Calculating Stack Distances Efficiently

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

ABSTRACT

This paper [1] describes our experience using the stack processing algorithm [6] 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.

Keywords: memory systems, measurement and monitoring, simulation, cache modeling

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.

The stack algorithm [6] 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 [7, 8, 4, 10].

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. This memory model is based on the stack distances obtained from a stack processing algorithm [6].

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, 4, 7] 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 and 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 algorithm, 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. [6] presented an evaluation of virtual memory page replacement strategies with a stack. His algorithm takes a trace of memory references – cache line references or virtual page references in a program – and builds a stack as follows: every memory location is pushed onto the stack when it is referenced. Locations that have been referenced a long time ago sink towards the bottom of the stack, but locations that are referenced again are extracted from the stack and pushed back on top.

The depth from which a certain reference had to be extracted in the stack is called the stack distance of that reference. By convention, the stack distance of first-time references is ∞.

The result of the stack processing algorithm is a trace of stack distances. These can be coalesced into a histogram of stack distances. Figure 1 shows the stack distance histogram for the QCD benchmark of the Perfect benchmark suite.

The stack distance histogram is a useful measure of locality, both visually and numerically. It 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 at 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

[1]This work was supported in part by NSF contract ACI98-70687.

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.

This is true because caches and virtual memory systems have the inclusion property: a cache of size $C+1$ will always include a cache of size C for any replacement strategy.

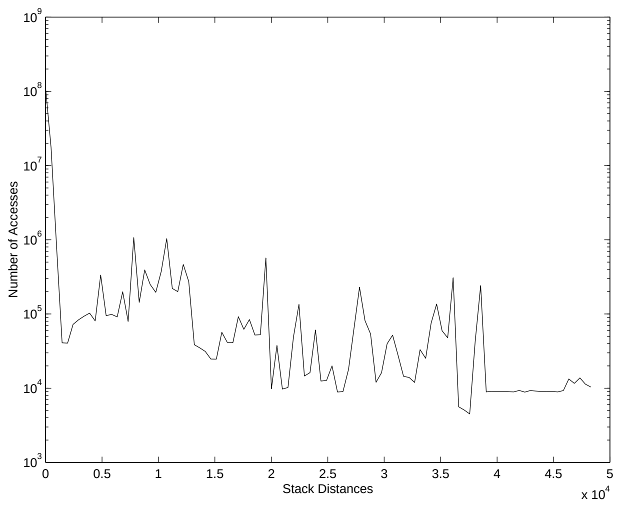

Figure 1: Stack Depth Histogram for QCD

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.

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

Following Mattson's work, other researchers extended the usefulness of the stack algorithm. Traiger and Slutz [9] 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 [8] 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 pre-allocated hierarchy of partial sums. Hill and Smith [4] used a forest of trees to simulate multiple cache associativities; Sugumar and Abraham [7] used a generalized binomial tree for the same purpose.

4. IMPLEMENTATIONS OF THE LRU STACK ALGORITHM

Figure 2 shows a formal three-step description of the LRU stack algorithm, as first described by Mattson in [6]. 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 $M \leq N$). The trace is indexed with the letter τ, where $1 \leq \tau \leq N$. For the notations used in this paper refer to Section 7.

Repeat the following steps for each memory reference x_τ, $0 \leq \tau < N$:

- **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 ∞.
- **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, ∞ is recorded as its stack distance and the element is pushed on top of the stack.

4.1.1 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 slow-downs, resulting in terrible overall performance.

4.2 Markers Algorithm

The slowest part of the naive algorithm is the linear traversal of the linked list that makes up the stack. The *markers* algorithm, presented next, replaces linear search with a combination of hasthtable lookup and two-level linked lists.

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.

4.2.1 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 $O(N * \sqrt{M})$ 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.

4.3.1 Definition 1.

To aid us in the search phase of the algorithm, we formalize the concept of the hashtable $\mathbf{P}$, which we already used informally in the markers algorithm. The hash table contains, for each memory reference in the trace, the time/index of its last use. We define $\mathcal{J}$ as the set of indices of references to z that occurred *before* an index τ in the trace:

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

Using $\mathcal{J}$ we define the hashtable $\mathbf{P}_\tau$ as follows:

$$\mathbf{P}_\tau(z) = \begin{cases} max\{i \mid i \in \mathcal{J}\} & \text{if } \mathcal{J} \neq \emptyset \\ undefined & \text{otherwise} \end{cases} \tag{1}$$

Note 1. $\mathbf{P}_\tau$ is time dependent; hence the τ subscript.

Note 2. $\mathbf{P}_\tau(z)$ is undefined when a cold miss occurs, i.e. when there is *no* previous reference to z.

4.3.2 Definition 2.

Next we define $\mathbf{B}$, a mapping from the trace indices $0 \ldots N-1$ to $\{0, 1\}$. Intuitively, $\mathbf{B}$ flags all the references that are on the stack at time τ. Like $\mathbf{P}$, $\mathbf{B}$ changes with time and therefore is subscripted with τ. $\mathbf{B}_\tau(i)$ is defined as follows:

$$\mathbf{B}_\tau(i) = \begin{cases} 1 & \text{if } \mathbf{P}_\tau(x_i) = i \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

$\mathbf{B}$ is an alternative representation of the stack: $\mathbf{B}_\tau(i) = 1$ precisely for the elements i in the trace that are part of the stack at any given moment.

4.3.3 Definition 3.

Given $\mathbf{P}$ and $\mathbf{B}$ we can define $dist(\tau)$, the stack distance of the element x_τ in the program trace: it is the number of 1's in $\mathbf{B}$ between the last reference to x_τ and τ.

$$dist(\tau) = \begin{cases} |\mathcal{H}| & \text{if } \mathbf{P}_\tau(x_\tau) \text{ is defined} \\ \infty & \text{otherwise} \end{cases} \tag{3}$$

where $\mathcal{H}$ is the subset of trace indices between $\mathbf{P}_\tau(x_i)$ and τ whose $\mathbf{B}$ values are 1:

$$\mathcal{H} = \{i \mid \mathbf{P}_\tau(x_\tau) \leq i < \tau \wedge \mathbf{B}_\tau(i) = 1\}$$

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

Repeat for each reference x_τ, $0 \leq \tau < N$:

- **search**: compute $\mathbf{P}_\tau(x_\tau)$;
- **count**: evaluate $dist(\tau)$. If $\mathbf{P}_\tau(x)$ is undefined then $dist(\tau)$ is defined as ∞;
- **update**: change $\mathbf{B}$ and $\mathbf{P}$ as follows:

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

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

Figure 3: Modified Stack Algorithm

4.4 Bennett and Kruskal's Algorithm

Given the alternative data structures we just presented, the stack algorithm boils down to maintaining $\mathbf{B}$ and $\mathbf{P}$ and computing $dist(\tau)$ from $\mathbf{B}$ and $\mathbf{P}$. All algorithms presented in the remainder of this paper attempt to compute *dist* in a more efficient way.

Bennett and Kruskal's algorithm [1] maintains a modified form of $\mathbf{B}$: it uses a hierarchy of partial sums $\mathbf{B}^1, \mathbf{B}^2 \ldots \mathbf{B}^L$, where $L = \lceil log(N) \rceil$. Renaming $\mathbf{B}$ to $\mathbf{B}^0$, the partial sum hierarchy is set up such that for some chosen interval m, at any time τ,

$$\mathbf{B}^s{}_\tau(j) = \sum_{i=j \cdot m}^{i=(j+1) \cdot m-1} \mathbf{B}_\tau^{s-1}(i)$$

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 $\mathbf{P}(x_\tau)$ and τ 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 31^{st} access, and x_31 last occurred in position 4.

The third step, updating $\mathbf{B}$, 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 $\mathbf{P}(x_\tau)$, deletes $\mathbf{P}(x_\tau)$ as the last reference to x by setting $B(\mathbf{P}(x_\tau))$ value to 0 and adjusting all partial sums along the path. The second traversal is from the root to index τ 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

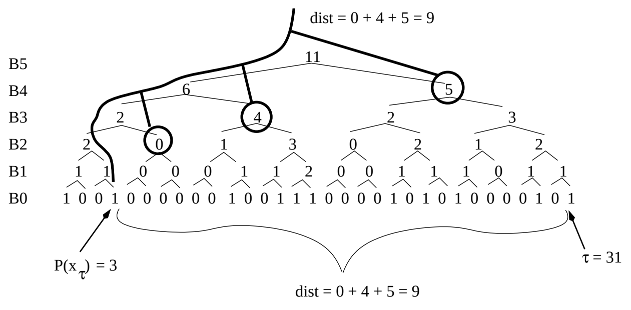

Figure 4: A Partial Sum Hierarchy

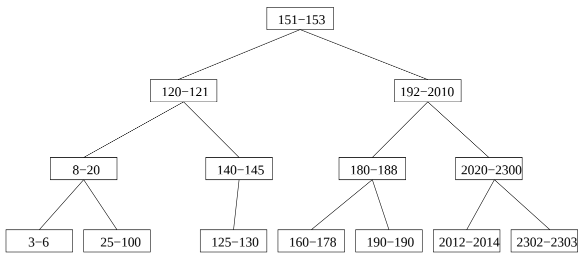

Figure 5: An Interval Tree

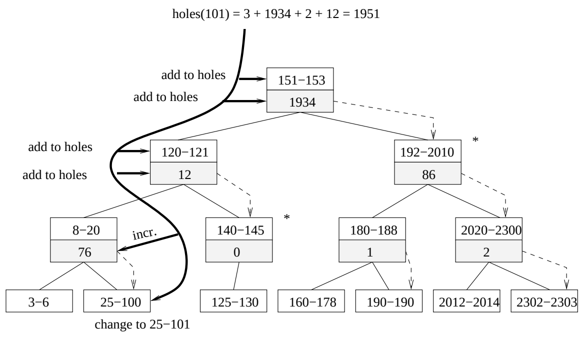

Figure 6: Updating the Tree of Holes

that our major improvement, to be presented in the next sections, replaces the two traversals with a single traversal of the tree.

4.4.1 *Analysis*

Since the tree traversal is an $O(log(N))$ operation and the location finder works in constant time (hashtable lookup is $O(1)$), the total execution time is $O(Nlog(N))$. The memory requirements of the algorithms are very large because $\mathbf{B}$ 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 τ. Holes are the zero elements in $\mathbf{B}_\tau$.

Whereas values of 1 in $\mathbf{B}$ (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 τ can be expressed as

$$dist(\tau) = \ \tau - \mathbf{P}_\tau(x_\tau) - holes_\tau(\mathbf{P}_\tau(x_\tau))$$

where $holes_\tau(i)$ is the number of holes in the program trace between indices i and τ. Here we are in effect counting the 0's in $\mathbf{B}$ 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 can be used to efficiently represent an ordered set of mutually disjunct 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 $\mathbf{BT}$ (such as red-black trees [3] or AVL trees [5]) in which each node $\mathbf{n}$ represents the closed interval $[k_1(\mathbf{n}), k_2(\mathbf{n})]$. The tree ordering corresponds to the order of the intervals in I; thus $k_1(\mathbf{n}) > k_2(left(\mathbf{n}))$ and $k_2(\mathbf{n}) < k_1(right(\mathbf{n}))$, where $left(\mathbf{n})$ and $right(\mathbf{n})$ are respectively the left and right children of $\mathbf{n}$.

4.5.2 *The Partial Sum Hierarchy*

We use the interval tree to evaluate the number of holes between $P(x_\tau)$ and τ. There are no holes beyond the current index τ (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 $sum(\mathbf{n})$ with each interval node $\mathbf{n}$, to hold the sum of holes contained in the children of $\mathbf{n}$. Our hole tree now becomes equivalent to Bennett and Kruskal's partial sum hierarchy.

A slight optimization is to make $sum(\mathbf{n})$ hold the sum of holes in the *right subtree* of $\mathbf{n}$ instead of $\mathbf{n}$ 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 dereferentiations 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 re-balancing of the whole tree.

- p may not be adjacent to any intervals in $\mathbf{BT}$. 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. The Appendix lists the algorithm that performs this operation.

4.5.4 Analysis

The algorithm presented in Figure 6 is based on a quasi-balanced binary tree. $dist(p, \mathbf{n})$ 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 $\{\mathbf{B}^0, \mathbf{B}^1, ...\mathbf{B}^{log_2(N)}\}$, 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^{31} (a realistic number for todays programs) we need to allocate $1 + 2 + 2^2 + ... + 2^{30} = 2^{31} - 1$ locations.

The silver lining is that not all locations need to be of the same size. Elements of $\mathbf{B}_0$, for instance, need to hold only one bit; elements of $\mathbf{B}_1$ need to be two bits each, and so on; the total memory requirement is $\frac{1}{8} \times (2^{30} + 2 \times 2^{29} + 3 \times 2^{28} + \ldots + 31 \times 2^0) \cong 536.87$MB, 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.

4.6.1 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 becomes the fastest we tried so far, outperforming Bennett and Kruskal's by a factor of up to 2:1. The reason 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 experimental base and instrumented them with a source-to-source translator to generate a program trace. Rather than storing

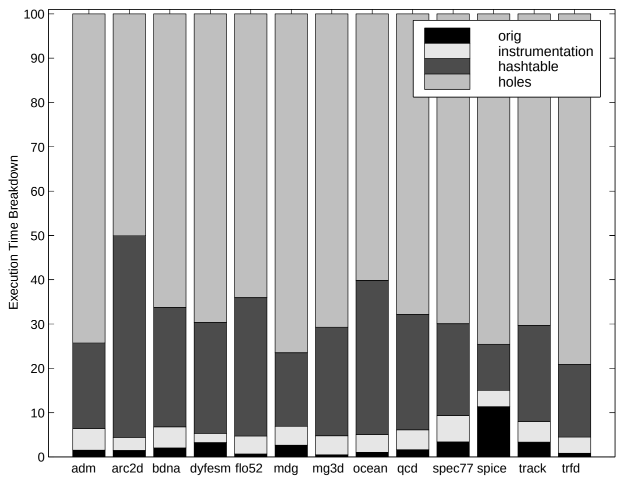

Figure 7: Execution Time Breakdown for the Preallocated Hole Tree Algorithm

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 slowdown. In an effort to find better implementations of the LRU algorithm we experimented with all algorithms described in this paper.

We ran our experiments on a 270MHz Ultrasparc Solaris machine. Table 1 summarizes the results we obtained. Benchmarks are listed by name; the total number of references and the maximum stack are included.

The algorithms we measured are the following:

- `orig` is the runtime of the original non-instrumented benchmark.

- `nul` measures the trace generation overhead, but the stack processing part is not implemented. We measured "nul" 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's, respectively.

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

- `mrk` 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:

Parameters			Overhead		Algorithms				
Bmark	N=#refs	M	orig	nul	pre	kru	avl	rb	mrk
adm	460405734	16520	11.81	180.87	703.21	1010.35	1036.01	1128.84	5652.68
arc2d	1961059165	899122	81.13	2680.86	5372.14	6382.12	12391.1	14986.1	>24h
bdna	436191438	191344	17.59	278.16	823.83	1092.76	1137.56	1492.08	61693.60
dyfesm	410077680	16075	19.35	175.87	579.36	998.65	858.14	969.78	>24h
flo52	740622750	207857	10.18	513.1	1427.88	1912.36	2994.88	3744.76	>24h
mdg	2390578661	29053	94.70	836.22	3557.01	6127.57	4576.22	5363.06	4897.80
ocean	1705808702	149990	39.08	1443.31	3625.56	4561.16	4981.03	7343.2	>40h
qcd	150442837	459275	4.69	86.59	268.91	487.7	367.82	500.64	>24h
spec77	2489578773	216430	170.96	1505.91	5008.94	7001.32	7640.15	10653.4	>24h
spice	8228372	63559	2.90	5.24	20.58	34.58	17.03	19.24	34.04s
track	40569358	72548	3.21	23.9	80.51	113.71	87.01	99.88	4570.19
trfd	666159025	675681	9.63	238.33	1139.39	1800.28	1735.31	2474.74	>40h

Table 1: Perfect Benchmark run times (seconds)

- 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 [6] 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 [9]. However we have simulated only fully associative caches. Finding a way to extract information on arbitrary associativity [4] 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. NOTATION

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.

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

- x_τ: denotes the memory reference at index τ in the program trace.

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

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

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

- $holes_\tau(i)$: the number of holes in the program trace between indices i and τ at moment τ. $i < \tau$ by definition.

8. REFERENCES

[1] B. T. Bennett and V. J. Kruskal. LRU stack processing. *IBM Journal for Research and Development*, pages 353–357, July 1975.

[2] M. Berry et al. The Perfect Club Benchmarks: Effective Performance Evaluation of Supercomputers. *International Journal of Supercomputer Applications*, 3(3):5–40, 1989.

[3] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[4] Mard D. Hill and Alan Jay Smith. Evaluating associativity in CPU caches. *IEEE Transactions on Computers*, 38(12):1612–1630, December 1989.

[5] Donald E. Knuth. *The Art of Computer Programming*, volume 3 (Sorting and Searching). Addison Wesley Longman, 2nd edition, 1998.

[6] R. L. Mattson, J. Gecsei, D.R. Slutz, and I.L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2), 1970.

[7] Rabin A. Sugumar and Santosh G. Abraham. Set-associative cache simulation using generalized binomial trees. *ACM Trans. Comp. Sys.*, 13(1), 1995.

[8] James G. Thompson and Alan Jay Smith. Efficient stack algorithms for analysis of write-back and sector memories. *ACM Trans. Comp. Sys.*, 7(1):78–117, February 1989.

[9] I. L. Traiger and D. R. Slutz. One-pass techniques for the evaluation of memory hierarchies. Technical report, IBM T. J. Watson Research Center, 1971.

[10] Wen-Hann Wang and Jean-Loup Baer. Efficient trace-driven simulation methods for cache performance analysis. *ACM Transactions on Computer Systems*, 9(3), 1991.

APPENDIX

The Interval Tree Algorithm

```
function dist (n, p)
begin
  if (p < $key_1(\mathbf{n}) - 1$)
    /* continue search left */
    if ($left(\mathbf{n}) \neq nil$)
      return $sum(\mathbf{n})$+dist($left(\mathbf{n})$)
    else
      /* can't continue left - no nodes left */
      left(n) := new interval (p,p)
      sum(left(n)) := 0
      return $sum(\mathbf{n})$
    end if
  else if (p > $key_2(\mathbf{n}) + 1$)
    /* continue search right */
    if ($right(\mathbf{n}) \neq nil$)
      $sum(\mathbf{n})$ := $sum(\mathbf{n}) + 1$
      return dist($right(\mathbf{n})$)
    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 = $key_1(\mathbf{n}) - 1$ AND
           p = $key_2(left(\mathbf{n})) + 1$)
    /* merge left node */
    $key_1(\mathbf{n})$ := $key_1(left(\mathbf{n}))$
    remove_node($left(\mathbf{n})$)
    rebalance(n)
    return $key_2(\mathbf{n}) - p + sum(\mathbf{n})$
  else if (p = $key_1(\mathbf{n}) - 1$)
    /* add to node */
    $key_1(\mathbf{n})$ := p
    return $key_2(\mathbf{n}) - p + sum(\mathbf{n})$
  else if (p = $key_2(\mathbf{n}) + 1$ AND
           p = $key_1(right(\mathbf{n})) - 1$)
    /* merge right node */
    $key_2(\mathbf{n})$ := $key_2(right(\mathbf{n}))$
    remove_node($left(\mathbf{n})$)
    rebalance(n)
    return $sum(\mathbf{n})$
  else if (p = $key_2(\mathbf{n}) + 1$)
    /* add to node */
    $key_2(\mathbf{n})$ := p
    return $sum(\mathbf{n})$
  else
    /* internal error */
  end if
end
```