Calculating Stack Distances Efficiently

George Almasi, Calin Cascaval and David A. Padua
Department of Computer Science
University of Illinois at Urbana-Champaign
{galmasi, cascaval, padua}@cs.uiuc.edu

October 20, 1999

Abstract

This paper describes our experience using the stack processing algorithm [8] for estimating the number of cache misses in scientific programs. By using a new data structure, and various optimization techniques, we obtain instrumented run-times within 50 to 100 times the original optimized run-times of our benchmarks.

1 Introduction

Caches have become one of the most important components in computer systems. As processor speed increases much faster than memory speed, the memory subsystem becomes the bottleneck. To alleviate this behavior, computer architects have designed deeper memory hierarchies, including several levels of cache. In fact, improved technology provides many transistors on chips, much more than are needed for the processor's logic. In the recent past most of these transistors have been used for caches, increasing both the size and the number of levels of on-chip caches (e.g. the Alpha 21364 has 100 mil. transistors, of which 92 mil. are used for caches).

The stack algorithm [8] was originally designed for modeling virtual paging, i.e. to operate on a program trace consisting of virtual page references, but in the recent past has been used mainly to model cache behavior, by tracing cache line references [9, 10, 6, 12].

Different architectures exhibit different memory systems with caches organized in different configurations. In order to make the best use of the memory hierarchy, either the programmer or the compiler must be aware of the cache organization of the machine. We believe that the programmer should not be burdened with this responsibility, therefore we have developed a memory hierarchy model that helps the compiler to predict the number of cache misses, and thus reorganize the code to optimize the cache behavior [4]. This memory model is based on the stack distances obtained from a stack processing algorithm [8].

The main advantage of the stack algorithm in simulating cache behavior is that it allows the estimation of the number of misses for caches of any size in a single pass through the trace. Variants of the algorithm have been used to simulate caches of multiple line sizes.

Unfortunately, since most programs reference about one hundred times more cache lines than virtual pages, the stack algorithm in its original form becomes very time-consuming. Several researchers [1, 6, 9] proposed changes to the original algorithm to improve its speed.

Seeking to further improve the performance of the stack algorithm, we introduce two new data structures and corresponding algorithms, each of which is more suitable for a particular kind of application. The interval tree approach works well for programs with long traces but relatively good locality, whereas the preallocated tree approach is more suited to shorter traces with bad locality.

The rest of this paper is organized as follows. First we give a short description of the LRU stack algo-
rithm, and briefly describe the work done by others. In section 4 we describe the new data structures and algorithms we used. Section 5 details the experiments we ran using the Perfect Benchmarks [2] benchmark suite. We conclude by qualifying our results and identifying possible areas of future research.

2 Stack Distances

Simulation of memory hierarchies dates back to 1970, when Mattson et al. [8] presented an evaluation of virtual memory page replacement strategies with a stack.

The stack processing algorithm takes a trace of memory references, cache line references or virtual page references in a program, and builds a stack as follows: if a memory location has been previously referenced, we record the distance from the top of the stack at which the reference is found, and move the reference on the top of the stack if the reference is the first access to that memory location, we record as the distance and push the reference on the stack.

The result of the stack processing algorithm is a histogram that counts the number of accesses for all stack distances. Figure 1 shows a histogram computed in this way.

This histogram can be used to calculate the number of out-of-core page references, or equivalently, the number of cache misses, for any memory or cache size. For physical memory size $C$, all the accesses and stack distances of less than $C$ are in-core, and all the others are out-of-core. Splitting the stack depth histogram at $C$, the area under the histogram curve at the left of $C$ is the number of in-core references, whereas the area to the right of $C$ represents the out-of-core accesses.

The stack algorithm is based on the inclusion property, which states that a cache of size $C+1$ will always include a cache of size $C$ for any replacement strategy.

The LRU replacement policy ensures that if a cache line $L$ is accessed several times and the number of distinct references to other cache lines between two consecutive accesses to $L$ is less than the total number of cache lines in the cache, the subsequent accesses to $L$ are hits.

![Figure 1: Stack Depth Histogram for QCD](image)

The stack processing algorithm can therefore be used exactly compute cache misses in fully associative caches, and to approximate them in set associative caches.

3 Related Work

After Mattson researchers extended the usefulness of the stack algorithm. Traiger and Slutz [11] described a one-pass algorithm to simulate multiple page sizes in virtual memory. Their technique can be used to simulate multiple cache line sizes in one pass. Thompson and Smith [10] extended the stack algorithm to simulate write-back caches, which do not satisfy the inclusion property. By adding a dirty bit to each block in the stack they were able to count the number of writes that can be avoided when the dirty block is still resident in the cache.

The stack algorithm is however very expensive to run, especially if the stack becomes large enough. It was soon recognized that more efficient data structures were needed to do the job of the stack search. Bennett and Kruskal [1] presented an algorithm which replaces the stack with a preallocated hierar-

4 Implementations of the LRU Stack Algorithm

Figure 2 gives a formal three-step description of the LRU stack algorithm, as first described by Mattson in [8]. We use this description as the basis for the algorithms we present.

It is assumed henceforth that the algorithm is operating on a memory trace of length \(N\) that contains \(M\) distinct memory references (obviously \(M \leq N\)). For the notations used in this paper refer to Section 7.

Repeat the following steps for each memory reference \(x_\tau, 0 \leq \tau < N\):

- **search**: find the location in the stack of the most recent reference to the current location.
- **count**: compute \(dist(\tau)\), the stack distance for the current location, by finding the last reference to the current location and counting the number of elements on the stack above it. If the most recent reference is not found, \(dist(\tau)\) is defined as \(\infty\).
- **update**: bring the most recent reference to the top of the stack.

Figure 2: Stack Algorithm

4.1 Naive Implementation

This implementation directly follows the algorithm presented above. The stack is represented as a doubly linked list. For each reference in the trace, the first two operations (search and count) are executed simultaneously by traversing the stack top to bottom. If the element exists in the stack, its distance from the top of the stack is recorded. Finally the element is moved from its current position to the top of the stack— the update stage. If the element is not found, \(\infty\) is recorded as its stack distance and the element is pushed on top of the stack.

**Analysis** For each reference in the trace the work done is, in the worst case, \(M\) (due to the traversal of the linked list). The total complexity is thus \(O(NM)\).

The worst case doesn’t happen very often. In fact, many programs exhibit excellent locality, causing many references to lie close to the top of the stack. Unfortunately the few references that are near the bottom of the stack cause huge slowdowns, resulting in overall bad performance.

4.2 Markers Algorithm

The major cause of slowness in the naive algorithm is the linear traversal of the linked list that makes up the stack. The markers algorithm attempts to replace linear search wherever possible.

The search phase of the markers algorithm is done using a hash table that associates a cache line/memory/page reference with its current place in the linked list. Given enough hash buckets, hashtable access and update are \(O(1)\) operations. The number of necessary hash buckets can be approximated with \(M\), the number of distinct references in the trace.

Unfortunately finding an element in the middle of the stack by using the hashtable is not enough. The stack depth of the element needs to be counted. To avoid traversing the stack from top to bottom, a set of markers are interspersed in the linear list implementing the stack, one about every \(D\) elements. The markers form another doubly linked list, and each marker records its distance from the top. To find out the depth of a memory reference in the stack, one needs to find the nearest marker by traversing the stack (a marker would be at most \(D\) steps away) and then look up the marker’s distance from the top.

When an element is removed from the stack and inserted at the top, the markers between the top and the element need to be updated. This involves at most \(M/D\) steps.

**Analysis** The cost per memory reference of this algorithm is at most \(D + M/D\) (the cost of finding
a marker, plus the cost of updating all markers up to the beginning of the stack). $D$ can be varied at runtime by adding or removing markers, in order to minimize the cost; assuming $D = \lceil \sqrt{M} \rceil$, the cost evaluates to $O(\sqrt{M})$ per element, or $O(N \ast \sqrt{M})$ total.

### 4.3 Alternative Data Structures

The major stumbling block in implementing more efficient versions of the LRU stack algorithm is the implementation of the stack as a linear list. We will present a formulation of the LRU stack algorithm that does not use a stack. We will closely follow Bennett and Kruskal’s [1] notation.

**Definition 1.** We formalize the concept of the hashtable $P$, which we already used informally in the markers algorithm. Let us define $\mathcal{J}$ as the set of indices of references to $z$ that occurred before an index $\tau$ in the trace:

$$\mathcal{J}_\tau = \{ i \mid 0 \leq i < \tau \land x_i = z \}$$

Using $\mathcal{J}$ we define the hashtable $P_\tau$ as follows:

$$P_\tau(z) = \begin{cases} 
\max\{i \mid i \in \mathcal{J}\} & \text{if } \mathcal{J} \neq \emptyset \\
\text{undefined} & \text{otherwise}
\end{cases}$$

(1)

$P_\tau(z)$ is undefined when a cold miss occurs, i.e., when there is no previous reference to $z$.

**Definition 2.** Next we define $B$, a mapping from the trace indices $0 \ldots N - 1$ to $\{0, 1\}$. Like $P$, $B$ changes with time and therefore is subscripted with $\tau$. $B_\tau(i)$ is defined as follows:

$$B_\tau(i) = \begin{cases} 
1 & \text{if } P_\tau(x_i) = i \\
0 & \text{otherwise}
\end{cases}$$

(2)

$B_\tau(i)$ is 1 if at time $\tau$ there is no reference to $x_i$ in the program trace at any index larger than $\tau$.

**Definition 3.** Given $P$ and $B$ we can define $dist(\tau)$, the stack distance of the element $x_\tau$ in the program trace: it is the number of 1’s in $B$ between the last reference to $x_\tau$ and $\tau$.

$$dist(\tau) = \begin{cases} 
|\mathcal{H}| & \text{if } P_\tau(x_\tau) \text{ is defined} \\
\infty & \text{otherwise}
\end{cases}$$

(3)

where $\mathcal{H}$ is the set of trace indices after $P_\tau(x_i)$ whose $B$ values are 1:

$$\mathcal{H} = \{ i \mid P_\tau(x_\tau) \leq i < \tau \land B_\tau(i) = 1 \}$$

We can now reformulate the stack algorithm by using $P$ and $B$ instead of the stack.

**Figure 3:** Modified Stack Algorithm

**4.4 Bennett and Kruskal’s Algorithm**

We present Bennett and Kruskal’s algorithm [1] first because it introduces ideas we need later.

The algorithm represents $P$ and $B$ explicitly. The first step of the algorithm, evaluating $P(x_\tau)$, is a hash table lookup.

The **count** step of the algorithm counts the number of true values in $B$ between the indices $P(x_\tau)$ and $\tau$. To make the counting step efficient, Bennett and Kruskal use a hierarchy of partial sums $B^1, B^2 \ldots B^L$, where $L = \lceil \log(N) \rceil$. Renaming $B$ to $B^0$, the partial sum hierarchy is set up such that for some chosen
interval \( m \), at any time \( \tau \),
\[
B^\tau(j) = \sum_{i=j}^{i=j+1-m-1} B^\tau_{m-1}(i)
\]

This formula describes an \( m \)-ary tree of nodes having the value of each node being equal to the sum of the values of its children.

Calculating the number of 1's between the indices \( P(x_\tau) \) and \( \tau \) is now a matter of traversing the partial sum hierarchy, as shown in Figure 4. The figure presents the first 31 elements of a trace. We trace the 31st access, and \( x_{31} \) last occurred in position 4.

![Figure 4: A Partial Sum Hierarchy](image)

The third step, updating \( B \), also becomes a matter of tree traversal, since all partial sums on the path from the root of the hierarchy to the leaf node are affected.

The algorithm needs two traversals of the tree. The first traversal, from the root to index \( P(x_\tau) \), deletes \( P(x_\tau) \) as the last reference to \( x \) by setting \( B(P(x_\tau)) \) value to 0 and adjusting all partial sums along the path. The second traversal is from the root to index \( \tau \) and sets \( B(\tau) \) to 1, again adjusting partial sums on the path. For reasons of brevity we are not going to fully explain the algorithm, except to mention that our major improvement, to be presented in the next sections, replaces the two traversals with a single traversal of the tree.

**Analysis** Since the tree traversal is an \( O(\log(N)) \) operation and the location finder works in constant time (hashable lookup is \( O(1) \)), the total execution time is \( O(N\log(N)) \). The memory requirements of the algorithms are very large because \( B \) and its partial sums are represented explicitly in memory.

### 4.5 Hole-based Algorithms

We define a **hole** as a memory reference in the program trace that is not the latest reference to a particular location at time \( \tau \). Holes are thus elements in \( B_\tau \) that have been set to 0.

Whereas values of 1 in \( B \) (and corresponding latest references) are newly created and then destroyed all the time, holes have the property of being created and never destroyed.

Using the concept of holes, the stack distance at index \( \tau \) can be expressed as

\[
dist(\tau) = \tau - P(x_\tau) - holes_\tau(P(x_\tau))
\]

where \( holes_\tau(i) \) is the number of holes in the program trace between indices \( i \) and \( \tau \). Here we are in effect counting the 0's in \( B \) instead of counting the 1's, and adjusting Equation (3) to reflect this.

Holes can be represented more efficiently than latest references. We will present two kinds of algorithms based on holes, a variant in which holes are held by an interval tree and another which is a faster version of Bennett and Kruskal's algorithm.

#### 4.5.1 Interval Tree of Holes

An interval tree is used to efficiently represent an ordered set of mutually disjoint intervals \( I = \{[i_{11}, i_{12}], [i_{21}, i_{22}], ..., [i_{n1}, i_{n2}]\} \). In our case the intervals in \( I \) are all bounded by natural numbers (indices in the program trace). The intervals represent contiguous sets of indices that are holes in the trace.

Interval trees (Figure 5) are represented as a quasi-balanced binary trees \( BT \) (such as red-black trees [5] or AVL trees [7]) in which each node \( n \) represents the closed interval \([k_1(n), k_2(n)]\). The tree ordering corresponds to the order of the intervals in \( I \); thus \( k_1(n) > k_2(left(n)) \) and \( k_2(n) < k_3(right(n)) \), where \( left(n) \) and \( right(n) \) are respectively the left and right children of \( n \).
4.5.2 The Partial Sum Hierarchy

We use the interval tree to evaluate the number of holes between $P(x_\tau)$ and $\tau$. There are no holes beyond the current index $\tau$ (a logical impossibility considering the definition of a hole). Thus we are left with counting the number of holes at indices larger than $P(x_\tau)$. To do this, we follow Bennett and Kruskal’s method and associate a value $sum(n)$ with each interval node $n$, to hold the sum of holes contained in the children of $n$. Our hole tree now becomes equivalent to Bennett and Kruskal’s partial sum hierarchy.

A slight optimization is to make $sum(n)$ hold the sum of holes in the right subtree of $n$ instead of $n$ itself. In Figure 6 the shaded boxes contain partial sums of the right subtree of their nodes, as indicated by the dashed arrows. This optimization reduces the number of right subtree dereferenciations when the next target of the tree traversal is the left subtree (in Figure 6 the nodes marked with (*) will not need to be dereferenced).

The counting algorithm works like this: we traverse the interval tree from the root towards the leaf node closest to index $i = P(x_\tau)$. We carry a partial sum along the path, and add to it the sum of holes in all subtrees encountered to the right of the path (i.e. having indices larger than $i$).

4.5.3 Updating the Interval Tree

We now extend the counting algorithm to include the third component of the LRU stack algorithm: update. We need to update the tree structure as well as the partial sum hierarchy residing in it.

With regard to inserting a new hole $p$ into the interval tree, there are several cases to consider:

- $p$ may be adjacent to a single existing interval $[k_1, k_2]$ in the tree, i.e. $p = k_1 + 1$ or $p = k_2 + 1$. In this case the interval is adjusted to include $p$.

- $p$ may be adjacent to two intervals $[k_1, p - 1]$ and $[p + 1, k_3]$. In this case the intervals are fused into a single interval $[k_1, k_3]$ prompting the deletion of one of the nodes and the potential rebalancing of the whole tree.

- $p$ may not be adjacent to any intervals in $BT$. In this case a new node is created, to hold the interval $[p, p]$. Again, the tree may need to be rebalanced.

The partial sum hierarchy is updated by changing the $sum$ values of the nodes on the path from the root to the affected interval. Figure 6 illustrates the operation of counting holes and inserting a new hole at location 101 in an example tree. Appendix A lists the algorithm that performs this operation.
Analysis The algorithm presented in Figure 6 is based on a quasi-balanced binary tree. dist\(p, n\) is a variant of the insertion operation for quasi-balanced
binary trees, which makes it an \(O(log(M))\) operation (the number of disjunct hole intervals, and thus nodes
in the tree, is always less than \(M+1\)). Thus the total execution time of the Binary Tree Hole Algorithm
is \(O(Nlog(M))\).

4.6 Preallocated Tree of Holes

A tree of holes can also be implemented as a preallocated fixed tree \(\{B_0, B_1, \ldots, B_{log(N)}\}\), like the one of Bennett and Kruskal. Unfortunately the memory
requirements for the whole tree get quickly out of hand: for a program trace of length \(2^M\) (a realistic number for today’s programs) we need to allocate
\(1 + 2 + 2^2 + \ldots + 2^{30} = 2^{31} - 1\) locations.

The silver lining is that not all locations need to be of the same size. Elements of \(B_0\), for instance, need
to hold only one bit; elements of \(B_1\) need to be two
bits each, and so on; the total memory requirement is
\(\frac{1}{8} \times (2^{30} + 2 \times 2^{29} + 3 \times 2^{28} + \ldots + 31 \times 2^0) \approx 536.87MB\),
which fits into the virtual memory of most modern
workstations. Also, the algorithm does not use all
of this memory at once, but rather progresses slowly
through it as the trace is analyzed. This allows for
huge portions of the preallocated tree to reside in
virtual storage.

Analysis Since the tree is preallocated, and has \(N\)
leaf nodes, tree traversal is now an \(O(log(N))\) operation
rather than \(O(log(M))\), which would seem to
make this algorithm impractical. Also, the tree needs
to be allocated before the program is run, which
means that the user has to guess \(N\).

However, once \(N\) is calibrated, the algorithm be-
comes the fastest we tried so far, outperforming Bennett
and Kruskal’s by a factor of up to 2:1. The rea-
son for this is that only one tree traversal is needed
per element, as opposed to two for Bennett and
Kruskal’s algorithm.

5 Experimental Results

We selected the Perfect Benchmarks [2] as our ex-
perimental base and instrumented them with a custom
version of the Polaris source-to-source translator [3]
to generate a program trace. Rather than storing
the program trace we hooked up the analyzer to the
instrumented benchmark directly, and generated the
trace and the histogram on the fly.

At first we used the naive implementation of the
LRU stack algorithm, and experienced a drastic slow-
down. In an effort to find better implementations of
the LRU algorithm we experimented with all algo-

"
the stack processing part is not implemented. We measured “ml” to find out how much the benchmarks are affected by just generating the trace.

- kru and pre are preallocated implementations of Bennett and Kruskal’s, and the preallocated tree hole based algorithm, respectively.

- avl and rb are interval tree implementations using AVL and red-black trees respectively.

- mkr is the markers algorithm. Many of the numbers are missing because we had to abort runs that were taking too long.

We also measured the relative overhead of the stack computation. Figure 7 breaks down the total runtime of each benchmark into the time spent in the original benchmark, instrumentation overhead (i.e., time spent generating the program trace), hash table lookup overhead and stack computation overhead.

The interval tree based algorithms have better theoretical bounds than the preallocated tree algorithms, $O(Nlog(M))$ versus $O(Nlog(N))$. There are several reasons why the preallocated algorithms tend to yield better execution times in practice:

- The interval tree implementation severely stresses the memory bandwidth of the host processor. For each element in the program trace the interval tree algorithm generates about $3 \cdot log(M)$ additional references while traversing the interval tree: in each tree node at least one node key is accessed; in addition the node’s partial sum is accessed and one of the leaves is dereferenced.

The value of $M$ can be approximated with the measured maximum stack depth, which for most of our algorithms yields an AVL tree height of around 20 to 25, resulting in up to 75 extra memory accesses per element in the memory trace. In the case of red-black trees the number of references is even higher.

By comparison the preallocated tree implementation generates only $log(N)$ (or $2 \cdot log(N)$, in the case of Kruskal’s algorithm) references. In practice we limited $N$ to $2^{31}$, which means 31 memory references for each tree. In addition the preallocated tree is built such that adjacent nodes at lower levels tend to be clustered into the same cache line, resulting in good spatial locality.

- The interval tree implementation relies on dynamic memory allocation as the interval tree shrinks and expands in the course of the process. We were able to gain up to 33% in execution speed by writing our own memory allocators (this gain is included in the performance figures).

The better speed of the preallocated strategy comes, however, at the cost of extremely high memory usage (about 600 MBytes of virtual memory for
the preallocated tree) and a hard limit of $2^{31}$ references in the memory trace. For a few of the benchmarks this limit is exceeded.

The interval tree implementation, if slower, has no inherent limitation with respect to the trace size and delivers reasonable performance. We see it as a more useful tool on the whole. The AVL tree is preferable to the red-black tree, since the higher reordering cost is clearly amortized by the lower average tree height.

In conclusion, the preallocated implementation works better for programs with short traces, bad locality and large cores (that is, large $M$ and relatively small $N$ values), whereas the interval tree implementation works better on long traces with good locality and small cores (larger $N$, smaller $M$ values).

6 Conclusions, Future Work

In this paper we have presented our experience using stack algorithms [8] to simulate the cache behavior of scientific programs. We have implemented several versions of an algorithm that was devised to simulate virtual paging, and although accurate for caches, is not performing very well due to the large number of memory references.

The main contribution of this paper is a novel way of computing stack distances based on holes. This allows the algorithm to perform fast enough to be convenient to use.

In its present form, the algorithm is easily modifiable for simulating multiple cache line sizes, as described in [11]. However we have simulated only fully associative caches. Finding a way to extract information on arbitrary associativity [6] using a single stack, or interval tree, is a matter of future research. Further improvements in simulation speed, and reductions in memory footprint, would also be welcome.

7 Notations

This section enumerates and explains some of the symbols we used throughout this paper.

- $N$: number of references in the program trace under consideration.

- $M$: number of distinct memory references in the program trace. In effect $M$ is equal to the size of the memory used by the program we are analyzing.

- $\tau$: used as the current index in the trace. As such, $0 \leq \tau < N$. The stack algorithm processes the program trace sequentially; $\tau$ always denotes the current position processed by the algorithm.

- $x_\tau$: denotes the memory reference at index $\tau$ in the program trace.

- $P$: a mapping from memory references to trace indices. Since $P$ changes in time, it is normally indexed with $\tau$, the current index in the program trace.

- $B$: a mapping from trace indices to booleans. $B_{\tau}(i)$ is set if at moment $\tau$ the location referenced at position $i$ in the trace is the latest reference to its location.

- $\text{dist}(\tau)$: the stack distance corresponding to the location referenced in position $\tau$ in the trace. This is the number we compute for each element in the trace.

- $\text{holes}_\tau(i)$: the number of holes in the program trace between indices $i$ and $\tau$ at moment $\tau$, $i < \tau$ by definition.

References


A The Interval Tree Algorithm

function dist (n, p)
begin
if (p < key2(n) − 1)
    /* continue search left */
    if (left(n) = nil)
        return sum(n)+dist(left(n))
    else
        /* can’t continue left - no nodes left */
        left(n) := new interval (p, p)
        sum(left(n)) := 0
        return sum(n)
end if
else if (p > key2(n) + 1)
    /* continue search right */
    if (right(n) = nil)
        sum(n) := sum(n)+1
        return dist(right(n))
    else
        /* can’t continue right - no nodes left */
        right(n) := new interval (p, p)
        sum(right(n)) := 0
        return 0
end if
else if (p = key2(n) − 1 AND
    p = key2(left(n)) + 1)
    /* merge left node */
    key2(n) := key2(left(n))
    remove_node(left(n))
    rebalance(n)
    return key2(n) − p + sum(n)
else if (p = key2(n) − 1)
    /* add to node */
    key2(n) := p
    return key2(n) + p + sum(n)
else if (p = key2(n) + 1 AND
    p = key2(right(n)) − 1)
    /* merge right node */
    key2(n) := key2(right(n))
    remove_node(right(n))
    rebalance(n)
    return sum(n)
else if (p = key2(n) + 1)
    /* add to node */
    key2(n) := p
    return sum(n)
else
    /* internal error */
end if
end