Chopping Up Trees to Improve Spatial Locality in Implicit *k*-Heaps

Robert Niewiadomski, José Nelson Amaral

{*niewiado, amaral, holte*}*@cs.ualberta.ca*
Department of Computing Science, University of Alberta
Edmonton, AB, Canada

Abstract. Research on the performance of implicit k-heaps has shown that aligning data with cache lines and increasing heap arity are effective techniques for improving the data reference locality of heap operations. The technique of tree blocking has long been used to enhance the data reference locality of tree-based search methods. In this paper we propose c-clustered tree blocking, a new tree blocking method designed to further enhance the data reference locality of implicit k-heap operations. We examine the effect of our method on the performance of a traditional aligned implicit 2-heap using internal memory benchmarks based on the Hold model. Our empirical results, reproduced on four contemporary architectures, show that our method produces speedups of up to 2.0 in either benchmark, while reducing data cache misses by up to 85% and TLB misses by up to 65%. For larger heap arities our method matches the performance of traditional implicit k-heaps while improving page level locality.

1 Introduction

High memory latency and limited memory bandwidth are the most important performance bottlenecks in modern computers. The ever-increasing gap between processor speed and memory latency is providing stimulus for research on cache-conscious algorithms. Compiler designers and programmers seek to increase the reuse of data brought into the cache and to hide memory latency through prefetching. A source of performance improvement that remains largely untapped in the implementation of many algorithms is the implicit prefetching that occurs within the memory hierarchy. This implicit prefetch results from the difference between the size of the data transfer unit between layers of the memory hierarchy and the size of the data unit referenced by a processor. For instance, contemporary architectures implement 64-byte or 128-byte cache lines, but processors reference individual 32-bit or 64-bit data elements. Therefore when one data item is referenced, 7 or 15 additional items are brought into the cache. The current processor design trend is in the direction of implementing even larger cache lines. Standard data reuse techniques do not benefit from this implicit prefetching because the additional data that is brought into the cache may not

be referenced in the first place. To tap this resource, we need to improve the order in which data is referenced.

In this paper we revisit the design of a fundamental data structure: the implicit k-heap. Classic applications of implicit k-heaps include priority queues and the Heapsort algorithm. Priority queues have numerous applications in domains such as discrete event simulation, operating system task scheduling, and greedy algorithms. The Heapsort algorithm facilitates in-place sorting. The efficiency of the heap implementation can significantly affect the performance of programs employing either heap based priority queues or the Heapsort algorithm, particularly in the case of real-time systems.

We present the c-clustered implicit k-heap, as an alternative to the traditional implicit k-heap. The c-clustered implicit k-heap is based on a new form of tree blocking, c-clustered tree blocking, designed to improve the data reference locality of heap operations. The main contributions of this paper include:

- An examiniation of the mismatch between the memory layout of the traditional implicit k-heap and heap operation node access patterns.
- A description of c-clustered tree blocking and the c-clustered implicit k-heap.
- Experimental evidence that c-clustered implicit k-heaps produce speedups of as much as 2.0 over traditional implicit k-heaps, in addition to incurring less cache and TLB misses, in state-of-the-art computer architectures.

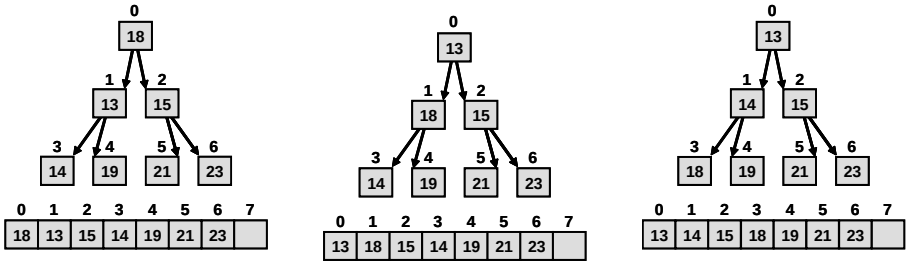
In Section 2 we examine heaps and heap operations, with emphasis on traditional implicit k-heaps. We describe a major shortcoming of the traditional implicit k-heap in Section 3. The next section is then used to introduce the c-clustered implicit k-heap. Section 5 contains a study of the performance of the c-clustered implicit k-heap. Finally, the last two sections are dedicated to related work and conclusions.

2 Heaps and Heap Operations

A *heap* is a data structure formed by a rooted tree that adheres to the *heap-property*. A rooted tree T adheres to the heap-property if and only if for any node $k \in T$, such that if $p = \textsc{Parent}(k)$, $\textsc{Key}(k) \geq \textsc{Key}(p)$.[1] There are four *fundamental* heap operations: $\textsc{Add}(T, x)$ adds node x to T, $\textsc{RemoveMin}(T)$ removes from T the node with a key that is less-than or equal-to the smallest key in T, $\textsc{DecreaseyKey}(T, x, d)$ decreases the key of node x by d, where x is a node in T and d is a positive value, and $\textsc{Remove}(T, x)$ simply removes x from T.

For instance, Figure 1(a) shows a 7-node heap after a copy of the root is made and its leaf node, with key 18, is moved to the root. Figure 1(b) shows T after the first down percolation of 18, and Figure 1(c) shows the tree after the RemoveMin function has completed.

[1] In this paper, the heap property refers to the min-heap property.



(a) Leaf moved to root.

(b) After first swap.

(c) After down percolation.

Fig. 1. RemoveMin Example

In a k-ary tree each node has at most k children. A spatially efficient storage for a k-ary tree is a one-dimensional vector $V[0, \ldots, n-1]$. In an *implicit* k-ary tree the indices of the parent, children, and sibling of $V[i]$ are computed based exclusively on the value of i. In the *traditional* node indexing scheme for k-ary trees, the root's index is 0 and the index of the parent of $V[i]$ is computed by $\left\lfloor \frac{(i-1)}{k} \right\rfloor$. The indexes of the k children of $V[i]$ are $ki+1$ to $ki+k$. When k is a power of 2, computing the index of the parent of $V[i]$, or of $V[i]$'s first child, requires two simple operations: a binary shift and an addition. A *traditional implicit k-heap*, an implicit k-heap based on the traditional node indexing scheme, is space efficient and has a computationaly efficient index computation method.

3 A Memory Layout Mismatch

In [6, 7] LaMarca and Ladner investigate the impact of data caches on the performance of traditional implicit k-heaps. Their study indicates that efficient cache utilization is key to improving heap operation performance. They recommend two techniques for enhacing the data reference locality of heap operations. The first technique minimizes the number of cache lines spanned by any set of sibling nodes. The technique assumes k to be a power of 2 and aligns V such that $V[0]$ maps to the last b bytes of a cache line, where b is the number of bytes required to store $V[0]$. This technique improves the spatial locality of data references made by RemoveMin because the children of $V[0]$ are more likely to fit in one cache line. Their second technique increases the value of k. The height of a nearly complete k-ary tree decreases as k increases. A shorter tree results in Add and RemoveMin performing less percolations. In the case of RemoveMin, however, a larger k may require RemoveMin to perform more comparisons at each level. As k increases, RemoveMin degenerates toward a linear search of V. Thus, if k is too large the benefits of increased fanout may disappear. LaMarca and Lad-

ner produce empirical evidence that their techniques improve the performance of implicit k-heaps by reducing data cache misses.

The techniques outlined by LaMarca and Ladner work well. Could we further improve the cache utilization of an implicit heap? Lets examine the ADD and REMOVEMIN operations in a traditional implicit 2-heap. Revisiting the example of Figure 1, we now examine the node indexing and the implicit vector representation at the bottom of each figure. We make two observations: (1) The traditional node indexing scheme arranges nodes in memory in an order based on a left-to-right breadth-first traversal of the heap, and (2) the node percolations performed by heap operations exhibit node access patterns that are depth-first oriented. For instance, ADD and DECREASEKEY traverse simple paths towards the root, accessing one node at each visited tree level. REMOVEMIN traverses a simple path that originates at the root and advances towards a leaf node. At each traversed level, REMOVEMIN accesses one set of siblings. Consider the traversal of the left-most root-to-leaf path in an aligned traditional implicit k-heap as performed by the REMOVEMIN operation. The indices of the first node visited in each level of the tree are: 0, 1, $(k+1)$, (k^2+k+1), (k^3+k^2+k+1) and so forth. At each level k nodes are referenced. The distance between two sets of k nodes referenced in succession grows exponentially with the distance from the root. As a result, the spatial locality for references between tree levels is poor. Thus, there is a mismatch between the memory layout of the traditional node indexing scheme and the node access patterns of heap operations.

4 c-Clustered Implicit *k*-Heaps

Tree blocking is a form of data clustering where contemporaneously accessed data items are grouped such that they reside in close proximity of each other in memory thereby improving data reference locality. The conventional method of tree blocking partitions the tree into disjoint sub-trees of some limited height to produce a memory layout where the nodes of each sub-tree are packed into a contiguous memory region. Tree blocking improves the spatial locality of memory references made during traversals of root-to-leaf paths and leaf-to-root paths.

A major shortcoming of conventional tree blocking is that sibling nodes can be the roots of the sub-trees produced by the blocking. Since the nodes of each sub-tree are grouped in close proximity of each other in memory, sibling nodes that are sub-tree roots may end-up in distant memory regions. As LaMarca and Ladner point out, having sibling nodes adjacent to each other in memory can enhance the data reference locality of the REMOVEMIN. Our *c-clustered* node indexing scheme implements a form of tree blocking that ensures that siblings reside in close proximity of each other in memory.

Given a blocking factor c, where $c \geq 1$, the c-clustered blocking of a k-ary tree is its decomposition into disjoint sets of sub-trees that share a common parent node and have a height of at most $c-1$.

Let T be a nearly complete k-ary tree with height h, where h is a multiple of c. Given a blocking factor c, where $c \geq 1$, the c-clustered blocking of T is

obtained via a two-step tree decomposition process. The first step partitions T into $h/c+1$ layers: $L_0, ..., L_{h/c}$. Layer L_0 contains only the root node. Layer L_i, $1 \leq i \leq h/c$, contains nodes at depth d, where $d \in [c(i-1)+1, c \times i]$.

Lemma 1. *Given a node $a \in T$, all children of a belong to the same layer L_i.*

Proof. Follows directly from the definition of a layer because all the children of a have the same depth.

The second step partitions each layer of T into one or more *group*. A group is the set of all nodes in L_i that have a common ancestor at depth $c(i-1)$. Because a tree node has exactly one ancestor at a given depth, the groups are disjoint. Since a complete k-ary tree has exactly $k^{c(i-1)}$ nodes at depth $c(i-1)$, L_i features as many groups. It follows that for $1 \leq i \leq h/c$, each group in L_i is composed of k sub-trees with a height of $c-1$. L_0 has only one group containing the root node of T.

Lemma 2. *Given nodes $x, y \in T$, if Parent(x) = Parent(y), then x and y belong to the same group.*

Proof. The lemma is trivially true for L_0 because L_0 has a single node. For groups in L_i, $i > 0$, the proof follows immediately from the definition of a group, since a group is formed by the set of all nodes in L_i that share a single ancestor at depth $c \times (i-1)$.

The c-clustered node indexing scheme implements the c-clustered tree blocking of T. A *c-clustered implicit k-heap* is an implicit k-heap based on the c-clustered node indexing scheme.

Definition 1. *Given a c-clustered tree blocking of a k-ary tree, the c-clustered node indexing scheme sequentially numbers the nodes of each group from left to right and from top to bottom.*

In the c-clustered node indexing scheme the root's index in V is 0. We compute the indices of the parent and child nodes of $V[i]$ as follows.

B is the the number of nodes in each group:

$$B = \frac{k^{c+1}-1}{k-1} - 1 \tag{1}$$

The number of the group to which $V[i]$ belongs to is given by:

$$G(i) = \left\lfloor \frac{i-1}{B} \right\rfloor \text{ if } i > 0 \tag{2}$$

The offset of node $V[i]$ in the group $G(i)$ is given by:

$$\mathit{Offset}(i) = (i-1) \bmod B \text{ if } i > 0 \tag{3}$$

L is the offset of the leftmost and bottomost node in $G(i)$:

$$L = B - k^c \text{ if } i > 0 \tag{4}$$

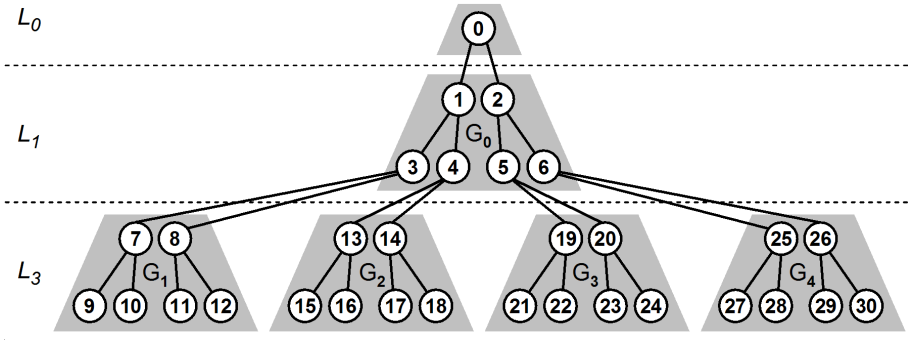


Fig. 2. A 2-clustered implicit 2-heap.

Definition 2. *The nodes in a group are divided into three categories:*

i. $V[i]$ *is a* top node *if the parent of* $V[i]$ *is not in the same group as* $V[i]$, $G(i) \neq G(Parent(i))$*;*
ii. $V[i]$ *is a* bottom node *if the children of* $V[i]$ *are not in the same group as* $V[i]$, $G(i) \neq G(FirstChild(i))$*;*
iii. $V[i]$ *is an* internal node *if both the parent of* $V[i]$*, and the children of* $V[i]$ *are in the same group as* $V[i]$, $G(i) = G(Parent(i)) = G(FirstChild(i))$.

It follows that node $V[i]$ is a top node if and only if $\mathit{Offset}(i) < k$, a bottom node if and only if $\mathit{Offset}(i) \geq L$, an internal node if and only if $k \leq \mathit{Offset}(i) < L$.

If $\mathit{Offset}(i) < L$ then:

$$\begin{aligned} \mathit{ParentOffset}(i) &= (G(i) - 1) \bmod k^c + L + 1 \\ \mathit{ChildOffset}(i) &= (\mathit{Offset}(i) + 1) \times k \\ \mathit{Parent}(i) &= \mathit{ParentGroup}(i) \times B + \mathit{ParentOffset}(i) \\ \mathit{FirstChild}(i) &= i + \mathit{ChildOffset}(i) - \mathit{Offset}(i) \end{aligned}$$

If $\mathit{Offset}(i) \geq L$, then:

$$\begin{aligned} \mathit{ParentOffset}(i) &= \lfloor \mathit{Offset}(i)/k \rfloor - 1 \\ \mathit{ChildOffset}(i) &= 0 \\ \mathit{Parent}(i) &= i - \mathit{Offset}(i) + \mathit{ParentOffset}(i) \\ \mathit{FirstChild}(i) &= \mathit{ChildGroup}(i) \times B + 1 \end{aligned}$$

When $V[i]$ is a top node, its parent's group number is:

$$\mathit{ParentGroup}(i) = \left\lfloor \frac{G(i) - 1}{k^c} \right\rfloor \text{ if } \mathit{Offset}(i) < L \tag{5}$$

Similarily, when $V[i]$ is a bottom node and $Offset(i) \geq L$, its child's group number is:

$$ChildGroup(i) = (k^c \times G(i) + 1) + (L - Offset(i)) \quad (6)$$

Two boundary cases need to be treated outside of the equations above. If $i \leq k$ then $Parent(i) = 0$. If $i = 0$, then $FirstChild(i) = 1$.

Figure 2 shows a 2-clustered implicit 2-heap. The tree is divided vertically into the three layers defined during the first step of the c-clustered tree blocking process. Gray trapezoids outline the node groups produced during the second step. Each trapezoid is annotated with the corresponding group's group number. Finally, all nodes are annotated to show the order in which they appear in V.

Theorem 1. *Given the i^{th} node in V in a c-clustered implicit k-heap, all children of $V[i]$ appear in a contiguous memory region.*

Proof: Consecutive positions in a vector are placed in contiguous memory positions. Thus we have to show that the children of $V[i]$ occupy consecutive positions in V. If $V[i]$ is a top node or an internal node the theorem is trivially true because of definition 1. If $V[i]$ is a bottom node, then according to Lemma 2 the children of $V[i]$ must belong to the same group. According to definition 2 all children of $V[i]$ must be top nodes in their own group. Definition 1 specifies that nodes at the same level within a group must be numbered consecutively. Therefore the children of $V[i]$ must be placed in consecutive positions of V.

In order to improve memory utilization we require the tree to be *groupwise nearly complete*. A tree is groupwise nearly complete if:

i. All group layers, except for the last one, are completely populated with nodes; and
ii. At the last group layer, all nodes occupy the leftmost groups in the layer.

Maintaining the groupwise nearly complete property is trivial. Although this property can result in more node percolations, it greatly simplifies node index computations.

When implementing c-clustered implicit k-heaps, we align the first node of each group such that it maps to the first byte of a cache line. This alignment is an adaptation of the LaMarca and Ladner alignment technique to minimize the number of cache lines spanned by sibling nodes. We pad groups to match the size of cache lines, which are powers of two, to improve dynamic memory usage. During percolation we save the results of equations 5 and 6, thereby eliminating the need to repeatedly compute equation 2. Our implementations perform at most one multiplication per node percolation step; everything else is computed via binary-shifts and bit-masks.

The crux of c-clustered implicit k-heaps is that the additional index computation time is likely to be amortized by the improved memory reference pattern. However, for small heaps, c-clustered implicit k-heap performance is expected to be worse than that of the traditional implicit k-heap because: (1) there is not much latency to hide when the entire heap fits in the higher levels of the memory hierarchy, and (2) padding can cause the heap to no longer fit into these levels.

5 Experimental Evaluation

We compared the performance of the c-clustered k-heap with that of the traditional k-heap using a benchmark based on the Hold model. The following is a summary of our findings:

- When the heap does not fit in cache, the performance of a c-clustered 2-heap can be twice the performance of a traditional 2-heap. Here, a c-clustered 2-heap has superior data reference locality at the cache line and page levels.
- For larger values of k, the performance of c-clustered k-heaps is similar to that of traditional k-heaps. Although c-clustered k-heaps have better page level locality, it only offsets the index computation overhead.

5.1 Experimental Framework

We implemented a c-clustered implicit k-heap for various values of k and c. We pad groups with enough nodes so that the first node maps to the begin of a cache line. Our baseline is an efficient traditional implicit k-heap with LaMarca and Ladner's cache line alignment. Heap nodes contain a 32-bit key and a 32-bit data field. We run these programs, compiled with GCC at `-O3`, in the four machines described in Table 1. All machines have a virtual memory page of 4,096 bytes.

Table 1. Summary of our testbed systems (all systems ran a UNIX based OS).

System	Processor	L1 Data Cache	L2 Data Cache	L3 Data Cache	L1 TLB Entries	L2 TLB Entries	Main Memory
`Itanium`	Itanium II 1.3 Ghz	16 KB 64 byte lines	256 KB 128 byte lines	3 MB 128 byte lines	32	128	2 GB
`Power`	Power4+ 1.7 Ghz	32 KB 128 byte lines	1,440 KB 128 byte lines	128 MB 512 byte lines	1,024	*none*	24 GB
`Pentium`	Pentium 4 2.3 Ghz	8 KB 64 byte lines	512 KB 128 byte lines	*none*	128	*none*	1 GB
`Athlon`	Athlon XP 1.7 Ghz	64 KB 64 byte lines	256 KB 64 byte lines	*none*	32	256	1 GB

Our benchmark, `Hold`, is based on the Hold model as described in [5]. For a heap with p nodes, where each node has a randomly generated key between zero and $p-1$, `Hold` performs fetch-and-return cycles. A fetch-and-return cycle consists of a RemoveMin followed by an Add. As a result, heap size oscillates between $p-1$ and p nodes. The key for each Add is the value of the last removed node's key plus a random value between zero and $p-1$. The hold period refers to the number of fetch-and-return cycles executed. In our experiments we use a hold period of $4p$.

We measured the wall-clock time. Our random number generator is deterministic, portable, and has low-overhead [10]. Each experiment requires less than

the 1 GB of memory available. Thus, memory traffic is constrained to internal memory.

5.2 c-Clustered Heaps Outperform Traditional *k*-Heaps

The first experiment compares the performance of c-clustered 2-heaps to a traditional 2-heap. The best performace gains are for $c = 3$, but heaps with $c = 2$ and $c = 4$ also outperform traditional 2-heaps.[2] Figure 3 presents the speedups produced by a 3-clustered 2-heap over the traditional 2-heap for increasing values of $m = \log_2 n$, where n is the total number of elements in the heap. When n is small, c-clustered heaps are slower than traditional binary heaps. However, for large n the speedup of the c-clustered heap can reach between 1.5 on the `Athlon` to 2.0 on the `Pentium`.

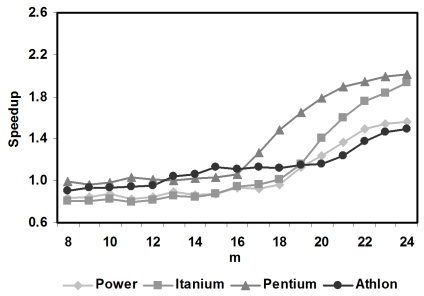


Fig. 3. Speedup k=2, c=3 vs. k=2. Points on the x-axis are annotated with labels where the value of a label indicates the value of n at that point, where $m = log_2 n$.

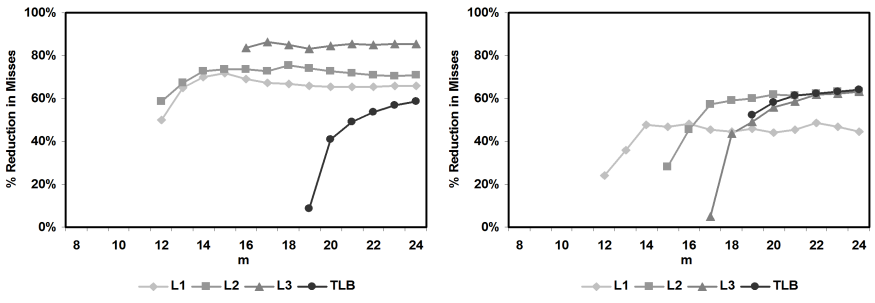
The crosspoints in which a c-clustered 2-heap starts to outperform a traditional 2-heap reveals that the heap has to be large enough to exceed the capacity of intermediate levels of cache, *i.e.,* c-clustered heaps improve performance when the data has to be stored to and fetched from main memory.

The improvements in `Power` are smaller because the payoff for a better memory reference pattern is greater when the penalty of accessing lower levels of the hierarchy is higher. `Power` has a 128 MB L3 data cache with a latency of approximately 100 cycles. This latency is much smaller than the typical latency to access main memory in the other systems. In the case of the `Athlon`, the smaller improvement is likely due to limited bandwidth between the L1 and L2 caches, *i.e.,* the *fill rate* of an L1 cache line on a miss is one fourth of the same rate on the other machines. With this limited fill rate, a c-clustered heap cannot fully benefit from the implicit prefetching offered by large cache lines.

[2] In some instances $c = 2$ outperforms $c = 3$ on the `Athlon`.

5.3 c-Clustering Improves Memory Reference Locality

Figure 4 is a study of the percentage reductions in the total number of L1, L2, L3, and TLB misses incurred by a 3-clustered 2-heap compared to a traditional 2-heap. Results are presented for the `Power` and `Itanium` systems. For clarity of presentation we only show results for instances where the c-clustered heap produced miss reductions. As expected, for small values of n, the c-clustered heap increases the number of misses because a small traditional 2-heap often generates no cache misses. However, once the heap grows large enough we see reductions in cache and TLB misses upwards of 85% and 65%, respectively. These improvements are more than sufficient to offset the 30% to 35% increase in the number of executed instructions observed in 3-clustered 2-heaps compared to the traditional 2-heap.



(a) `Power`. (b) `Itanium`.

Fig. 4. Percentage reduction in the total number of misses incurred by our 3-clustered 2-heap compared to the traditional 2-heap. In the x-axis, $m = log_2 n$.

5.4 k-Heaps with Larger k Are Harder to Beat

Our next experiment compares the performance of c-clustered k-heaps against traditional k-heaps for larger values of k. Our experimental results indicate that $k = 8$ produces the best performance for traditional k-heaps. Figure 5 plots the speedups produced by these heaps over the traditional 2-heap. Compare these curves with the speedup curves of Figure 3 to notice that with the exception of `Power`, the traditional 8-heap produces better speedups over the traditional 2-heap than the 3-clustered 2-heap. How did c-clustered heaps do with higher values of k? We found the performance of 2-clustered 8-heaps to be on par with that of the traditional 8-heap. Hardware event profiles show the 2-clustered 8-heap incurring between 30% to 40% less TLB misses. Cache miss reductions

occured, but they never exceeded 15%. It is likely that these improvement are insufficient to compensate for the overhead of the more complex index computations (10% to 15% more executed instructions). However, the improvements in page level locality are significant. In particular, improved page level locality is likely to pay larger dividends in instances where the heap resides primarily on disk. Furthermore, current trends in microprocessor design have cache line widths growing due to increased memory bandwidth requirements by the CPU. Consequently, the performance of c-clustered k-heaps could improve in future architectures due to the ability to pack more nodes into a single cache line.

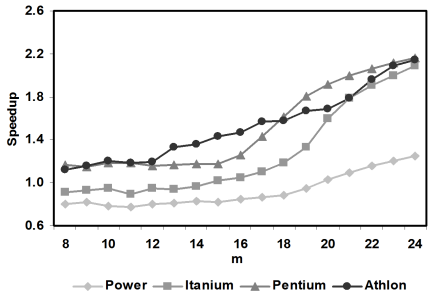


Fig. 5. Speedup k=8 vs. k=2. In the x-axis, $m = log_2 n$.

6 Related Work

LaMarca and Ladner [6, 7] propose the techniques of alignment and increasing tree arity as a means of enhancing traditional implicit k-heap performance. Their empirical results show the implicit k-heap delivering better performance than methods once thought to be superior [5]. For this they credit the increasing gap between CPU speed and memory latency.

The technique of tree blocking appears to originate in the field of *external memory algorithms* where data clustering is used to enhance the data reference locality of path traversals in planar graphs [8, 9]. In [4] Chilimbi *et al.* apply tree blocking to an explicit binary search tree to enhance the performance of a commercial database application. A *cache oblivious* approach to implicit binary search tree blocking is presented in [1]. For top-to-bottom tree traversals they use a combination of the traditional indexing scheme and a translation step using four support vectors. The overhead of this approach results in poor performance for small heaps, but the performance improves when the tree size exceeds the main memory capacity. We experimented with a similar approach for index computation in the c-clustered memory layout and found the computational overhead to be too large when the heap resides in internal memory. Chatterjee *et al.* [3] improve data reference locality for dense matrix codes.

Such techniques improve data fetching, but a compiler cannot re-index the array V or insert padding to benefit from the implicit prefetching that occurs in the memory hierarchy.

7 Conclusion

This is the only work that we are aware of that tailors tree blocking for implicit k-heap implementations. We described an efficient tree blocking method for implicit k-heaps, with the following properties: (1) we can perform cache line alignment, (2) sibling nodes are placed contiguously in memory, (3) the overhead for index computations is low enough to obtain performance improvements in internal memory. The key intuition motivating c-clustering is to use the data implicitly prefetched through large cache lines before referencing data further afield. Future work will encompass a more comprehensive experimental and analytical evaulation of c-clustered k-heaps in addition to exploring the implicit binomial heap [2].

References

1. G. Brodal, R. Fagerberg, and R. Jacob. Cache oblivious search trees via binary trees of small height. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 39–48, 2002.
2. Svante Carlsson, J. Ian Munro, and Patricio V. Poblete. An implicit binomial queue with constant insertion time. In *SWAT 88, 1st Scandinavian Workshop on Algorithm Theory*, volume 318, pages 1–13. Springer, 1988.
3. Siddhartha Chatterjee, Vibhor V. Jain, Alvin R. Lebeck, Shyam Mundhra, and Mithuna Thottethodi. Nonlinear array layouts for hierarchical memory systems. In *International Conference on Supercomputing*, pages 444–453, 1999.
4. Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Cache-conscious structure layout. In *Conference on Programming Language Design and Implementation*, pages 1–12, 1999.
5. Douglas W. Jones. An empirical comparison of priority-queue and event-set implementations. *Communications of the ACM*, 29(4):300–311, 1986.
6. Anthony LaMarca and Richard E. Ladner. The influence of caches on the performance of heaps. *ACM Journal of Experimental Algorithms*, 1:4, 1996.
7. Anthony LaMarca and Richard E. Ladner. The influence of caches on the performance of sorting. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 1997.
8. Anil Maheshwari and Norbert Zeh. A survey of techniques for designing i/o-efficient algorithms. In G. Goos J. Hartmanis and J. van Leeuwen, editors, *Algorithms for Memory Hierarchies*, chapter 3, pages 36–61. Springer, 2003.
9. Mark H. Nodine, Michael T. Goodrich, and Jeffrey Scott Vitter. Blocking for external graph searching. *Algorithmica*, 16(2):181–214, 1996.
10. William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing 2nd Edition*, chapter Chapter 7, Random Numbers, page 284. Cambridge University Press, 1992.