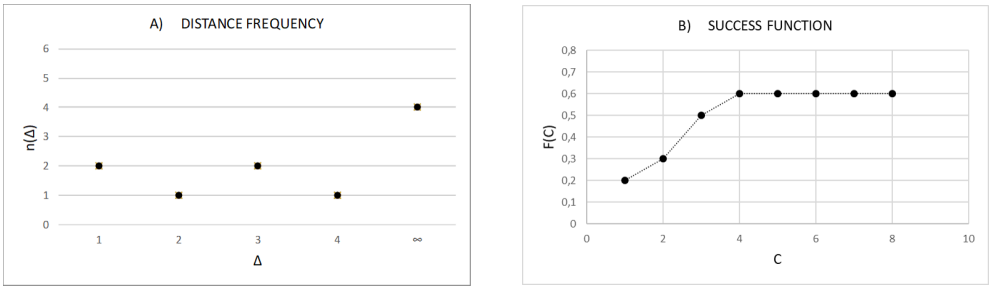


Figure 1.4: Distance frequencies and success function.

We can note that under LRU replacement, as intuition may have suggested, $F(C)$ is always a monotonic, non-decreasing function of C. Moreover, let V be the number of distinct pages in the whole address trace: $F(C)$ never exceeds $(L - V)/L$ for any capacity, since all the pages are initially located in the backing storage and need to be brought in the buffer at least once.

1.3 The stack distance framework

We will now formally review some of the concepts outlined before as well as the notation used in the remainder of the thesis.

Addresses, buffers, traces. Let A be a totally ordered set of *virtual addresses* and $\{I_x \in A\}$ the corresponding space of *virtual items*, where I_x is the unique item identified by address x. A *buffer* of *capacity* C is a container for up to C items. An *address trace* is a sequence $\boldsymbol{x} = (x_1, x_2, ..., x_L) \in A^L$ of virtual addresses, where L indicates the length of the trace. With $B_t(C)$ we indicate the set of addresses of the items in the buffer after having processed

$x_1, x_2, ..., x_{t-1}$. We call this set the *buffer state*.

Policies, hits, misses. A replacement policy is a function with three inputs: (i) an address trace $\boldsymbol{x}$, (ii) a buffer capacity C and (iii) an initial buffer state $B_1(C)$. Its output is a sequence of buffer states $B_2(C), ..., B_{L+1}(C)$. In our notation $x_t \in B_{t+1}(C)$, that is the item x_t at time t is brought in the buffer at time $t+1$. If at time $t = 1, 2, ..., L$ $x_t \in B_t(C)$, a *hit* occurs, while we will have a *miss* otherwise. An algorithm which implements a given replacement policy is called a replacement algorithm. A *demand policy* is a policy with the following proprieties:

a) in case of a hit the buffer does not change;

b) upon a miss (b1) if the buffer is full ($|B_t(C)| = C$) only one item is discarded and (b2) the requested item x_t is brought into the buffer.

Inclusion, stack, stack distance. Let V_t be the number of distinct pages processed up to time t and $\boldsymbol{x}$ the address trace being processed. A buffer satisfies the *inclusion property* if at any time t ($1 \leq t \leq L$), for some V_t:

a) $|B_t(C)| = min(C, V_t)$ for any $C \geq 1$;

b) $B_t(C-1) \subseteq B_t(C)$ for any $C \geq 2$.

An *inclusion policy* P is defined as a demand policy where the buffers satisfy the inclusion property $\forall t > 1$, when they satisfy it for $t = 1$. For $C \leq V_t$, the *distinguished item* $\Sigma_t(C)$ for C is defined as $B_t(C) - B_t(C-1)$. The stack for an inclusion policy can then be defined as $\Sigma_t = (\Sigma_t(1), ..., \Sigma_t(V_t))$, that is the sequence of distinguished items brought in the buffer. This means that the stack is a permutation of the elements of $B_t(V_t)$. Note that if x_t is a hit for the buffer, then $V_{t+1} = V_t$, else $V_{t+1} = V_t + 1$. The stack distance Δ_t is defined as h if, for some $h \leq V_t$, we have that $\Sigma_t(h) = x_t$, and as $+\infty$ otherwise. Hence the stack distance is the depth of a page from the top of the stack, whether it has already been brought inside the buffer. The *critical capacity* is labeled $C_t^* = min(\Delta_t, V_t + 1)$, namely the smallest capacity that does not discard an item at time t.

Time of previous access, LRU policy, LRU stack. We denote the *time of previous access* to an address $z \in A$ as $r_t(z)$ and define it as:

a) if $z \in \{x_1, ..., x_{t-1}\}$, then $r_t(z) = max\tau : 1 \leq \tau \leq t-1 : x_\tau = z$;

b) if $z \notin \{x_1, ..., x_{t-1}\}$, but $\Sigma_1(i) = z$, then $r_t(z) = -i$;

c) $r_t(z) = -\infty$ in all other cases.

When the buffer is full, the Least Recently Used policy discards the item z with the smallest time of previous access $r_t(z)$; let us remember once more that LRU is an inclusion policy. We denote the LRU stack Σ_t^{LRU} as $\Lambda_t = (\Lambda_t(1), ...\Lambda_t(V_t))$ and its stack distance as $\Delta_t = \Delta_t^{LRU}$. In this stack, items are ordered according to their time of previous access, decreasing from top $(\Lambda_t(1))$ to bottom $(\Lambda_t(V_t))$. When element x_t is to be accessed, Σ_t^{LRU} is updated by inserting x_t in position $min(\Delta_t, V_t + 1)$, if it is not already the first element, and then by cyclically shifting the top $min(\Delta_t, V_t+1)$ positions of the stack by one position to the bottom; by doing so, x_t is moved to the top position $\Lambda_{t+1}(1)$ of the stack (Figure 1.5).

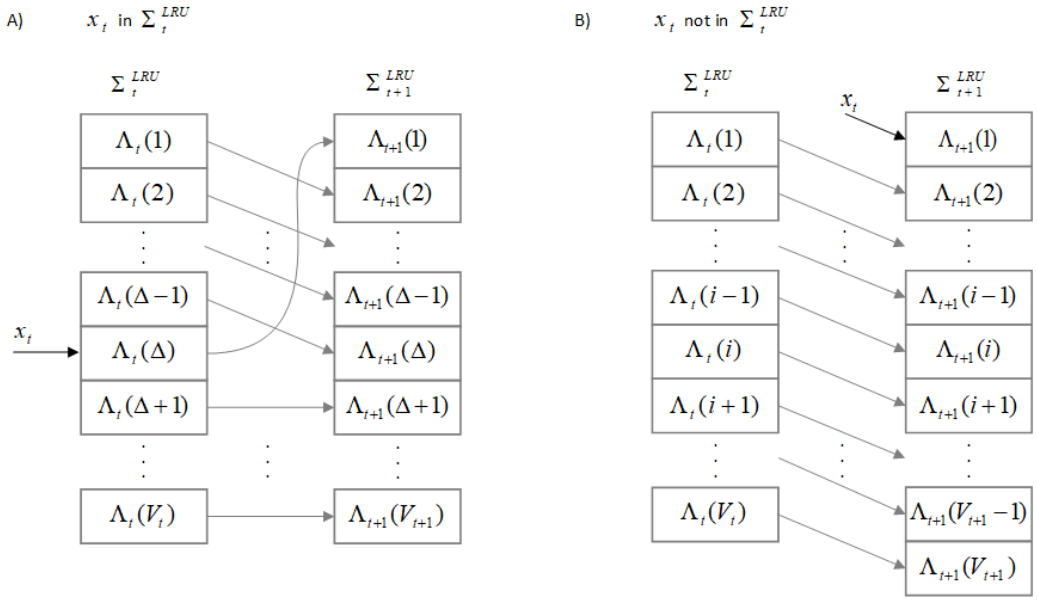


Figure 1.5: Update of an LRU stack.

1.4 Sequential algorithms for LRU stack distance computation

In this section we will examine two different implementations of the LRU *stack algorithm*. We define a stack algorithm as a replacement algorithm whose buffer contents in a demand-paged, two-level hierarchy satisfy the inclusion property for every address trace and every point in time. These algorithms are of particular interest to us, since it has been shown that

their stacks can be iteratively maintained, and that we can use their stack distance frequencies to obtain the corresponding success function $F(C)$ for a given address trace, as already shown in Equations 1.1 and 1.2.

Assuming that the algorithm is operating on an address trace $\boldsymbol{x}$ of length L with V distinct pages (of course $V < L$) and that the letter t is used to index the trace, a three step description of the algorithm is as follows:

while $(0 \leq t < L)$:

- **search:** in the stack Σ_t^{LRU} find the position of the most recent reference to page x_t;
- **count:** compute the stack distance Δ_t^{LRU} for the current location, by finding the last reference to the current location and counting its depth from the top of the stack (i.e. the number of elements above it). If there is no such reference, the stack distance is defined as $+\infty$;
- **update:** bring the most recent reference to the top of the stack.

1.4.1 $O(LV)$ algorithms

A naive implementation of the LRU stack algorithm computes the stack distances in time $T = O(LV)$ and auxiliary space $S_{aux} = O(V)$, needed to create the buffer Σ^{LRU}. The LRU stack can be implemented as a doubly linked list. For each page x_t in the address trace, the first two operations (**search** and **count**) are executed at the same moment, traversing the stack from top to bottom. If the element is found in the stack, its stack distance is recorded in a given distance counter. The **update** procedure, already shown in Figure 1.5, finally moves the element to the top of the stack. As already discussed, if no reference is found in the stack the element is still pushed to the top, but the recorded distance is $+\infty$.

Under these hypothesis it is easy to see that the algorithm runs in time $T = O(LV)$. Since many programs exhibit good locality (i.e. the program works on small amount of data at the same time), the worst case does not happen very often and many references are found close to the top of the stack. However, the few cases where the reference is found near the bottom of the stack cause severe slow-downs, and the overall performance is hence greatly affected.

1.4.2 $O(L \log V)$ algorithms

This section is heavily based on [4]. The bottleneck of the naive implementation is the linear traversal at each step of the linked list that makes up the stack. To implement a more efficient version of the LRU stack algorithm, we hence need a different data structure. In the following definitions we will use Bennet and Kruskal's [5] notations:

- **hashtable P:** the hashtable contains, for each reference x_i in the address trace $\boldsymbol{x}$, the time/index of its last use. We define $\mathcal{J}$ as the set of indexes of references to z that happened *before* a certain index t in the address trace:
$$\mathcal{J}_t = \{i | 0 \leq i < t \wedge x_i = z\}$$
We hence define the hashtable $\mathbf{P}_t$ as:
$$\mathbf{P}_t(z) = \begin{cases} max\{i | i \in \mathcal{J}\} & \text{if } \mathcal{J} \neq \emptyset \\ undefined & \text{otherwise} \end{cases} \tag{1.3}$$
We can note that $\mathbf{P}_t$ is time dependent and that $\mathbf{P}_t(z)$ is empty in case a miss occurs;

- **mapping B:** we define a mapping from the trace indexes $0, ..., L-1$ to $\{0, 1\}$. $\mathbf{B}$, which is time dependent as $\mathbf{P}$, intuitively flags all the references that are already in the stack at time t. So $\mathbf{B}_t(i)$ is:
$$\mathbf{B}_t(i) = \begin{cases} 1 & \text{if } \mathbf{P}_t(x_i) = i \\ 0 & \text{otherwise} \end{cases} \tag{1.4}$$
$\mathbf{B}$ is hence an alternative representation of the stack directly mapped on the address trace: $\mathbf{B}_t(i) = 1$ only for the elements x_i of $\boldsymbol{x}$ that are currently stored in Σ_t^{LRU};

- **stack distance Δ_t:** with $\mathbf{P}$ and $\mathbf{B}$, we can now define the stack distance of an element x_t: it is the number of 1's between the last reference to x_t and the current index t.
$$\Delta_t = \begin{cases} |\mathcal{H}| & \text{if } \mathbf{P}_t(x_t) \text{ is defined} \\ +\infty & \text{otherwise} \end{cases} \tag{1.5}$$
where $\mathcal{H}$ is defined as:
$$\mathcal{H} = \{i | \mathbf{P}_t(x_t) \leq i < t \wedge \mathbf{B}_t(i) = 1\}$$

Intuitively, $\mathcal{H}$ is the subset of trace indexes between $\mathbf{P}_t(x_i)$ (i.e. the last position where x_i is referenced) and the current index t whose $\mathbf{B}$ values are 1.

The LRU stack algorithm problem can then be redefined as follows without using the stack:

while $(0 \leq t < L)$:

- **search:** compute $\mathbf{P}_t(x_t)$;
- **count:** compute the stack distance Δ_t. If $\mathbf{P}_t(x_t)$ is undefined, the stack distance is set to $+\infty$;
- **update:** change $\mathbf{B}$ and $\mathbf{P}$ as follows:

$$\mathbf{B}_{t+1}(i) = \begin{cases} 1 & \text{if } i = t \\ 0 & \text{if } i = \mathbf{P}_t(x_t) \\ \mathbf{B}_t(i) & \text{otherwise} \end{cases}$$

$$\mathbf{P}_{t+1}(z) = \begin{cases} t & \text{if } z = x_t \\ \mathbf{P}_t(z) & \text{otherwise} \end{cases}$$

Having defined the alternative data structures, the stack algorithm just needs to keep $\mathbf{B}$ and $\mathbf{P}$ updated, and compute the stack distances as defined in Equation 1.5. However, to achieve time $T = O(L \log V)$ we need a more efficient way to evaluate Δ_t. An algorithm that reaches this bound is the *hole-based algorithm*.

We define a *hole* as a memory reference that is *not* the latest reference to a certain memory location at time t. Hence the holes are the 0's in $\mathbf{B}_t$. Holes are of interest since, contrary to ones that are constantly created and destroyed, they have the property of never being destroyed once they have been created. The stack distance at time t can then be expressed as

$$\Delta_t = t - \mathbf{P}_t(x_t) - holes_t(\mathbf{P}_t(x_t)) \tag{1.6}$$

where $holes_t(i)$ is the number of zeros in the address trace $\boldsymbol{x}$ between indexes i and t. This means that we are counting the 0's instead of the 1's in $\mathbf{B}$, and adjusting Equation 1.5 to this new strategy.

Holes are more easily represented than last references, and will be stored in an *interval tree*. This data structure is used to find intervals that overlap

with another given interval or point in an efficient way. These intervals need to be mutually disjunct and ordered in a set $I = \{[i_{11}, i_{12}], [i_{21}, i_{22}], ..., [i_{n1}, i_{n2}]\}$. In our case, the intervals in I are bounded by natural numbers that are indexes in the address trace; therefore, each interval represents a contiguous set of indexes t that are 0's in the trace.

Interval trees (Figure 1.6) can be represented as quasi-balanced *Binary Trees* **BT** such as the red-black trees ([6]). Each node $\mathbf{n}$ will then store a closed interval $[k_1(\mathbf{n}), k_2(\mathbf{n})]$. These intervals, contained in I, regulate the ordering of the tree: defining as $left(\mathbf{n})$ and $right(\mathbf{n})$ the left and right children of $\mathbf{n}$ respectively, we have that $k_1(\mathbf{n}) < k_2(left(\mathbf{n}))$ and $k_2(n) < k_1(right(\mathbf{n}))$.

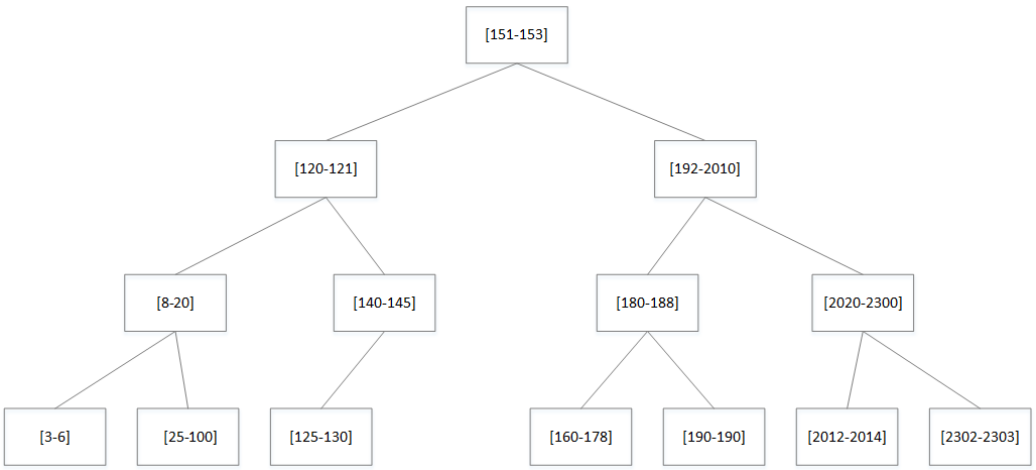


Figure 1.6: An interval tree.

As shown in Equation 1.6, we can use the interval tree to compute Δ_t by counting the number of holes between $\mathbf{P}(x_t)$ and t. However, since there are certainly no holes beyond the current index t, our task is just to count the number of holes beyond index $\mathbf{P}(x_t)$. To this end, we need to store in each node $\mathbf{n}$ of the tree the number of holes $sum(\mathbf{n})$ contained in its two children. The tree has now become equivalent to the partial sum hierarchy of Bennet and Kruskal.

An optimization that reduces the number of dereferentiations on the right subtree is to make $sum(\mathbf{n})$ store only the number of holes in $right(\mathbf{n})$, instead of $\mathbf{n}$ itself.

In Figure 1.7 the shaded boxes store the number of holes in their right subtree, as indicated by the dashed arrows. The cells marked with * will not need to be dereferenced during the tree traversal. To count the number of holes after index $\mathbf{P}(x_t)$, we traverse the tree from top to bottom, searching for

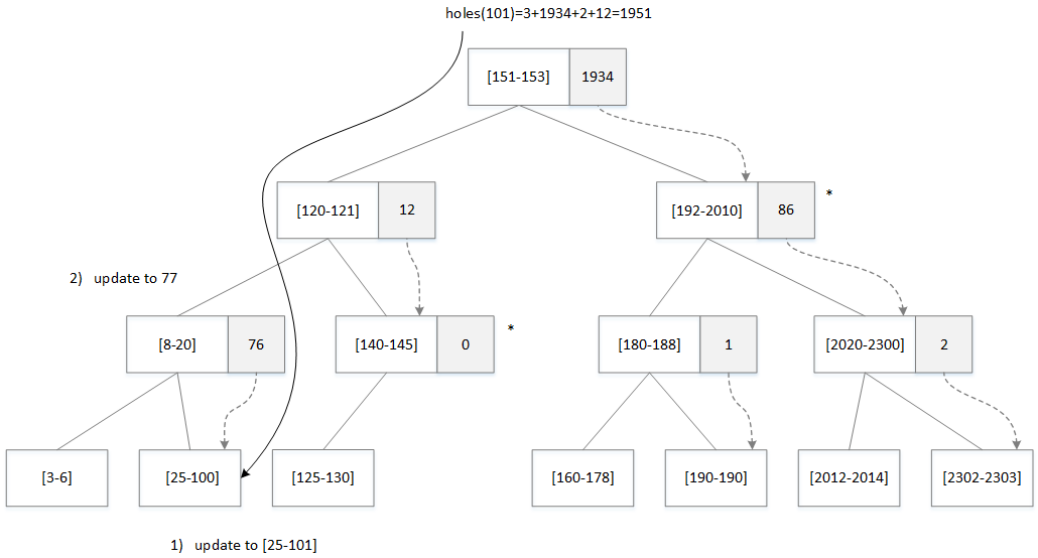


Figure 1.7: Updating the interval tree.

the leaf node closest to $\mathbf{P}(x_t)$. We then sum all the holes of the *right* subtrees encountered along the path (i.e. those with indexes larger than $\mathbf{P}(x_t)$).

The **update** procedure of the LRU stack algorithm needs to consider three cases, when we insert a new hole p into the tree:

- if p is adjacent to a stored interval $[k_1, k_2]$ (i.e. $p = k_1 - 1$ or $p = k_2 + 1$), the interval is updated to include p;
- if p is adjacent to two existing intervals $[k_1, p-1]$ and $[p+1, k_2]$, the two intervals are merged into a single $[k_1, k_3]$. This operation deletes one of the two nodes and may trigger a rebalance of the tree;
- if p is not adjacent to any interval in the tree, a new node is created to store the interval $[p, p]$. A rebalance may be needed.

Lastly, to update the partial sum hierarchy we need to change the values of the sum of the holes in $right(\mathbf{n})$ of all the nodes on the path from the root to the leaf. The whole operation is shown in Figure 1.7.

The algorithm, based on a quasi-stationary binary tree, takes time $O(\log V)$ to find Δ_t. The total run time is hence $T = O(L \log V)$ and space $S_{aux} = O(V)$ is needed to store the binary tree.

1.5 Objective of the thesis: parallel computation

Parallel computing has been investigated since the late 1950s, and has recently become part of mainstream computing. Intuitively, the idea is to use more processors, or cores, to simultaneously carry out the execution of several processes. This is feasible when the problem to be solved can be easily divided in independent sub-problems, whose results will be combined to obtain the final solution once they have all been solved. The *divide and conquer* paradigm we have just described can be easily integrated with parallel computing, since we can assign each of the sub-problems to a different processor to speed up calculations.

There is however a trade-off to be aware of, since coordinating the work of the processors and moving the data between them comes with a temporal cost: the time of communication T_{comm} can sometimes even exceed the time T_{exec} needed to actually solve the problems. This means that simply using more processors will not automatically yield better performances, and since the total execution time $T_{tot} = T_{exec} + T_{comm}$, we need to strike a balance between these two times.

One more aspect worth of consideration is the usage of the processors. Let us suppose we have N CPUs available, so that we can solve N sub-problems on the first iteration. In most cases only one CPU can work on a given amount of data; this means that in the *conquer* phase, only a subset of those N CPUs, usually $N/2$, will be used again in the second iteration. $N/4$ processors will be used in the third cycle and so on. It is clear that there is a waste of resources, since we use our full computing capability only once in the whole execution; it is therefore important to devise a strategy that efficiently utilizes the available CPUs.

Let us now produce an example to show the usefulness of the parallel approach on this type of problem. Let us suppose we want to run an online simulation of a replacement policy on an address trace with $V = 1000000$ pages. In this hypothetical scenario, we are equipped with two hardware cores: the first one will generate the address trace of a given program, while the second one will run the simulation of the LRU policy and compute the stack distances. Qualitatively, we can estimate that for each memory access of the program we are observing, the algorithm that studies its trace can do around $5 \log V$ operations. Let us suppose the two cores run at the same speed; the simulation will be about 100 times slower than the trace generation. If we want to carry out an on-line analysis, we will have to compensate this disparity using parallelism. While we would need at least 100 processors

to keep up with the program, a more realistic number would be around 250 cores.

A final remark worth mentioning is that parallelism is the only solution we have to tackle problems of machine simulation. Solving this type of problems allows us to build better machines, which in return will be able to examine more complex programs. In this virtuous cycle, both our capabilities to analyze information and to generate it will grow. However, the length L of those programs as well as their number of pages V will increase, while our improved computational power can only compensate for one of these two factors, if we want to run a real-time analysis. By solving more complex problems we also generate more difficult ones, and parallelism is the answer to keep up in this vicious cycle.

1.5.1 Performance objective: $T = O((L \log V)/P)$

As we have seen in Section 1.4.2, using a binary tree it is possible to compute the LRU stack distance in time $T = O(L \log V)$. This result is obtained using a sequential algorithm; what we set out to achieve in this thesis is the implementation of a parallel algorithm that, using P processors, can achieve $T = O((L \log V)/P)$. Let $S_P(n)$ be the *speedup* obtained using P processors on a problem of size n:

$$S_P(n) = \frac{T^*(n)}{T_P(n)} \tag{1.7}$$

In equation 1.7 we labeled the complexity of the best sequential algorithm as $T^*(n)$ and as $T_P(n)$ the total run time of a parallel algorithm which uses P processors. We consider optimal a speedup proportional to the number of processors used, hence in our case we have that:

$$S_P(n) = \frac{O(L \log V)}{O((L \log V)/P)} \propto P$$

In other words, we hope to achieve a speedup proportional to P using a parallel approach to the stack distance evaluation problem.

1.5.2 Obstacles to be overcome

Let us examine the linear time sequential algorithm that computes the stack distance in time $T = O(LV)$; this algorithm could be easily sped up if we were given $P = V$ processors to use, since we could simply assign each of the V processors to a single position $\Lambda(i)$, $i = 1, ..., V$, of the LRU stack Σ^{LRU}. When a new page x_t appears in the address trace, all the V processors can

simultaneously check if their own position contains x_t. If the page is found, with a *broadcast* message they can inform the other processors and update the stack accordingly, moving the data between them. With a *Parallel Random Access Machine* (PRAM), this whole operation could be done in constant time, thereby achieving a total temporal complexity of $T = O(L)$. The main issue of this approach is that we are trying to parallelize an inefficient version of the algorithm: with V processors we obtain $T = O(L)$, while the binary tree version achieves $T = O(L \log V)$ using only one processor.

To truly take advantage of the parallel approach we hence have to focus on the logarithmic version of the algorithm. Our goal is to devise a strategy that allows us to work on different segments of the address trace at the same time, since the data structure already operates in logarithmic time. Our choice is furthermore justified by the realization that it is not easy to parallelize the tree traversal, which is the core of the logarithmic algorithm. There are however some complications with the divide and conquer approach we are hoping to utilize, as we are going to explain next.

To the best of our knowledge, the problem of devising a parallel algorithm for stack distance computation has only been tackled in [7], which proposes an ad-hoc solution for the LRU policy. A possible reason for such an occurrence is that this type of problem is not easy to parallelize. Let us recall the address trace $\mathbf{x}$ and its length L, while V is the number of distinct pages in $\mathbf{x}$; to determine the exact composition of the buffer B_t at all times $t = 1, ..., L$, we need to know the past history of the trace, that is which pages have been brought inside and outside each buffer in all the previous steps. Intuitively, if we want to compute the stack distances of address trace $\mathbf{x}$ starting from $t = L/2$, the first obstacle we encounter is the lack of knowledge of the buffer state $B_{L/2}$, which is needed as shown in Figure 1.5. At first glance, only a sequential algorithm seems to be able to satisfy this requirement, that is a procedure that computes the buffer state in each moment $t = 1, ..., L$. This means that simply dividing the trace between P processors, so that each one can independently evaluate the stack distances of a segment of length L/P pages, will not work, because we do not know the buffer state at the beginning of each segment, except for the first one.

To summarize, the $O(LV)$ algorithm is inefficient, but its inefficiency on V can be mitigated by parallelism, with a trade-off involving a significant number of processors; the $O(L \log V)$ algorithm instead is more work-efficient, but the single update step is hard to parallelize.

1.6 Related work

Parda was presented in 2012 ([7]) as the first parallel algorithm for stack distance computation. While the algorithm presented in this thesis employs a red-black tree as its core structure to simulate the stack, Parda uses a *splay tree*. This particular tree is specifically designed to simulate cache behavior, since recently accessed elements are positioned near the top of the tree and are quick to access again. Basic operations of SEARCH, INSERT and DELETE all take time $O(\log n)$, but the height of the tree in the worst case, though unlikely, is $O(n)$, which happens when all the n elements are accessed in a non-decreasing order.

Parda is an algorithm specifically devised for the computation of reuse distances (also known as LRU stack distances), namely it is tailored for the LRU replacement policy. The strategy presented in this work instead takes advantage of *prefix computation*, and has the potential to be developed for the computation of the stack distances under different replacement policies.

Chapter 2

Finite state machine evolution and prefix computation

In this chapter we will build the theoretical framework used in the remainder of the thesis. Our objective is the reduction of state machine evolution to prefix computation, since we already have efficient parallel algorithms to compute the latter.

The sequential algorithms we have described in Section 1.4.1 and 1.4.2 can be thought of as the activity of the following machine:

- **input:** the address trace $\boldsymbol{x} = (x_1, x_2, ..., x_t)$ up to time t;
- **output:** the stack distance Δ_t;
- **state:** the stack Σ_t^{LRU}, which stores informations about the trace up to time t.

Information stored in the states is later used to correctly handle future pages of the address trace. We also observe that the very description of LRU stack distance given in Section 1.3 can be interpreted as the definition of a *finite state machine*. From the classes on parallel computing by professor Gianfranco Bilardi, we realize that the problem of parallelizing the evolution of a finite state machine has already been studied. This means that we know a general theory to deal with our problem, via reduction to prefix computation on a suitable *semigroup*. However, if we were to directly insert all possible transition functions between states in our semigroup, we would soon realize that this semigroup would count $|Q|^{|Q|}$ elements, were $|Q|$ is the number of states of the machine. Therefore, we want to verify if the semigroup we actually need is smaller than the semigroup containing every transition function. Our objective is hence twofold:

- inspect the minimum semigroup generated by the *one-step transition functions*, which we will introduce in Section 2.5;
- find a suitable compact representation for the elements of the semigroup to efficiently execute operations on said semigroup (Section 3.2).

Moreover, we can reasonably expect that the complexity of the semigroup will somehow depend on the complexity of the finite state machine we are simulating (i.e. if the machine is composed of two states, it is sensible to presume that the corresponding semigroup will not be too complex).

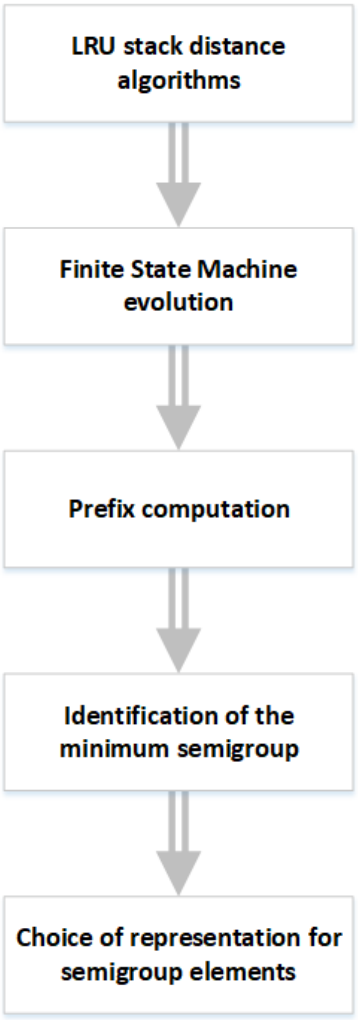


Figure 2.1: Schematics of the problems to tackle.

In Figure 2.1 is shown the path that leads to the problem of the choice of the semigroup. To summarize, it is already known how to set up the sim-

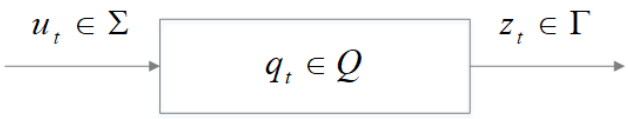
ulation of a finite state machine as a prefix computation problem on a given semigroup. We can run this simulation for N steps in $T = O(N/P)$, where $P \leq N \log N$ is the number of processors we can use. However, the constant in the big O notation depends on which specific machine we are using, and in particular on its number of states. Since this constant would be $|Q|^{|Q|}$, the problem of parallelizing the LRU stack distance computation would become essentially intractable; therefore, devising a minimum semigroup for our machine is a problem yet to be tackled and is the core of this thesis.

2.1 Definition of finite state machine

A finite state machine (FSM), or simply a *state machine*, is an abstract machine that is used as a computational model. A FSM is composed of *states* and *transitions* between states, the latter being triggered by input symbols from a given *alphabet*. A FSM is hence defined by its set of states, its initial state and the conditions for each transition.

An important FSM is the ***Mealy machine*** $\boldsymbol{M = (\Sigma, Q, \Gamma, \delta, \eta)}$, defined as follows:

- $\boldsymbol{\Sigma}$ the input alphabet;
- $\boldsymbol{Q}$ the set of states;
- $\boldsymbol{\Gamma}$ the output set;
- $\boldsymbol{\delta : Q \times \Sigma \to Q}$ the state transfer function;
- $\boldsymbol{\eta : Q \times \Sigma \to \Gamma}$ the output transformation function.



$$q_{t+1} = \delta(q_t, u_t)$$
$$z_t = \eta(q_t, u_t)$$

Figure 2.2: The FSM used for prefix computation.

In the Mealy machine, at time t, both the next state q_{t+1} and the output z_t depend on the current state q_t as well as on the current input u_t.

Since the memory of a FSM is limited by its number of states, this model has less computational power than other models such as the *Turing Machine*. Nonetheless, FSMs are used in many applications ranging from hardware to software (i.e. the lexical analyzer and parser of a compiler are implemented using several state machines).

2.2 The FSM corresponding to LRU stack

Following the definition of finite state machine, we immediately provide an example of the FSM we will use in the LRU stack distance problem. The state machine $\boldsymbol{M}$ is the same quintuple $\boldsymbol{(\Sigma, Q, \Gamma, \delta, \eta)}$ already defined in Section 2.1, where:

- $\boldsymbol{\Sigma} = \{\textit{set of trace addresses}\}$ where $|\Sigma| = V$;
- $\boldsymbol{Q} = \{(v, \Lambda) : v \in \{0, 1, ..., V\}, \Lambda = (\Lambda(1), \Lambda(2), ..., \Lambda(v))\}$ where v is the number of different addresses encountered up until now and Λ is the sequence of v distinct addresses;
- $\boldsymbol{\Gamma} = \{1, 2, ..., V, +\infty\}$ is the set of stack distances;
- $\boldsymbol{\delta((v, \Lambda), u)} = \begin{cases} (v+1, (u, \Lambda)) & \text{if } u \notin \{\Lambda\} \\ (v, (u, \Lambda(1), ..., \Lambda(i-1), \Lambda(i+1), ..., \Lambda(v))) & \text{if } u \in \{\Lambda\} \end{cases}$
  where $u = \Lambda(i)$ if $u \in \{\Lambda\}$;
- $\boldsymbol{\eta((v, \Lambda), u)} = \begin{cases} +\infty & \text{if } u \notin \{\Lambda\} \\ i & \text{if } u = \Lambda(i) \end{cases}$

In the previous definitions we have used the notation introduced in Section 1.3

2.3 Definition of prefix computation

Let us define

$$< A, \bullet >$$

where A is a finite set and $\bullet$ is an undefined operation on A, that is $\bullet : A \times A \to A$. We will consider *associative* operations, that is $\bullet$ satisfies the associative law. Hence we have that

$$\forall x, \forall y, \forall z, (x \bullet y) \bullet z = x \bullet (y \bullet z)$$

Under these conditions, $< A, \bullet >$ is a *finite semigroup*.
We define prefix computation the following problem:

input: $\boldsymbol{x} = (x_0, x_1, ..., x_{N-1}) \in A^N$;

output: $\boldsymbol{y} = (y_0, y_1, ..., y_{N-1}) \in A^N$, such that $y_0 = x_0$ and $y_j = y_{j-1} \bullet x_j$, $j = 1, ..., N-1$.

This means that $y_j = x_0 \bullet x_1 \bullet ... \bullet x_j$, and the problem requires the computation of the products of each prefix of the input, hence the name prefix computation. Let us provide an example of prefix computation: let

$$A = \{0, 1, 2, 3\}$$

be the finite set and let

$$a \bullet b = (a + b) mod 4$$

Furthermore, let $\boldsymbol{x} = (3, 1, 2, 1, 0, 3)$ be our input sequence; our output will be the sequence

$$\boldsymbol{y} = (3, 0, 2, 3, 3, 2)$$

We can see that, as defined above, $y_0 = x_0 = 3$, $y_1 = y_0 \bullet x_1 = (3+1)_4 = 0$ and so on. Finally, let us consider prefix y_4 to show the associativity of the semigroup operator: $y_3 = (3+1+2+1)_4 = 3$ and $y_4 = (3+1+2+1+0)_4 = (7)_4 = y_3 \bullet x_4 = (3+0)_4 = 3$.

2.4 Semigroup of a machine

We now disclose the standard semigroup we can make use of under any circumstances. This semigroup is not as efficient as the one we will utilize for the LRU problem, which will be introduced in Section 3.1, but it can be employed without the need for a thorough analysis of the specific problem to tackle. The standard semigroup

$$< f(x), \circ >$$

is composed of functions $f(x) : Q \to Q$ and by the composition operator $\circ$ of functions, which satisfies the associative law. This semigroup satisfies the *closure property*, namely it is closed under $\circ$, since the composition of

two functions $f(x)$ and $g(x)$ will always yield another function $(f \circ g)(x) = f(g(x))$.

The formal justification behind this standard semigroup stems from *Cayley's theorem*. Knowing that an invertible function in a finite set is a *permutation*, the theorem states that every finite group is isomorphic to a group of permutations. The analogous theorem for the semigroup is as follows:

Theorem 1. *Each finite semigroup is isomorphic to a semigroup of functions with a composition operator.*

Hence the standard semigroup, and FSMs by extension, allows us to represent any finite semigroup.

2.5 Reduction of state evolution to prefix computation

The problem of state evolution can be defined as follows:

> **input:** $q_0, u_0, u_1, ..., u_{t-1}$;
>
> **output:** $q_1, q_2, ..., q_t$, such that $q_{t+1} = \delta(q_t, u_t)$.

In this description, q_0 is the initial state of the machine, while $u_0, ..., u_{t-1}$ are the inputs from the alphabet at times $0, ..., t-1$. We can see that $q_1 = \delta(q_0, u_0)$, $q_2 = \delta(q_1, u_1)$ and so on; our objective is to overcome this dependency on previous states. Similarly to what happens in binary addition, while we have to wait for previous carries it is still possible to simultaneously work on the most significant bits. In order to achieve such a result, we define *one-step transition functions*

$$\delta_a(q) \triangleq \delta(q, a), a \in \Sigma$$

which are a simple change of notation. If we set a and let q be free to change, we have that $\delta_a : Q \to Q$. Hence

$$q_t = \delta_{u_t}(\delta_{u_{t-1}}(\delta_{u_{t-2}}(...\delta_{u_0}(q_0)...)))$$

Defining $f(g(x)) = (g \circ f)(x)$ which is another change of notation, we finally obtain

$$q_t = (\delta_{u_0} \circ \delta_{u_1} \circ ... \circ \delta_{u_t})(q_0) \tag{2.1}$$

In this equation, $\circ$ is the composition of functions, already introduced in Section 2.4; we can therefore combine the functions in the previous equation as we please.

With this result we have reached our objective, since we can now work in parallel on different parts of the sequence of δ_a thanks to the associative property, while we only need the initial state q_0 to find the state of the machine at any moment. We have thus reduced the evolution of a FSM to the prefix computation problem of Section 2.3.

2.5.1 Output computation

Starting from the problem defined in Section 2.5, we are now interested in finding the output of prefix computation in the FSM framework. Namely, we have to find a way to determine any state q_i of the machine knowing only the initial state q_0. With Equation 2.5 at hand, this becomes a simple task, since any state q_i can be expressed as

$$q_i = (\delta_{u_0} \circ \delta_{u_1} \circ ... \circ \delta_{u_i-1})(q_0)$$

We recall that the composition operator $\circ$ satisfies the associative law, so this sequence of one-step transition functions can be evaluated in any order we see fit. This result will be of crucial importance in the remainder of the thesis.

2.6 Analysis of the machine semigroup and choice of representation

The main drawback of the semigroup described in Section 2.4 is its dimension. More specifically, in the context of the LRU stack distance problem, the number of functions is directly related to V, the number of pages in the address trace. The number of different states is $|Q| = V!$; this means that functions $f(x)$ from a state to another total $|Q|^{|Q|} = V!^{V!}$. Nowadays, it is not uncommon for programs to use hundreds of thousands if not millions of different pages. With $V = 10$ using only ten different pages, we would have $3628800^{3628800}$ different functions in our semigroup, which clearly make the problem intractable.

We now have to choose a suitable representation for the elements of the semigroup. It is worth noting that domain and codomain of the functions of the standard semigroup coincide, since $f(x) : Q \to Q$ is a function from

state to state. We can assume to already have a representation of the set of states $\boldsymbol{Q}$ of the state machine, for instance via binary strings. To represent the functions, we can firstly establish an ordering on $\boldsymbol{Q}$, i.e. $\{Q_1, Q_2, ..., Q_r\}$ and consider it our domain. In this case, the codomain will simply be the list $\{F(Q_1), F(Q_2), ..., F(Q_R)\}$. If, instead, no ordering is established on the set of states, a straightforward solution is to provide the list of couples $\{(Q_1, F(Q_1)), (Q_2, F(Q_2)), ..., (Q_r, F(Q_r))\}$. It is obvious that the cardinality of this list is $|Q|$. This can be considered the analogous of the *table of truth* of a boolean function.

To save some space if an ordering is established, it is possible to determine the exact value of the function from the list $\{F(Q_1), F(Q_2), ..., F(Q_r)\}$. Since in the LRU problem the set $\boldsymbol{Q}$ is generic, it is convenient to represent the functions via list of couples, even if establishing an ordering would allow us to find $F(Q)$ in logarithmic time instead of employing a linear search.

2.7 Algorithms for semigroup multiplication under the chosen representation

Devising an algorithm for semigroup multiplication is a straightforward task if we are dealing with a boolean function; having available its table of truth, via Karnaugh maps it is possible to synthesize a hardware circuit using the minimum number of AND and OR gates.

However, in most cases, functions from states to states will not be strictly boolean. In Section 2.6 we have discussed two viable solutions to represent a generic function:

- $\boldsymbol{\{Q_1, Q_2, ..., Q_r, F(Q_1), F(Q_2), ..., F(Q_r)\}}$ if an ordering is established, namely we simply list domain and codomain in sequence;
- $\boldsymbol{\{(Q_1, F(Q_1)), (Q_2, F(Q_2)), ..., (Q_r, F(Q_r))\}}$ which is the list of couples already mentioned in the previous section.

Devising the semigroup multiplicator means building the block that will perform the $\circ$ operation in our prefix computation circuit. We hence have to conceive a strategy to realize the composition $f(g(x))$ of two functions $f(x) : Q \to Q$ and $g(x) : Q \to Q$. Intuitively, the most immediate approach is to substitute in both representation $F(Q_i)$ with its composition $F(G(Q_i))$. Thus we obtain

- $\boldsymbol{\{Q_1, Q_2, ..., Q_r, F(G(Q_1)), F(G(Q_2)), ..., F(G(Q_r))\}}$ if an ordering is established;

- $\{(\boldsymbol{Q_1}, \boldsymbol{F}(\boldsymbol{G}(\boldsymbol{Q_1}))), (\boldsymbol{Q_2}, \boldsymbol{F}(\boldsymbol{G}(\boldsymbol{Q_2}))), ..., (\boldsymbol{Q_r}, \boldsymbol{F}(\boldsymbol{G}(\boldsymbol{Q_r})))\}$ otherwise.

Analogously to the reasoning at the end of the previous section, it is possible to save half the space by listing only the final states if an ordering is established on the set $\boldsymbol{Q}$, namely $\{F(G(Q_1)), F(G(Q_2)), ..., F(G(Q_r))\}$.

2.8 Review of parallel algorithms for prefix computation

In this section we analyze the *twisted reflected tree* (TRT), that was introduced in [8] and which allows parallel computation of the prefixes of a sequence. The advantage of this type of structure is that it reuses already computed prefixes when they are needed, instead of treating them as independent entities. A schematic of the TRT is given in Figure 2.3.

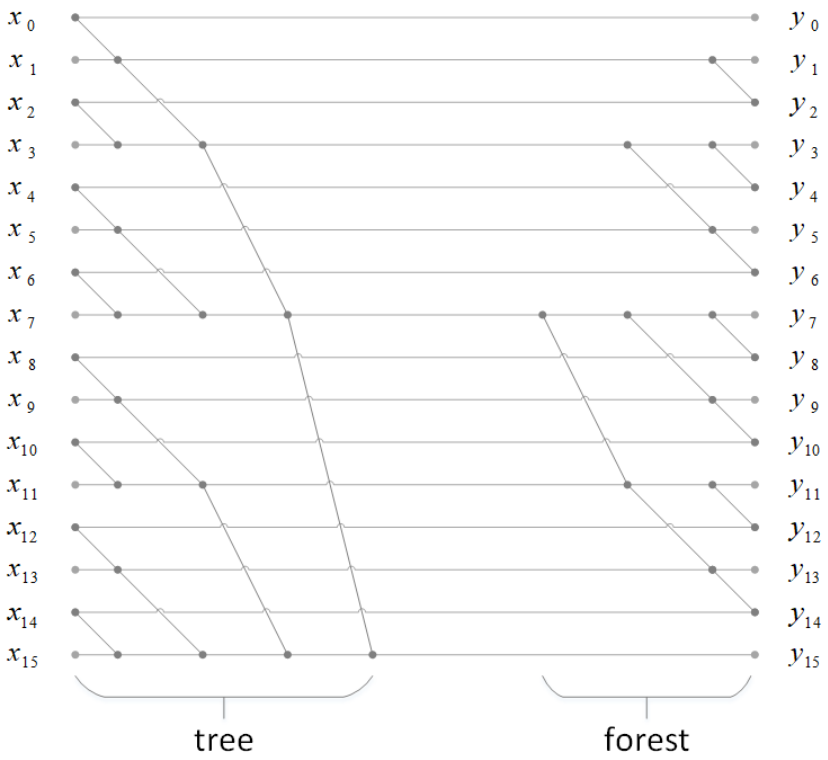


Figure 2.3: The TRT of a sequence of 16 elements.

In this network there are different types of nodes:

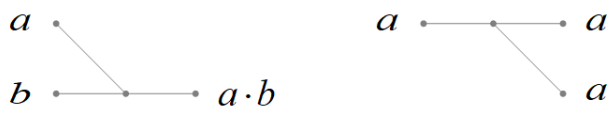


Figure 2.4: Operational nodes.

- **input nodes** with no incoming edges are associated to the sequence $\boldsymbol{x}$, and make up the input of the TRT;
- **output nodes** with no leaving edges are the output of the network, associated to the sequence $\boldsymbol{y}$ of prefixes;
- **operational nodes** divided in the two categories shown in Figure 2.4:
  - the nodes with two input edges and one output (left side of the figure) are the so called *semigroup multiplicators* (Section 2.7), which realize the $\circ$ operation, that is $x_t \circ y_{t-1} = y_t$;
  - the nodes with two output edges simply split their input (right side of the figure).

Each one of the multiplicators can be assembled with a boolean circuit and a flip flop to store its state, but first of all it is important to choose a suitable representation for the elements of the semigroup. Let us finally recall that the $\circ$ operation is associative, but does not necessarily satisfy the commutative law, so we can not switch the order of the input edges in the nodes.

As an example, let us compute y_6, the prefix of the first seven elements of the input sequence $\boldsymbol{x}$: $y_6 = x_0x_1x_2x_3x_4x_5x_6$. We can note that the first four elements form the prefix y_3, and that $x_4 \circ x_5$ is already computed by the network as well. This means that instead of calculating six products, thanks to the associative property we only need to compute two of them, because $y_6 = (y_3)(x_4x_5)(x_6)$. As shown in Figure 2.3, the product of y_3 and $x_4 \circ x_5$ is linked by the network to the semigroup multiplicator that outputs y_6, which will just have to compute the last product.

In the TRT, the number of operations is $|V| < 2N$ while the critical path L is $L \leq 2\log_2 N$.

Chapter 3

The FSM-prefix framework for LRU stack computation

In this chapter, which makes up the core of the thesis, we present the semigroup to tackle the LRU stack distance problem. The state machine corresponding to said problem has already been presented in Section 2.2; starting from this result, we devise a suitable semigroup and a compact representation of its elements. The LRU semigroup represents the conjunction between the theoretical framework built in Chapter 2 and the algorithm presented in Section 3.3.

The FSM-prefix computation strategy we employed theoretically allows us to use the same TRT scheme to calculate the stack distances under different replacement policies: it is sufficient to find a suitable semigroup and the appropriate multiplication to perform at every node of the tree. This aspect constitutes the main difference between our work and [7], since the latter is entirely conceived to work with the LRU policy.

Two distinct experiments could give us some insight into the actual performances of the two solutions:

- **compare communications:** since both Parda and our algorithm employ Message Passing Interface (MPI) to allow communications between processors, it would be interesting to compare the results of these two approaches regarding data exchanges;
- **exchange trees:** although the red-black tree we used grants that its height will always be $O(\log V)$, the splay tree employed in Parda could work quite as well in the stack distance computation problem, while allowing easier maintenance.

Intuitively, our more general approach would not achieve the same efficiency as the ad-hoc solution already known.

3.1 The semigroup

As we explained in Section 2.4, the main issue with the standard semigroup for the LRU problem lies in its dimension: the number of transition functions between states totals $V!^{V!}$ elements, which would make the stack distance evaluation intractable. The semigroup we devised does not represent the entire semigroup of functions, but only a fraction of the whole set. This subset is sufficient to tackle our problem, since we realize that transition functions between states are either the already mentioned one-step transition functions or a composition of the former. Hence, the finite set of our semigroup is composed of

- **one-step transition functions**, or *generators*, of the semigroup, that is transition functions δ_u associated to a single input u;
- **compositions of generators**, namely transition functions resulting from the multiplication of several δ_u;
- **an associative operator**, which is again the composition of functions $\circ$.

By construction, the semigroup we have just described is a sub-semigroup. Let us analyze the cardinality of the set of functions $\boldsymbol{\delta_{LRU}}$:

$$\boldsymbol{\delta_{LRU}} = \sum_{k=0}^{V} \left(\begin{array}{c} V \\ k \end{array} \right) k! = \sum_{k=0}^{V} \frac{V!}{(V-k)!} < e^{V!}$$

where V is again the number of pages in the address trace, while index k indicates the number of different pages encountered up until now in the current multiplication. As we can see, this intuition reduces the dimension of the semigroup from $V!^{V!}$ to $V!$, which is a remarkable saving.

This achievement stems from several observations:

- generators are idempotent, that is the composition $\delta_u \circ \delta_u$ does not alter the stack further after we apply the first function;
- composition of functions is not commutative, namely $\delta_a \circ \delta_b \neq \delta_b \circ \delta_a$, but the result of both transitions is the same except for the first two elements of the stack;
- if a generator δ_u appears multiple times in a composition, we can simplify the multiplication by considering only its rightmost occurrence.

The last intuition is a conjecture we will prove in Theorem 2. This observation allows us to simplify any given composition, which in the general case could be of arbitrary length, by considering only the last V different generators. Let us provide an example; we examine the following multiplication of generators:

$$\delta_a \circ \delta_c \circ \delta_a \circ \delta_f \circ \delta_c \circ \delta_a \circ \delta_d$$

If our conjecture holds, the former composition will yield the same final stack as this one:

$$\delta_f \circ \delta_c \circ \delta_a \circ \delta_d$$

As we will see in Section 3.2, each element of the semigroup is thus representable as a product of V distinct elements. Let us now provide a formal proof of our conjecture:

Theorem 2. *Let δ_i be the i-th generator, that is the one-step transition function that places i at the top of the stack and shifts down all the other pages by one position. Let also $\boldsymbol{\delta_{rand}}$ be an arbitrary sequence of generators. Finally, let $\boldsymbol{\delta} = \boldsymbol{\delta_{rand}} \circ \delta_a \circ \boldsymbol{\delta_{rand}} \circ \delta_a$ be the composition of a sequence of generators. If we are under LRU replacement policy, the transition function $\boldsymbol{\delta'} = \boldsymbol{\delta_{rand}} \circ \boldsymbol{\delta_{rand}} \circ \delta_a$ yields exactly the same final stack as $\boldsymbol{\delta}$.*

Proof. We are going to prove the previous statement by contradiction. Let us suppose that $\boldsymbol{\delta}$ and $\boldsymbol{\delta'}$ yield two different stacks Q and Q'. Let us now analyze the two transition functions to determine said final stacks. The symbol at the top of both buffers will be a, since LRU always places the last seen symbol on top. This means that the difference between Q and Q' will lie in the lower positions, due to sub-sequences $\boldsymbol{\delta_{rand}} \circ \delta_a \circ \boldsymbol{\delta_{rand}}$ and $\boldsymbol{\delta_{rand}} \circ \boldsymbol{\delta_{rand}}$ respectively. The difference between this two sequences is the generator δ_a, whose element a is already included in the resulting stack. Hence, the only way to obtain two different final stacks would be to change the order of composition of the generators in the two sub-sequences, but, since LRU does modify the ordering of the stack, we can only obtain two different final stacks by using another replacement policy. However, this contradicts our initial hypothesis of employing LRU replacement. □

To conclude this section we provide a visual example of how transition functions works:

In Figure 3.1 are shown two different transitions, resulting from applying different functions to the same initial state. Let us suppose $V = 5$, so that the set $\{A, B, C, D, E\}$ includes all the pages of our address trace. By examining the label of δ_{DB}, we automatically know which pages will be at the top of the final stack. Also notice that the order of pages A, C, E, not included in the

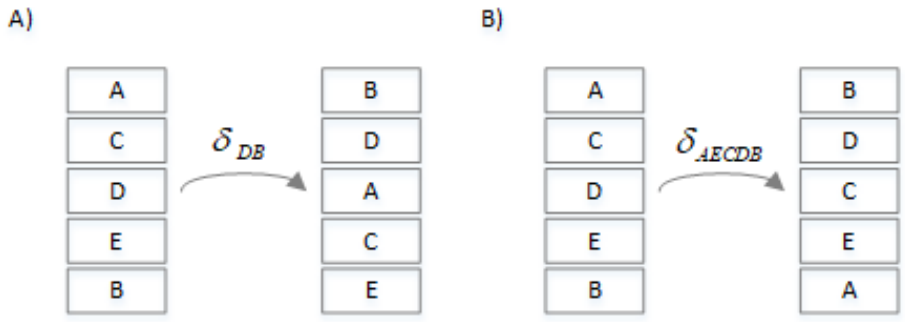


Figure 3.1: Stack update with two different functions.

label, will not be altered by the permutation. If we instead take a look at the label in Figure B), which includes all the V pages of our trace, we can observe that sequence A, E, C, D, B univocally specifies the exact composition of the final buffer, even if we had not known the arrangement of the pages in the initial stack.

3.2 Representation and multiplication

As explained in Section 3.1, once we realize that we do not need to represent every transition function between states, simplifying the semigroup is a straightforward task. We then need a suitable representation for the elements of said semigroup.

Let us recall from Section 1.3 that each stack, that is each state of the FSM, is a permutation of the elements of the address trace. Hence, in case the number of different pages V is set, all transition functions δ are functions from permutations to permutations. The representation we require will have to satisfy two requirements:

- it will have to be compact, that is it will have to employ the minimum number of bits possible;
- the composition of functions, namely the semigroup multiplication, will have to be efficient.

Firstly, we can observe that an input sequence $\boldsymbol{z} = z_1, z_2, ..., z_k$ univocally identifies a function $\delta_{z_1,z_2,...,z_k} : Q \to Q$. Therefore, we can recognize each function by simply labeling every transition δ with the shortest sequence $\boldsymbol{z}$ that leads to that function, that is to the final state reached when δ is applied to an initial state q_0. As highlighted in Section 3.1, this shortest sequence

will be a list of pages with no repetitions, that is a "simplified" sequence we name *sub-permutation*.

Let us now demonstrate that if an input sequence has no repetitions, namely it is a sub-permutation, no shorter sequence can lead to the same transition function δ.

Theorem 3. *Let $\boldsymbol{z} = z_1, z_2, ..., z_k$ be a sub-permutation and let $\delta_{z_1,z_2,...,z_k}$ be the transition function univocally identified by that label. If $z_1, z_2, ..., z_k$ has no repetitions, no input string shorter than $\boldsymbol{z}$ can lead to the same transition function $\delta_{z_1,z_2,...,z_k}$.*

Proof. We provide a proof by contradiction. Let $\boldsymbol{z'} = z_1, z_2, ..., z_j$ be a sub-permutation shorter than $\boldsymbol{z}$, that is $\boldsymbol{z'}$ lacks at least one page z_i compared to $\boldsymbol{z}$. Furthermore, let $\delta_{z_1,z_2,...,z_j}$ and $\delta_{z_1,z_2,...,z_k}$ be the two transition functions univocally identified by $z_1, z_2, ..., z_j$ and $z_1, z_2, ..., z_k$ respectively. If we assume this two functions to be identical, applying them to an initial state q_0 will yield the same final state, that is $\delta_{z_1,z_2,...,z_j}(q_0) = \delta_{z_1,z_2,...,z_k}(q_0) = q_1$. Intuitively, the labels of these functions are exactly the pages they will insert at the top of the final stack. However, since sequences $\boldsymbol{z}$ and $\boldsymbol{z'}$ differ by at least one element, the corresponding transition functions will lead to two different final states, unless there are duplicated pages in sequence $z_1, z_2, ..., z_k$. This contradicts our initial hypothesis, since we assumed sequence $\boldsymbol{z}$ to have no repetitions. □

To summarize, we can represent each state of our machine as a permutation of the V pages of the address trace, hence $|\boldsymbol{Q}| = V!$. However, our transition functions as well are labeled with sequences of at most V different pages, which in turn are permutations. It is hence important to bear in mind the double role played by permutations in the LRU semigroup, that is being both the elements of the set of states $\boldsymbol{Q}$ and the labels of functions from state to state.

We now focus on devising an efficient strategy to multiply functions of the semigroup. Firstly, let us recall the definition of composition of functions.

Given two functions $f : X \to Y$ and $g : Y \to Z$, we denote *composite function* the function $f \circ g : X \to Z$, defined by $(f \circ g)(x) = g(f(x)) \forall x \in X$.

The composition of functions satisfies the associative law, that is, given three function f, g and h with suitable domains and codomains, $(f \circ g) \circ h = f \circ (g \circ h)$.

Let us now define the *cleanup* of a sequence, which stems from our conjecture enunciated in Section 3.1.

Let $z_1, z_2, ..., z_k$ be an arbitrary sequence of pages. We define $cleanup(z_1, z_2, ..., z_k) = y_1, y_2, ..., y_h$ the sub-sequence of pages such that

$$z_i \in cleanup(z_1, z_2, ..., z_k) \Leftrightarrow \nexists z_j, i+1 \leq j \leq k : z_i = z_j$$

Hence, the cleanup of a sequence contains all the different single pages of the original sequence $z_1, z_2, ..., z_k$ and the rightmost occurrence of the repeated ones. Let us give an example; let $z_1, z_2, ..., z_k = 2, 3, 5, 4, 3, 5, 3, 8, 2$. Then

$$cleanup(2, 3, 5, 4, 3, 5, 3, 8, 2) = y_1, y_2, ..., y_h = 4, 5, 3, 8, 2$$

Having introduced the idea of cleanup, we can now define the multiplication of two sequences of generators.

Let $z_1, z_2, ..., z_k$ be a sub-permutation, that is a sequence with no repetitions. Let $\delta_{z_1,z_2,...,z_k}$ be the state transition univocally identified by $z_1, z_2, ..., z_k$. We define the composition of two functions identified by sub-permutations

$$\delta_{z_1,z_2,...,z_k} \circ \delta_{y_1,y_2,...,y_h}$$

as the new function

$$\delta_{cleanup(z_1,z_2,...,z_k,y_1,y_2,...,y_h)}$$

where $cleanup(z_1, z_2, ..., z_k, y_1, y_2, ..., y_h)$ will contain at most $h + k$ elements.

We now need to prove that any function identified by a sequence is equivalent to the transition labeled with the cleanup of that sequence, that is they both lead to the same final state when applied to an initial state q_0.

Theorem 4. *Let $z_1, z_2, ..., z_k$ be an arbitrary sequence of pages and let $y_1, y_2, ..., y_h = cleanup(z_1, z_2, ..., z_k)$. Furthermore, let $\delta_{z_1,z_2,...,z_k}$ and $\delta_{y_1,y_2,...,y_h}$ be the transition functions identified by $z_1, z_2, ..., z_k$ and $y_1, y_2, ..., y_h$ respectively. If we are under LRU replacement, $\delta_{z_1,z_2,...,z_k}$ and $\delta_{y_1,y_2,...,y_h}$ are equivalent functions.*

Proof. We will prove the former statement by contradiction. Let us suppose that $\delta_{z_1,z_2,...,z_k}$ and $\delta_{y_1,y_2,...,y_h}$ are not equivalent functions, that is, given an initial state q_0, $\delta_{z_1,z_2,...,z_k}(q_0) = q_1$ and $\delta_{y_1,y_2,...,y_h}(q_0) = q_2$, where q_1 and q_2 are two different final states. In Theorem 2 we have proven that, under LRU replacement, the composition of only the rightmost repeated generators of a sequence will yield the same final state as the multiplication of the original sequence. Since the cleanup of a sequence is exactly a sequence of symbols where every duplicated page is deleted except for its rightmost occurrence, the only way to obtain two different final states would be to employ a different replacement policy than LRU. This however contradicts our initial hypothesis. □

To conclude this section we provide a naive pseudocode to implement the composition of two functions $\delta_{z_1,z_2,...,z_k} \circ \delta_{y_1,y_2,...,y_h}$.

```
   Data: sequences z_1, z_2, ..., z_k and y_1, y_2, ..., y_h
   Result: cleanup(z_1, z_2, ..., z_k, y_1, y_2, ..., y_h)
 1 initialization;
 2 cleanup ← y_1, y_2, ..., y_h;
 3 while cleanup.length()≤ h + k do
 4 |   for i ← k to 1 do
 5 |   |   if z_i == any y_j then
 6 |   |   |   i- -;
 7 |   |   else
 8 |   |   |   cleanup ← z_i + cleanup;
 9 |   |   |   i- -;
10 |   |   end
11 |   end
12 end
13 cleanup.print();
```

Algorithm 1: Naive cleanup

3.3 The prefix algorithm for LRU stack computation

In this section we will describe the algorithm developed for the parallel computation of the stack distances. Since our implementation stems from the procedure described in Section 1.4.2, the underlying data structure for the

distance computation will be a red-black tree, that is a binary tree that satisfies the following ***red-black properties***:

1. Every node is either red or black.
2. The root is black.
3. Every leaf (an empty node NIL) is black.
4. If a node is red, then both its children are black.
5. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

The usefulness of this data structure lies in its balance, since the basic operations such as SEARCH, INSERT and DELETE all take time $O(\log n)$, where n is the number of elements in the tree. The red-black tree used in our algorithm is slightly different from the standard structure, since each node contains two key values *(key1,key2)* and a *sum* field, alongside the values of *color,left,right* and *parent*.

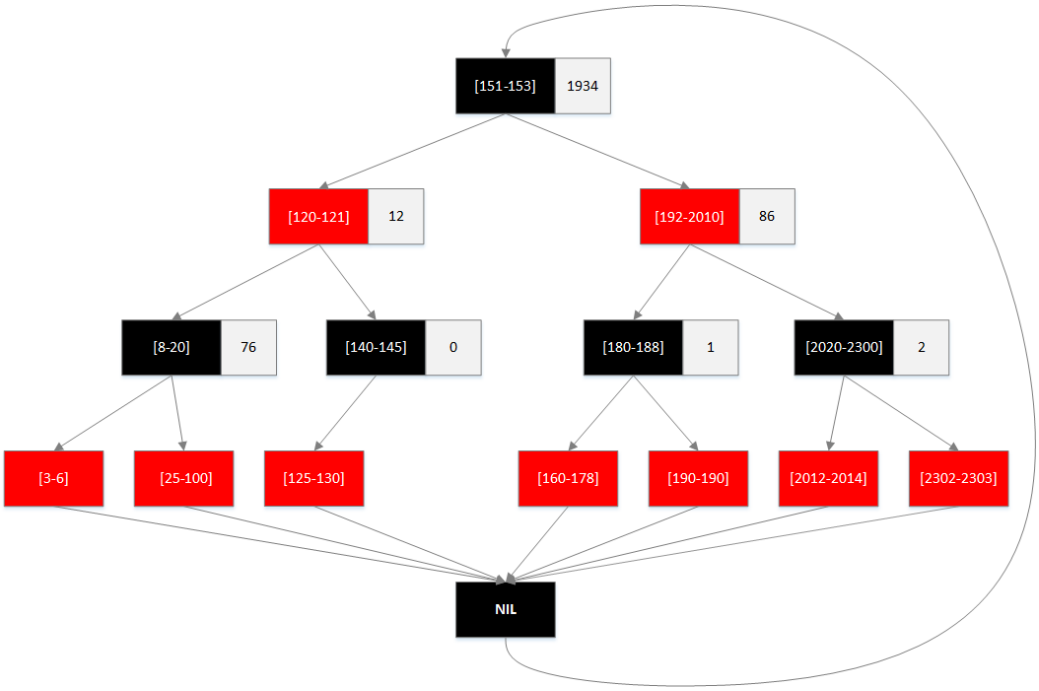


Figure 3.2: A red-black tree.

In Figure 3.2 is shown a possible red-black tree of the interval tree in Figure 1.7. Please note the double role played by the NIL node, acting as both left and right child of the leaves and as parent of the root node.

Besides the methods for the management of the tree, the core of our algorithm is composed by the procedure explained in [4] to calculate the number of holes in an interval. This is a key step in evaluating the stack distance, as shown in Equation 1.6, and it is coupled with a method to update the *sum* value of every node when the structure of the tree is modified, namely when the insertion or the deletion of a node takes place. With this set of methods, we are able to sequentially compute the stack distances on a trace of length L with V different pages in time $OL(\log V)$.

The joint employment of the twisted reflected tree (Section 2.8) and prefix computation on this sequential algorithm finally allows us to make use of more processors and achieve parallelism. For the sake of simplicity, we will consider an address trace of length L, where L is a power of 2, and we will describe the various operations performed by our algorithm with P processors (P power of 2 as well).

Algorithm description:

- **input split up:** knowing L and P, the address trace is divided in sections of equal length among the processors, so that each one will analyze only a segment of length L/P;
- **backtrack:** in this step, each processor computes the element of the semigroup of its segment (the $\boldsymbol{Q}$ in Section 2.2). To do so, each core traverses its assigned segment from the latest time to the least recent index, to find the most recent reference to each page. The result of this operation is an LRU stack computed on the sequence of addresses assigned to each processor; the final stack of every segment is stored in a hashtable, where each page is associated to its most recent index;
- **message passing:** the hashtable storing the stack of each segment makes up the input of the operational nodes of the TRT. Following the schematic of Figure 2.3, each processor sends and/or receives the element of the semigroup of previous segments and performs the semigroup multiplication;
- **semigroup multiplication:** the processors involved in this step receive the stack of previous segments in an *inputStack*. Since both

stacks, namely the stack computed by the processor and the stack received as input, are stored as hashtables, the multiplication, that is the cleanup procedure, simply consists in merging the two hashtables so that new pages and their time references can be added to the stack. The steps of message passing and semigroup multiplication are repeated until every processor receives in input its initial stack;

- **tree creation:** once all the cores have received their input stack, that is the LRU stack at the first index of their segment, the related hashtable is transformed in an interval tree. This allows the processor the perform the procedure explained in Section 1.4.2;
- **distances computation:** each processor can now independently calculate the stack distances on its sequence of pages and store such distances in its own hashtable;
- **gathering of the results:** once each core has collected all the distances of its segment in a hashtable, it sends these results to the master processor (the core with $id = num\_procs$). The master core collects the results of every segment and creates the final hastable which contains the stack distances of the whole trace.

Chapter 4

Experimental Results

We tested our code using sequences of randomly generated integers as our input, adjusting both the sequence length L and the number of different 'pages' V to create different address traces. While this methodology allows us to easily highlight several aspects of our code, such as the dependency between V and the time needed to process each instruction, our inputs do not correctly exhibits the *principle of locality* of actual address traces. This could lead to slightly degraded performances, since the elements of the semigroup processed in the first steps of the TRT could be a bit more complex than those gathered in a true address trace.

Our program was run on an IBM Power 770 with a 3.1GHz clock and 32kB of L1 cache. We repeated each experiment five times (thus $N = 5$), and in the figures below we reported the mean value and standard deviation for each set of tests. The arithmetic mean is defined as

$$\overline{x} = \frac{1}{N}\sum_{i=1}^{N} x_i \tag{4.1}$$

while standard deviations are estimated using formula

$$std\_dev = \sqrt{\frac{\sum_{i=1}^{N}(x_i - \overline{x})^2}{N-1}} \tag{4.2}$$

where $\{x_1, x_2, ..., x_N\}$ are the observed values and $\overline{x}$ is the mean value of these observations.

Our source code is written in C++ and, since we are employing MPI, compiled with the IBM mpCC compiler, using optimization flag -O2. We also made use of IBM HPC Toolkit to gather precise data on the number of instructions completed by each iteration of the program, but run times were evaluated utilizing MPI internal timers.

4.1 Summary of results

As already mentioned, our input consists of sequences of L integers taking random values between 0 and $V - 1$. In Table 4.1 we listed the run times of our algorithm on inputs with V set to 2^{17} and L ranging from 2^{21} to 2^{29}; between parentheses we reported the standard deviations evaluated with Equation 4.2. We tested our code with a number of threads varying from $P = 1$ to $P = 32$ and we ran each experiment five times.

Number of threads P	Address trace length L				
	2^{21}	2^{23}	2^{25}	2^{27}	2^{29}
1	3.99 (0.02)	18.82 (0.12)	73.21 (0.48)	348.31 (2.05)	1408.66 (9.39)
2	2.52 (0.07)	9.67 (0.03)	41.07 (1.36)	169.56 (6.86)	832.04 (47.27)
4	1.52 (0.09)	7.48 (0.72)	21.75 (0.25)	89.15 (4.51)	466.07 (3.45)
8	1.21 (0.11)	5.19 (0.21)	17.14 (1.76)	71.49 (9.43)	281.31 (36.46)
16	1.33 (0.14)	3.71 (0.56)	13.49 (2.02)	54.49 (8.15)	220.95 (29.24)
32	1.58 (0.13)	4.01 (0.54)	9.35 (0.87)	36.45 (2.46)	151.23 (30.02)

Table 4.1: Run times with $V = 2^{17}$

Performances are also depicted in Figure 4.1 on a logarithmic scale:

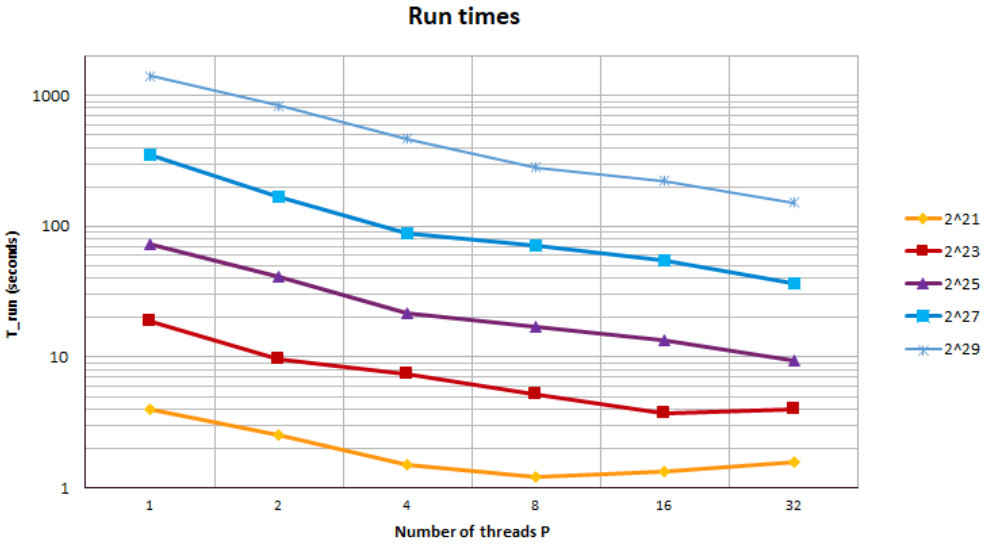


Figure 4.1: Run times with $V = 2^{17}$

Let us now examine these numbers: firstly, for $L = 2^{21}$ and $V = 2^{17}$, when employing 16 threads or more we obtain worse run times than having $P \leq 8$. This is to be expected, since we are splitting the address trace in segments

of length L/P, that is $2^{21}/2^4 = 2^{17}$ in this case. Remember that 2^{17} is also the number V of distinct pages in the trace: this means that the elements of the semigroup relative to each segment will be $O(V)$, that is around as big as the segments themselves. When fed to the TRT, such elements will take a considerable amount of time to be elaborated, thus negating the advantage of employing more processors.

In Figure 4.2 are shown the trends of the speedup (Equation 1.7) on the same address traces:

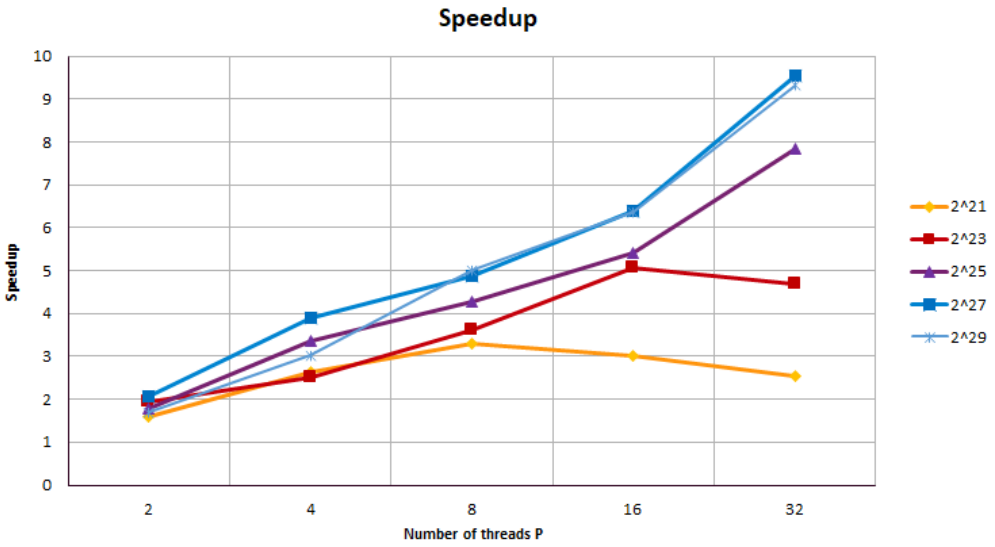


Figure 4.2: Speedup with $V = 2^{17}$

Before analyzing these results, let us summarize some ideas of parallel computing. The total work W to be done to analyze the address trace is $O(L \log V)$; if we have up to $P = L/V$ threads at our disposal, the work done by each one, which is equivalent to the run time of the algorithm, would be $W/P = O(V \log V)$. If we could achieve this result, we would have a linear speedup, that is a run time improvement proportional to the number of CPUs we are employing.

However, let us take a closer look at the operations performed by each thread:

- with $P = L/V$, every thread can extract the element of the semigroup of its segment in time

$$T_{sequential} = O(V \log V)$$

To this initial operation that each thread carries out independently, we have to add another factor $O(V \log V)$ needed to calculate the stack distances on its own segment;

- these elements are then sent to the Twisted Reflected Tree, which performs semigroup multiplication in time $O(V \log V)$. By construction, the TRT has a number of levels proportional to the logarithm of the number of threads we are employing; thus, communications between threads, which make up the parallel section of our program, takes up to

$$T_{communication} = O(V \log V) \log 2(P)$$

It is clear that $T_{communication}$ grows as we use more threads. Hence, we can no longer consider $P = L/V$ to be the optimal bound: we can achieve a linear speedup only if we employ a number of threads smaller than

$$P' = \frac{L/V}{\log L/V}$$

This means that we are increasing the sequential workload of each thread, ensuring that more time is spent computing the stack distances rather than the elements of the semigroup. Therefore, at the cost of a smaller degree of parallelism, we are better exploiting our computational power. Please note that in this equation, when $L = V$, that is when there are no two same pages in the sequence, the number of threads needed to achieve a linear speedup is $P' = 1$, namely it is useless to parallelize this problem.

To further prove our point regarding the correlation between run times and the size of V comparatively to L, we ran several experiments varying the number of different pages in the sequence, while keeping the same address trace length. In Table 4.2 are shown the run times when L is set to 2^{23} and V ranges from 2^{13} to 2^{21}:

Number of threads P	Number of distinct pages V				
	2^{13}	2^{15}	2^{17}	2^{19}	2^{21}
1	6.15 (0.01)	11.89 (0.13)	18.82 (0.12)	29.27 (0.01)	35.15 (0.05)
2	3.17 (0.04)	6.42 (0.05)	9.67 (0.03)	18.22 (0.03)	27.49 (0.08)
4	1.57 (0.02)	3.77 (0.26)	7.48 (0.72)	12.39 (0.99)	17.85 (0.21)
8	1.72 (0.31)	3.22 (0.80)	5.19 (0.21)	9.54 (0.43)	14.52 (0.33)
16	1.79 (0.07)	2.85 (0.62)	3.71 (0.56)	7.67 (0.47)	16.13 (0.64)
32	1.64 (0.11)	2.79 (0.41)	4.01 (0.54)	7.99 (0.47)	20.82 (0.63)

Table 4.2: Run times with $L = 2^{23}$

As to be expected, analyzing the trace with $L = 2^{23}$ and $V = 2^{21}$ we obtain another confirmation of our previous reasoning: once we exceed the

threshold $P = L/V = 2^2$, performances are not guaranteed to improve when we add more threads. Run times are graphically reported as well in Figure 4.3.

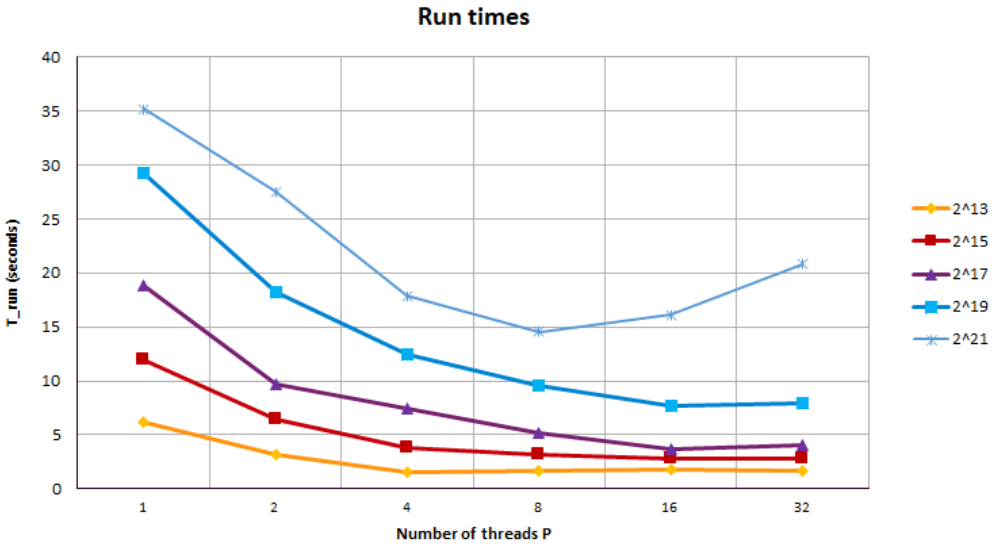


Figure 4.3: Run times with $L = 2^{23}$

To conclude this analysis of the $L = 2^{23}$, $V = 2^{21}$ trace, we note that the best run time is obtained when $P = 8$, which is a power of 2 higher than our theoretical limit of L/V. This could be caused by several factors we are going to discuss at the end of the section.

Using HPC Toolkit and the same set of traces, we collected the number of instructions completed by a single thread when calculating the stack distances. We then divided the average run times by the number of instructions, thus obtaining the average time needed to carry out a single instruction. These results are presented in Table 4.3.

Distinct pages V	Completed instructions	Time per instruction (avg)
2^{13}	$13.51*10^9$	0.46ns
2^{15}	$14.46*10^9$	0.82ns
2^{17}	$14.22*10^9$	1.32ns
2^{19}	$14.86*10^9$	1.97ns
2^{21}	$16.45*10^9$	2.14ns

Table 4.3: Performances on one thread with $L = 2^{23}$

These numbers exhibit the existence of a relationship between the total

number of completed instructions and the number of distinct pages V. This is not surprising, since V directly influences the size of the red-black tree, that is how many nodes we pass through to find a given interval, and by consequence the number of instructions needed to traverse it. The average time to complete one instruction increases as well, since the interval tree will occupy more space in the memory, thus involving lower levels of the storage hierarchy and inevitably increasing access times.

Let us now consider the clock of our machine: the IBM Power 770 has a clock of 3.1GHz, that is it can complete an instruction in around $0.32ns$. The trace with the smallest V we considered achieves the completion of an instruction in $0.46ns$. This time is very close to the actual speed of the machine; it is then worth nothing that with $V = 2^{13}$, a tree of 32 bit integers takes up around 32kB of memory, which is the size of the L1 cache of each processor of our computer. Thus, when the red-black tree can be almost entirely stored in the first level of cache, the time needed to recover data from the lower levels of the memory hierarchy is significantly reduced.

We now address the previous topic regarding the run times with $P = 8$ on the address trace with $L = 2^{23}$ and $V = 2^{21}$. The machine we used has three racks with two processors each, for a total of six processors. Every processor has eight cores that can have up to four threads running. Hence there is a trade-off to consider: if we have more processes, that is threads, on the same core, we will have smaller communication times at the cost of higher $T_{sequential}$, since there is concurrency between the threads. When, instead, the processes run on different cores, we will have no concurrency but $T_{parallel}$ will increase. Since we do not have full control over the scheduling of the jobs, this factor could explain the run times we obtained above.

To summarize, to improve performances of a parallel algorithm we could consider two different routes:

- **reduce the number of operations:** an approach independent from parallelism which decreases the workload on the computer;
- **boost parallelism:** when the number of operations has been reduced to the smallest possible amount, we can devise a strategy that takes advantage of more processors.

The first strategy is intuitively quite hard to pursue, since there are lower bounds to the number of operations that have to be performed. When we consider the second approach instead, we have to remember that employing more than a certain amount of processors will result in a waste of computational power, since we are doing more work than what is necessary. We have

already discussed that in our case this threshold is $P = L/V$, since with this number of processors we obtain segments of length $O(V)$, thus negating the advantages of parallelism. With segments shorter than V we don't obtain a reasonable speedup, since it is also important to remember that the TRT does not use all the processors at all times.

4.2 Open problems and future work

As shown in Section 4.1, some of our tests exhibit a non negligible standard deviation. In order to obtain more accurate data on the run times of our algorithm, all the experiments could be repeated increasing N from 5 to 20, thus hoping to halve the estimated standard deviation. Once we have more precise results at our disposal, an analysis that compares the performances of our code with the theoretical bound could be performed. In particular, the upper bound of $T = O(L \log V)$ is not particularly useful, since it does not provide additional informations regarding temporal constants C_1 and C_2 in the form of $T = C_1(L \log V) + C_2 L$. Therefore, a more in depth analysis should aim to estimate these two constants and provide a Θ temporal bound.

An interesting feature to add would be a method that outputs the stack distance found at each time t, $0 \leq t \leq L-1$. As of now, our code collects the stack distances of the entire address trace and lists them once the analysis of the whole trace has been completed, while to perform an on-line analysis of the trace, all time indexes t should be associated with the distance found at that moment.

However, the true purpose of the FSM-prefix computation paradigm we presented is to serve as a framework for the employment of different replacement policies: once the TRT scheme is established, it is sufficient to devise a suitable semigroup and its related multiplication to be able to simulate other replacement algorithms, such as LFU, MRU or OPT. OPT in particular could be the natural continuation of our work, since this policy naturally employs LRU to compute the stack distances of an address trace.

4.3 Conclusions

In this thesis we have tackled the stack distance computation problem. We have employed a strategy involving Finite State Machines and prefix computation; after having devised a suitable semigroup for the LRU policy, we have specified how to represent the elements of said semigroup, as well as the operation to be performed to multiplicate two different elements, that is the cleanup of a sequence. Using the Twisted Reflected Tree we have pinpointed which multiplications are needed to calculate the prefix of every segment, thus allowing several processors to independently compute the stack distance of different segments at the same time. We have then implemented this strategy using a suitable data structure, that is the red-black tree, and we have conducted an analysis on the run times of our algorithm, thus confirming the advantages of using parallelism to deal with this problem. While we have focused on the LRU replacement policy, the theoretical framework we have provided could be adopted to deal with different replacement algorithms.

Bibliography

[1] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation techniques for storage hierarchies," *IBM Systems Journal*, vol. 9, no. 2, pp. 78–117, 1970.

[2] G. Bilardi, K. Ekanadham, and P. Pattnaik, "On approximating the ideal random access machine by physical machines," *J. ACM*, vol. 56, pp. 27:1–27:57, Aug. 2009.

[3] G. Bilardi, K. Ekanadham, and P. Pattnaik, "Optimal on-line computation of stack distances for min and opt," in *Proceedings of the Computing Frontiers Conference*, CF'17, (New York, NY, USA), pp. 237–246, ACM, 2017.

[4] G. Almási, C. Caşcaval, and D. A. Padua, "Calculating stack distances efficiently," *SIGPLAN Not.*, vol. 38, pp. 37–43, June 2002.

[5] B. T. Bennett and V. J. Kruskal, "Lru stack processing," *IBM Journal of Research and Development*, vol. 19, pp. 353–357, July 1975.

[6] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd ed., 2001.

[7] Q. Niu, J. Dinan, Q. Lu, and P. Sadayappan, "Parda: A fast parallel reuse distance analysis algorithm," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pp. 1284–1294, May 2012.

[8] G. Bilardi and F. P. Preparata, "Digital filtering in VLSI," in *VLSI Algorithms and Architectures, Aegean Workshop on Computing, Loutraki, Greece, July 8-11, 1986, Proceedings*, pp. 1–11, 1986.