Efficient Dynamic Heap Allocation of Scratch-Pad Memory

Ross McIlroy
Department of Computing Science
University of Glasgow, UK
ross@dcs.gla.ac.uk

Peter Dickman [*]
Google Inc.
Zurich, Switzerland
pwd@google.com

Joe Sventek
Department of Computing Science
University of Glasgow, UK
joe@dcs.gla.ac.uk

Abstract

An increasing number of processor architectures support *scratch-pad memory* – software managed on-chip memory. Scratch-pad memory provides low latency data storage, like on-chip caches, but under explicit software control. The simple design and predictable nature of scratchpad memories has seen them incorporated into a number of embedded and real-time system processors. They are also employed by multi-core architectures to isolate processor core local data and act as low latency inter-core shared memory.

Managing scratch-pad memory by hand is time consuming, error prone and potentially wasteful; tools that automatically manage this memory are essential for its use by general purpose software. While there has been promising work in compile time allocation of scratch-pad memory, there will always be applications which require run-time allocation. Modern dynamic memory management techniques are too heavy-weight for scratch-pad management.

This paper presents the *Scratch-Pad Memory Allocator*, a light-weight memory management algorithm, specifically designed to manage small on-chip memories. This algorithm uses a variety of techniques to reduce its memory footprint while still remaining effective, including: representing memory both as fixed-sized blocks and variable-sized regions within these blocks; coding of memory state in bitmap structures; and exploiting the layout of adjacent regions to dispense with boundary tags for split and coalesce operations. We compare the performance of this allocator against Doug Lea's malloc implementation for the management of core-local and inter-core shared scratchpad memories under real world memory traces. This algorithm manages small memories efficiently and scales well under load when multiple competing cores access shared memory.

Categories and Subject Descriptors D.4.2 [*Operating Systems*]: Storage Management—Allocation/Deallocation strategies; D.3.3 [*Programming Languages*]: Language Constructs and Features—Dynamic storage management; C.1.3 [*Processor Architectures*]: Other Architecture Styles—Heterogeneous (hybrid) systems

General Terms Algorithms, Experimentation, Performance

1. Introduction

It is well known that processor speeds are increasing at a significantly faster rate than external memory chips. This has led to increasing costs for memory accesses relative to processor clock rate, a phenomenon sometimes known as the *memory wall* [23]. Since most applications exhibit temporal locality in data access patterns, memory access overheads can be significantly reduced by storing an application's working set in a smaller but much faster on-chip memory. Many architectures structure this on-chip memory as a cache, where movement of data between on- and off-chip memory is under hardware control. Caches perform very well under most situations, conveniently hiding the non-uniformity of the memory system from software, while decreasing average memory access latency. However, caches suffer from a number of penalties compared with non-associative memories, such as increased silicon area and energy requirements, increased access latency, complex coherency mechanisms for multi-core systems, and lack of realtime guarantees due to potential cache misses. These factors have led a number of processor architectures to provide explicit access to on-chip memory, in the form of *scratch-pad memory*.

Scratch-pad memories are small but very fast and tightly coupled memories, typically ranging from 4kB to 1MB in size. Unlike caches, transfer of data to and from these scratch-pad memories is under explicit software control. They can be found in embedded and real-time processor architectures [5, 16, 20], where they have been found to not only reduce power and silicon area requirements compared to an equivalent cache, but also potentially increase run-time performance [3]. Mutli-core processors, especially heterogeneous architectures [1, 16, 19], have also begun to incorporate scratch-pad memories in their designs. Multi-core architectures can employ this scratch-pad memory either as local memory accessible by only a single core, or as a shared data store that can provide high speed inter-core communication. Explicitly partitioning local and shared data in this way leads to simpler hardware design and can increase overall performance by reducing contention.

Unfortunately, scratch-pad memory places an additional burden on the software developer by being located in an address space disjoint from main memory. While it is potentially possible to statically allocate scratch-pad memory by hand for small systems, automatic management of this memory is required for complex systems. Automatic management of scratch-pad memory as a dynamic heap is possible, however, existing dynamic memory management algorithms are too heavyweight, in terms of memory state requirements, code complexity and execution performance.

Taking these constraints into account led us to design the Scratch-Pad Memory Allocator (SMA), an unconventional algorithm specifically targeted at the management of scratch-pad memories. This algorithm uses a variety of techniques to reduce its overall footprint while still remaining effective. These include: representation of memory both as fixed-size blocks and variable-sized regions within these blocks; use of bitmap based data-structures to code the current state of the memory being managed; and exploiting the ordering of adjacent region split and coalesce operations to dispense with wasteful boundary tags.

SMA is intended as a first step in the process of managing a complex memory hierarchy which contains multiple levels of explicitly accessible memory. We do not expect application developers to be forced to explicitly manage each memory hierarchy level

[*] Work carried out while at the University of Glasgow.

ISMM'08, June 7–8, 2008, Tucson, Arizona, USA.
Copyright © 2008 ACM 978-1-60558-134-7/08/06. . . $5.00

through algorithms such as SMA. Instead, we envisage a runtime system which abstracts the heterogeneous memory hierarchy from the typical user. This runtime system will use annotations, inserted into the code, either by the programmer or a tool, as hints to choose the most appropriate memory level heap for a particular data allocation request. If an allocation request on the preferred heap fails, the runtime will attempt to allocate on a less optimal heap, thereby gracefully handling exhaustion of a particular memory level heap, without burdening the user by having to explicitly deal with multiple heaps. In this paper we present a mechanism which enables efficient allocation of the smallest and most closely coupled memory in such a system, but leave the runtime which supports the full memory hierarchy to future work.

The key contributions of this paper are:

- a simple and lightweight algorithm for managing small areas of memory with minimal overheads
- a technique which employes a coded bitmap to manage region splitting and coalescing without employing boundary tags
- a scalable and lightweight concurrent implementation of this algorithm for management of shared memory by multiple cores

Section 2 overviews related work. The SMA algorithm is presented in Section 3, with some enhancements presented in Section 4. Experimental measurements of the performance of this algorithm and a more conventional allocator are compared in Section 5. Section 6 concludes.

2. Related Work

An extensive range of dynamic memory management strategies has been investigated over the years, many of which are reviewed by Wilson et al [22]. Typically, memory is managed by creating one or more *free-lists* – linked lists where every node is a free region of memory. The memory management algorithm will remove a node from a free-list in order to service a memory request. If the free-list node is larger than the requested size, then it is split, with the remainder being returned to the free-list. When memory is freed, it is coalesced with any neighbouring free regions, and returned to the appropriate free-list. Most memory management algorithms differ primarily in their placement strategy (how they choose the most appropriate free region for a given allocation) and the techniques used to perform splitting and coalescing of memory regions. A number of these techniques, such as binary buddy systems [13], deferred coalescing and bitmap allocation schemes [6], have inspired aspects of SMA.

Much of the work in scratch-pad memory management has eschewed dynamic management techniques due to their associated overheads, and have instead concentrated on allocating memory at compile time. Some of the earliest work into automatic scratch-pad management involves the compiler using this memory to store spilled register values [7]. This work has been extended by others to allow global variables to be allocated to scratch-pad memory by the compiler [3, 18], as well as enabling compiler allocation of stack variables to scratch-pad memory [2, 10]. However, these approaches are limited in that the allocation of scratch-pad memory cannot be modified at run-time.

Use of a software caching mechanism [9] enables automatic management of scratch-pad memory while allowing the set of data stored in scratch-pad memory to change at run-time. However, this approach introduces a run-time overhead for every memory access and leads to unpredictability in memory access times, which may be inappropriate for real-time systems. Kandemir et al [12] describe an approach where the compiler inserts routines to copy stack data from main memory to scratch-pad memory if it expects the data to be accessed frequently. This allows the stack data being held in scratch-pad memory to change depending upon run-time requirements, without the memory access overheads of software caches.

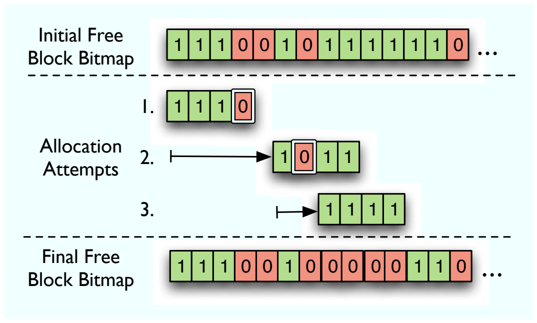

Figure 1. SMA large allocation example.

Dominguez et al [8] extend this approach to provide support for heap data variables as well as stack variables. While dynamically sized variables are supported, only a fixed-sized portion of each variable's data is stored in the scratch-pad, with any remainder relegated to main memory.

Our approach of a truly dynamic run-time heap management algorithm allows full storage of dynamically created data-structures in scratch-pad memory. We see this as a complementary technique to compiler based allocation of scratch-pad memory, allowing a choice between the flexibility of dynamic run-time memory management and the low overhead of compile time allocation.

3. Scratch-Pad Memory Allocator

The SMA algorithm represents memory as a coarse-grained series of fixed-size *blocks*, within which, memory can be split into finely-grained, variable-sized regions (referred to as *small regions* hereafter). Large-sized memory requests are serviced by allocating a large enough contiguous sequence of free blocks to meet the requested size. The last block of a large allocation is potentially split into small regions, such that this space can be reused by small-sized memory requests. Small-sized memory requests are allocated from a series of small regions' free-lists. If the allocated small region is larger than the requested size, any excess is split off and returned to the appropriate free-list. By representing memory as a series of coarse-grained blocks, the data-structure size requirements of SMA are significantly reduced; while allowing these blocks to be split into small regions, if required, alleviates the potential fragmentation this coarse memory representation could otherwise cause.

Many of the algorithms used by SMA consist of simple bitwise operations. These bitwise based algorithms lend themselves to hardware implementation. There is therefore potential for a reconfigurable or custom built embedded processor to significantly accelerate the SMA algorithm by incorporating a small number of simple custom bitwise instructions. We leave hardware acceleration of the SMA to future work.

3.1 Large Block Allocation

SMA partitions all available scratch-pad memory into fixed-sized blocks. Each block is represented by a single bit in a *free-block bitmap*, where a set bit represents a free block, and an unset bit represents an in-use block. The block size is fixed at run-time, but can be varied at compile-time based upon memory size.

To allocate a memory region which is larger than the block's size, SMA allocates enough full blocks to satisfy the allocation request (i.e., $\lceil \frac{request\_size}{block\_size} \rceil$). If the the requested size is not a multiple of the block's size, the last of these allocated blocks is split up to allow smaller-sized requests to reuse the remainder of this block. This section describes the technique SMA uses to allocate the complete blocks for a large-sized request. The process of splitting the final block is the same as that described in Section 3.2.2.

```
void* sma_large_malloc (int no_blocks) {
  void* result= MEM_START;
  int blocks_bm= memstate.b_bm;
  int mask= (1 << no_blocks) - 1; //request size mask

  while ((mask & blocks_bm) != mask) {
    int shift_amount= calc_shift(...);
    blocks_bm= blocks_bm >> shift_amount;
    if (!blocks_bm) { /*alloc failed*/ }
    result += shift_amount;
  }

  memstate.b_bm &= ~(mask << result);
  return result;
}
```

Listing 1. SMA large block malloc algorithm.

A *large-sized* allocation involves SMA searching through the free-block bitmap until it finds a contiguous series of set bits large enough to represent the number of blocks required by the memory request. Figure 1 outlines the method followed for an allocation request requiring 4 blocks of memory. SMA masks off a 4 bit sequence of the free-block bitmap, representing 4 contiguous free blocks, and checks if they are all set. Since one bit is unset in the first attempt, the block represented by the unset bit has been previously allocated, so this attempt fails. The mask is therefore shifted further along the free-block bitmap and the allocation is reattempted. Eventually, either an allocation will succeed or the end of the free-block bitmap will be reached, signalling an allocation failure. A successful allocation clears the contiguous series of set bits it finds, thus reserving the blocks for this memory request.

This algorithm is an *address-ordered first fit* allocation strategy [14], preferentially allocating from the bottom of the available memory space. Wilson et al [22] have shown that this strategy performs well under real world allocation patterns, having low levels of overall fragmentation.

3.1.1 Allocation Algorithm

Listing 1 shows the algorithm used by SMA to service large-sized requests, in C-like pseudo-code. SMA checks whether there are enough contiguous blocks at the current free-block bitmap position, shifting the free-block bitmap and updating the result if the current position cannot satisfy the request. Rather than shifting the free-block bitmap by a single bit on each iteration of this loop, SMA uses the knowledge gained from the previous allocation attempt to calculate how many bits it can skip over. This approach is inspired by KMP string matching [15], but is simpler as only a series of set bits, rather than a string of characters, is being matched.

The only situation in which a valid starting position could be within the previous allocation attempt is when the last bit of the previous allocation attempt is set, otherwise, by definition, all contiguous series of set bits within the allocation attempt are shorter than required by the allocation request. If a potentially valid allocation could begin within the previous attempt, the free-block bitmap is shifted to the only location from which this allocation could possibly begin – the start of the last contiguous series of free blocks encountered in the previous attempt. This location is found by first inverting and masking the result of the previous allocation attempt (Line 7 of Listing 2) to find the blocks, within the previous allocation attempt, that have been previously allocated. The last of these allocated blocks is found using the *find last set bit* (fls) operation, then SMA reattempts allocation from the block following this allocated block. If the allocation cannot start from within the previous attempt (Line 4), SMA finds the next free block after the previous attempt using the *find first bit set* (ffs) operation. Most processors implement the ffs and fls operations in their instruction set.

```
1   int calc_shift (int mask, int blocks_bm) {
2     int attempt= mask & blocks_bm;
3     if !(attempt & ~(mask >> 1) { //is the last bit set
4       int rem_blocks= blocks_bm &~mask;
5       return ffs(rem_blocks);
6     } else {
7       int in_use= mask & ~attempt; //
8       return fls(in_use) + 1;
9     }
10  }
```

Listing 2. Algorithm used to calculate bitmap shift.

```
void* sma_small_malloc (int req_size) {
  req_size = pow2(req_size);
  for (size= req_size; size<BLOCK_SIZE; size<<1) {
    result= free_list_remove_first(size)
    if (result != NULL) {
      sma_split (result, req_size, size);
      return result;
    }
  }
  //no free small regions, split full block
  result = alloc_block();
  sma_split (result, req_size, BLOCK_SIZE);
  return result;
}
```

Listing 3. SMA small region allocation algorithm.

Figure 1 shows both of these scenarios. Allocation attempt 1 fails with the final bit of the attempt unset, therefore, the next attempt is shifted to the first set bit after attempt 1. Attempt 2, also fails, however, the final bit of this attempt is set, therefore, the next attempt is moved to the start of the contiguous series of set bits at the end of attempt 2.

3.1.2 Freeing and Coalescing

Large block freeing in SMA is extremely simple. The contiguous block mask is recreated and shifted up by the block address of the memory being freed. This mask is then OR'd with the free-block bitmap, re-setting those bits which were cleared during the region's allocation. This process implicitly coalesces adjacent free regions.

3.2 Small Region Allocation

SMA uses a different technique to allocate memory requests smaller than a block size. Small free regions of memory are stored in multiple *free lists*. A free list is a linked list where each node is a free region of memory. Since the nodes are held in the free memory they represent, a free list uses very little otherwise usable memory.

SMA limits small-size allocations to a set of *valid* sizes, with invalid size requests being rounded up to the nearest valid size. A single free list for each valid size is used. This approach simplifies the SMA algorithm, since there is no need to search through a free list for an appropriate size. However, the number of valid sizes can have a major effect on memory wastage – too few will lead to unacceptable internal fragmentation, whereas, too many can cause complexity and wasted memory because of the increase in free lists. Due to our desire to limit the complexity of SMA, our current implementation of SMA limits valid *small* allocations' sizes to a power of two, in a similar approach to binary buddy allocators [13]. This enables efficient splitting and coalescing of neighbouring regions.

3.2.1 Allocation Algorithm

Listing 3 shows the general algorithm used by SMA to allocate small sizes. The requested size is first rounded up to a power of two. SMA then checks each free list, from the rounded request size up to the largest valid sub-block size, for an available free region. If

```
void sma_split (void* region, int size_req,
                int size_alloc) {
  for (size= req_size; size < size_alloc; size<<1)
    free_list_insert(size, (region + size));
}
```

Listing 4. SMA small region splitting algorithm.

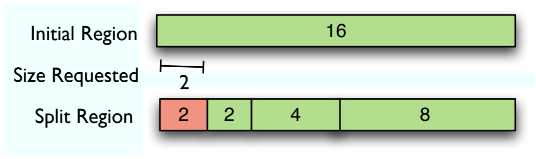

Figure 2. SMA small region splitting example.

a free region is found, it is removed from the free list. If it is larger than the requested size, it is split into a number of smaller regions, with the unused regions being inserted into the appropriate free lists for reuse. If there are no valid regions available in the free lists, one of the fixed-sized blocks is allocated, and split appropriately.

The above algorithm always allocates the smallest available free region that satisfies a given request, leading to a *best fit* allocation strategy [14]. Like the address ordered first fit strategy SMA employs for large region allocation, the best fit allocation strategy has been found to perform well, with limited memory fragmentation, when exposed to real world allocation patterns [22].

3.2.2 Small Region Splitting

Splitting of a region is necessary when the region allocated is larger than the requested size. Since regions are restricted to power-of-two sizes, splitting can be implemented very simply by the pseudo-code in Listing 4. A region is split by repeatedly freeing a region of size x at a position x into the region being allocated, where x starts at the size requested and is incremented by a power-of-two on each loop iteration until it reaches the size of the region which was allocated. Figure 2 shows the result of this splitting process when a region of size 16 is allocated as a result of a size 2 request.

3.2.3 Freeing and Coalescing

Freeing a *small* region involves simply inserting the region into the appropriate free list. However, since regions can be split, it is necessary to consider coalescing of adjacent free regions before they are inserted, otherwise the memory being managed will eventually be fragmented into a large number of small regions.

The standard way of dealing with region coalescing involves *framing* each region with boundary tags [14] and checking the neighbouring tags when performing a free operation to discover if a coalesce operation is necessary. We rejected this approach when designing SMA because boundary tags must frame a region even when it is in use and therefore waste valuable memory which could be otherwise allocated to an application.

Instead, SMA capitalises on the orderly way in which small regions are split into power-of-two sizes. SMA uses a single word sized *coalesce tag* to represent the layout of all the small-sized regions within a fixed-sized block. This coalesce tag (Figure 3) contains a coded representation of the regions into which the block has been split, separated into multiple partitions – one for each possible small region size class. Each bit in a partition represents a possible location for a memory region, of the size being represented by that partition, within the block. If set, a region of that size is free at that relative position within the block. For example, the first 16 bits of the coalesce tag in Figure 3 represent every possible position of a size 2 region within the block, the next 8 bits represent every possible size 4 region, and so on.

```
void sma_small_free (int pos, int size_idx,
                     int coalesce_tag) {
  int t_bit= (0x10 | pos) >> size_idx;
  int c_bit= t_no ⊕ 1; // bit no. of coalesce target

  while (coalesce_tag & (0x1 << c_bit) &&
       size_idx != BLOCK_SIZE_IDX) {
    size = idx_to_size (size_idx);
    free_list_remove(pos + size);
    coalesce_tag &= (0x1 << c_bit);
    size_idx += 1;
    t_bit = t_bit >> 1; //shift to next size class
    c_bit = t_bit ⊕ 1;
  }

  if (size_idx == BLOCK_SIZE_IDX) {
    /*free complete block*/
  } else { // free coalesced region
    coalesce_tag |= (0x1 << t_bit);
    free_list_insert(size_idx, pos);
  }
}
```

Listing 5. SMA small region coalescing algorithm.

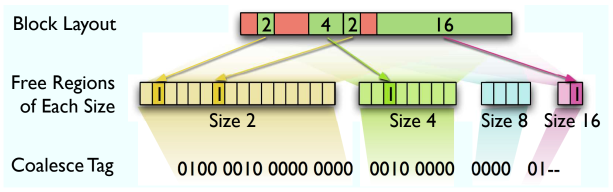

Figure 3. Coalesce Tag example.

The position of the bit representing a particular small region can be calculated with the simple code fragment:

```
tag_bit_no= (0x10 | position) >> size_idx;
```

where `position` is the region's location relative to the start of the block, and `size_idx` represents the region's size class.

When a small region is being freed, SMA checks whether it can be coalesced with its neighbour by examining the coalesce tag. Since only power-of-two sized small regions are valid in SMA, a region can only be coalesced with a neighbour of the same size. Furthermore, each region is aligned to a multiple of its size, therefore, a region can only be coalesced with the single neighbouring region which would create a coalesced region also, aligned to its size. These constraints mean that only one bit of the coalesce tag need be checked before coalescing (`1 xor free_region_bit_no`). If the region is coalesced, this process is repeated to check whether this newly coalesced region can also be coalesced (due to the coalesce tag having a partition for each size class, the bit checked at each iteration of this loop is simply `prev_region_bit_no » 1`). Coalescing repeats until it is either no longer possible, or the regions have coalesced into a fully free block, which is then released by updating the free-block bitmap. This algorithm is shown in Listing 5.

One coalesce tag is required for each fixed-size block which is split into small regions. This tag can be stored at a fixed location within the block. Compared with boundary tags, which require at least one word per allocated region, this approach only wastes one word for all the regions allocated within a fixed-size block. To reduce this overhead even further, we take advantage of the fact that when a fixed-size block has been fully allocated, the coalesce tag contains no real information and can be easily recreated (since no free regions exist in the block, all bits in the coalesce tag are set to zero). Therefore, the final region to be allocated within a fixed-sized block can overwrite the coalesce block, recovering the space used by the coalesce tag. When a region is freed from a previously fully allocated fixed-size block, the coalesce tag can be recreated and stored in the newly freed region. This approach reduces the overhead to only a few bits per fixed-size block (stored outside the block itself): one bit to signal that the fixed size block has been filled and that the coalesce tag should be recreated on the next free from this block; and a small pointer (4 bits for 128 byte block size) to the location of the coalesce tag within the block, since it will no longer be at a fixed location within the block.

4. SMA Enhancements

A number of enhancements can be made to SMA to improve run-time performance, reduce memory overheads and allow concurrent allocation of shared memory by multiple CPU cores.

4.1 Performance Optimisations

One of the most expensive stages in many memory management algorithms is the coalescing and splitting of adjacent memory regions. An algorithm can spend most of its time coalescing memory, only to have to immediately re-split the same memory region. SMA can be modified to defer coalescing of memory regions in the hope of reducing the number of unnecessary splits and coalesces. Memory regions of commonly requested sizes are added to a pool when they are freed, rather than being coalesced immediately. A subsequent request for such a size is allocated from the appropriate pool, without having to split a larger region. Each pool stores multiple free memory regions, all of the same size, as a small array. The choice of memory region size stored by each pool, can be either determined statically ahead of time, or dynamically at run-time. We implemented both options to determine their trade-offs.

SMA can choose the sizes which are pooled at run-time by making use of a small on-core content addressable memory (CAM). As a memory region is freed, its size is added to the CAM, replacing the least recently used size if the CAM is full. The region's address is then appended to the appropriate array. During allocation, SMA checks the CAM to see if any free regions of the requested size are in the pool, returning one of these if they are available.

SMA can also choose a fixed set of sizes to be pooled ahead of time. In this case each pool of free memory regions is statically assigned a single size at compile time. While profiling could be employed to choose the most appropriate sizes, the different treatment of *fixed-size blocks* and *small regions* by SMA makes this unnecessary. The bitmap method used to manage fixed-size blocks implicitly splits and coalesces neighbouring blocks during free and allocation operations, therefore deferred coalescing provides little benefit. However, splitting and coalescing small regions is more expensive. Most of the savings can, therefore, be gained with only the small region size classes being pooled.

4.2 State Memory Reduction

The small size of typical scratch-pad memories increases the relative space overhead required to store the state of memory allocation. While SMA was specifically designed to use as little memory for state as possible, our implementation reduces this overhead further by packing multiple elements into a single word. The small size of scratch-pad memory means that pointers require less than a word to address all of the available memory. SMA exploits this fact to store multiple pointers in a single word. For example, nodes in the doubly linked free-list require a single word, rather than two. Additionally, this technique is used to significantly reduce the size of the deferred coalesce pools described in the previous section. This optimisation reduced the size of the state required by SMA to manage 4kB of memory from 104 bytes to 56 bytes.

4.3 Size Tag Elimination

Most memory management routines store the size of a region just before the start of the region proper. This size is required to free the region, however, it induces additional memory overheads. SMA provides a potential optimisation where the size of the allocated memory region is encoded into the address returned to requesting code instead. Due to the small sizes of scratch-pad memories, the number of bits which are required to represent their address space is smaller than the architecture's word size. These unused bits are usually ignored by the memory subsystem, therefore, SMA can overwrite them with the size of the region. The free operation extracts the size from the address pointer before proceeding with the free. This optimisation is only appropriate for systems where the scratch-pad memory's address-space is smaller than the architecture's word size[1] and distinct from the main memory's address space.

Supplying the size of the region allocated alongside its address could cause some reliability issues, particularly if the encoded size is lost or changes due to operations on the address pointer. However, the size tag, placed in front of allocated regions in more conventional memory allocation systems, is equally vulnerable to being accidentally overwritten by an application. Corruption of the size encoded into a pointer in SMA is only possible when pointer arithmetic overflows or underflows with respect to the scratch-pad memory address size. To guard against any possible corruption, SMA supplies low-overhead macros which enable modification of a pointer without affecting the size encoded within it. If it is not possible to use these macros in existing code (e.g. in external libraries), this optimisation can either be turned off, or the size information can be stripped upon entry to library code, and replaced upon return. This optimisation was used by SMA in all our reported experimental results in Section 5. However, in experiments we performed with this optimisation turned off, we found that roughly half of SMA's reduction in memory wastage compared with Doug Lea's algorithm is due to this optimisation, and the associated coalesce tag optimisation[2] (Section 3.2.3).

4.4 Concurrent SMA

With the increased use of scratch-pad memories in multi-core systems, it is important that the SMA algorithm can be implemented in a concurrency safe way that is scalable to simultaneous access by multiple processing cores. One of the advantages of a scratch-pad memory, compared to caches in a concurrent setting, is the absence of *false sharing*, where two threads may allocate data to the same cache line. False sharing significantly slows each thread's access to this memory due to cache coherency operations. Without cache coherency hardware, false sharing cannot occur in scratch-pad memory systems, therefore, SMA does not implement potentially complex false sharing avoidance schemes, as most other good concurrent memory allocators (e.g. Hoard [4]) must.

Since SMA manages *blocks* and *small-sized regions* with very different data-structures, the most effective form of synchronisation depends upon the data-structure being updated. We use a lock-free update algorithm to protect the block management data-structures, while using locks to protect the small region data-structures.

Since SMA manages large blocks using a simple bitmap data structure, a relatively simple lock-free update algorithm can be used to protect this data-structure from multiple concurrent accesses. To perform a large size allocation using SMA, a thread must read the free-block bitmap, examine it to find an appropriate region of memory to allocate, then write this update to the free-block bitmap by clearing a series of bits. This sequence of operations must

[1] An architecture with 32 bit words can use this optimisation for scratch-pad memories up to 1MB in size, and 64 bit architectures could address multi-gigabyte memories while still performing this optimisation.

[2] Since boundary tags can be encoded in the same memory word as size tags, both boundary tags and size tags must be eliminated before any reduction in the memory overhead caused by these tags is possible.

```
do {
  attempt[]= // find free blocks as before
  for i in 0..len(attempt) {// for each word
    prev= test&clr(blocks_bm[i],attempt[i]);
    if (prev & attempt[i] != 0) {
      alloc_failed= true; // another thread beat us
      // ensure we don't reverse other threads alloc
      attempt[i]&= ~(prev[i] & attempt[i]);
      break;
    }
  }
  if (alloc_failed) {// reverse updates
    for j in 0..i-1 {
      atomic_set(blocks_bm[j], attempt[j]);
    }
  }
} while(alloc_failed);
...
```

Listing 6. SMA concurrent large size allocation pseudo code.

be performed atomically with respect to other threads, otherwise, a race condition exists where one thread can perform an update between the read and update operation of another thread. Since the second thread is not aware of this new update it could allocate its request to the location already allocated to the first thread.

To protect against such races, SMA uses an *atomic test and clear* instruction to update the free-block bitmap during allocations. This instruction reads a location, clears a given sequence of bits at that location, then returns the value just read. This allows a thread to check that the blocks it just allocated are still free and have not been allocated by another thread in the intervening time since it read the free-block bitmap. If any of the blocks are no longer free, the thread backs off by releasing those blocks it did succeed in allocating, and then retries its allocation. Freeing a series of blocks must also be performed atomically, in our case by using an *atomic set* instruction. If these instructions are not available, a similar approach can be taken using *compare and swap* instructions[3].

If an update to the free-block bitmap spans multiple words, and cannot be completed atomically, the free-block bitmap must be updated in order, one word at a time. A potential live-lock condition exists, where thread A performs an update to a word of the free-block bitmap, which fails due to a successful update to the same word by thread B. If thread A then goes on to perform a successful update to the next word of the free-block bitmap, before thread B performs its update to that word, thread B's update will fail. Each thread has caused the other to fail, thereby preventing any overall progress. To prevent this live lock, each thread must check that its update to a particular word has succeeded before progressing to the next update. This lock-free algorithm is outlined in Listing 6.

The data structures used by SMA to manage the small regions are more complex, consisting of a linked list (free-list) for each size class, and coalesce tags for each block of small regions. One way to protect these data structures is to use a single lock which must be held before any thread can perform a small region allocation. This is the approach taken by our *SMA Coarse* implementation. However, this has limited scalability, so we also implemented a more fine-grained locking approach (*SMA Fine*).

For this fine-grain locking, each size class has an individual lock which must be held before a thread can allocate or free an object of this size, increasing potential scalability by a factor of the number of small size classes. These locks protect the free lists, but they must also protect the coalesce tags. They succeed in doing this for common operations due to the coalesce and splitting strategy employed by SMA. A region will only ever be coalesced with another region of the same size, therefore, as long as a thread has locked the size class in which it is interested, it can safely check and update the coalesce tag as required. However, our implementation of SMA allows coalesce tags to be moved within the block which they represent to reduce the memory wastage overhead they cause (see Section 3.2.3). If an allocation operation requires that a coalesce tag be moved (in our experiments this occurred during roughly 15% of allocation operations, but not in any free operations), then the thread must acquire all size class locks (using lock ordering to prevent deadlocks). This ensures that no other thread tries to modify the tag while it is being moved. Since this occurs relatively infrequently, the overall effect on scalability is limited.

5. Results and Analysis

In order to evaluate the effectiveness of the this Scratch-Pad Memory Allocator, we investigated its performance when servicing requests from real world application memory traces. We created an implementation of SMA for the Intel IXP network processor, which was chosen as a typical example of a heterogeneous multi-core architecture. We feel these results can be generalised across most other heterogeneous multi-core processors, as well as embedded processors, with scratch-pad memories.

The Intel IXP network processor [1, 21] is targeted at network packet processing applications. As a heterogeneous multi-core architecture, the IXP consists of different processing core types, specifically: an xScale for control purposes; and a number of *micro-engines* for data processing. We implemented SMA for the micro-engine cores as these employ a variety of scratch-pad memories.

Each micro-engine has access to two scratch-pad memories, one local to that particular micro-engine, and one shared between all the cores on the system. On the IXP 2350 hardware used in our evaluation, each micro-engine has 4kB of local scratch-pad memory, with a 1-4 cycle access latency and 16kB of shared scratch memory, with an access latency of 50-100 cycles depending upon bus contention. Since the shared scratch-pad is concurrently accessible to multiple executing cores, the micro-engines can execute certain atomic operations on this memory (e.g. test and set, compare and swap, etc.), allowing synchronisation of data access.

5.1 Experimental Set-up

These experiments were run on a cycle accurate simulator of the Intel IXP 2350. A simulator was used for convenience reasons only; identical results were achieved on real hardware. The IXP 2350 contains four micro-engine cores, clocked at 900MHz. In most of these experiments only one micro-engine is enabled. The shared memory concurrency experiments (Section 5.5) are an exception, increasing the number of micro-engines executing to investigate the scalability of the SMA algorithm.

To gain realistic memory allocation traces for these experiments, the memory allocation and free requests of a number of commonly used Unix applications were traced with the mPatrol library[4]. The applications traced were: **a2p** – conversion of a 15kB text file to postscript; **gcc** – compilation of the file "combine.c" in the gcc source, using gcc; **gst** – ghostscript extraction of a 682kB postscript file; **cvt** – application of the charcoal filter to a 1024x768 Jpeg image using ImageMagick; **ogg** – encoding of a 20 second wav file using the ogg encoder; **pyt** – execution of the python example file "md5driver.py"; and **tar** – archive and gzip compression of 27 files in 4 directories into a 1Mb archive.

These application runs all request much more memory than is available in scratch-pad memory, therefore, the traces used in these experiments are subsets of the original traces. Our modification of

[3] An advantage of using the atomic bitwise instructions is that multiple threads can perform non-overlapping updates concurrently without interfering, whereas, with compare and swap instructions, only updates on different words of the free-block bitmap can succeed concurrently.

[4] http://www.cbmamiga.demon.co.uk/mpatrol

Resource	SMA	SMA (LM)	SMA (CAM)	DLmalloc
State Memory (Bytes)	40	56	104[1]	516
Code Memory (Instructions)	297	364	443	1634
Avg. Allocation Time (Cycles)	72.8	53.2	50.9	70.7
Avg. Free Time (Cycles)	52.4	36.9	34.6	95.2

[1]64 of these words are in content addressable memory

Table 1. Resources required to manage 4kB local memory.

Resource	SMA	SMA (LM Defer)	DLmalloc
State Memory (Bytes)	112	128	516
Code Memory (Instructions)	430	503	1723
Avg. Allocation Time (Cycles)	335.6	150.3	530
Avg. Free Time (Cycles)	207.4	189.2	721

Table 2. Resources required to manage 16kB shared memory.

these traces first removed all allocations for sizes bigger than the size of memory being managed. Randomly chosen allocation requests and their corresponding frees were then repeatedly removed until the trace's peak requested memory was below 95% of the size of the scratch-pad memory being managed. This approximates an approach where a runtime system chooses whether to allocate memory requests in scratch-pad or non-scratch-pad memory based upon the size of the request and addition data usage information in the form of annotations as proposed in Section 1. The allocation requests retained in these subset traces represent allocation requests which had a "use fastest" or similar annotation. Those thrown out would be allocated from the slower main memory heap. Ten subset traces were created for each original trace, with error bars showing the standard deviation between these ten traces.

We contrast SMA's performance with Doug Lea's malloc implementation [17]. Doug Lea's malloc (referred to as DLmalloc from here onwards) forms the basis of the Linux allocator in the GNU C library, and is widely considered to be one of the fastest and most space efficient allocators available. We considered other well used memory allocators, such as FreeBSD's PHKmalloc [11], however, these allocators often make optimisations which do not suit size constrained scratch-pad memories, such as, limiting each page to allocations of a single size class. Version 2.8.3 of DLmalloc was ported to the IXP micro-engine core. While operating system specific calls, such as requesting additional memory pages, were removed, the algorithm itself remained unchanged. Every effort was made to optimise the compilation of DLmalloc for the IXP micro-engine core, however, no attempt was made to alter DLmalloc's underlying algorithm in order to make use of IXP specific features such as the CAM.

5.2 Overall Performance

Table 1 outlines the resources required by both SMA and DLmalloc for management of the 4kB local memory available to IXP micro-engines. SMA uses a block size of 128 bytes and can further subdivide these *large-size blocks* into *small regions* of sizes 8, 16, 32 or 64 bytes. As well as the standard SMA algorithm, we implemented both the local memory and content addressable memory deferred coalesce region pools as described in Section 4.1 – SMA (LM Defer) and SMA (CAM Defer) respectively.

The amount of memory required for state information by SMA to represent the memory being managed is less than 8% of that required by DLmalloc. SMA (LM Defer) has a pool for each small size class, in this case four, to defer coalescing of these sizes. Our implementation stores each pool in a single word[5], leading to an additional 4 words (16 bytes) of state and 11% of the overall state memory requirements of DLmalloc. SMA (CAM Defer) uses all of the 16 word content addressable memory to store the deferred coalesce pools, thereby using 20% of the state memory required by DLmalloc. With respect to code complexity, SMA uses only 18% of the code required by DLmalloc. Adding deferred coalescing slightly increases SMA's code requirements to 22% and 27% of DLmalloc for the LM and CAM pools respectively.

At 16kB, the shared scratch-pad memory is four times larger than the micro-engine local memory. The same 128 byte block size is used by SMA in these shared scratch-pad memory experiments. We dispensed with SMA (CAM Defer) as it uses more resources than SMA (LM Defer), without providing a significant performance improvement.

The state memory requirements of SMA (Table 2) are slightly increased compared with the local memory figures, however they are still less than 22% of DLmalloc's requirements (25% when deferred coalescing is implemented). The increase, compared to the local memory implementation, is due to a larger free-block bitmap required to represent the larger shared scratch-pad and an increased number of coalesce tag pointers. This could have been alleviated by increasing the block size, at the expense of higher internal fragmentation. While DLmalloc's static state memory requirements have not changed, compared with its local memory implementation, examining the causes of memory wastage (Figure 11) shows that its dynamic state memory requirements have increased due to boundary tags. Code complexity of all the algorithms is slightly increased compared with the local memory implementations, due to the more complex nature of the memory being accessed. However, SMA is still significantly less complex, requiring 25% (29% with deferred coalescing) of the instruction memory needed by DLmalloc.

5.3 Execution Performance

The average execution time required to service an allocation request, or free a region of memory across all traces is also presented in Tables 1 and 2. These results are split by benchmark application and normalised to Doug Lea's malloc algorithm in Figures 4 and 5 for the 4kB local memory experiments, and Figures 6 and 7 for the 16kb Shared memory experiments.

SMA's allocation performance, when managing the 4kB local memory, is roughly comparable to that of DLmalloc (about 11% slower on average). However, the performance of SMA during free operations is considerably faster than that of DLmalloc, due to SMA's simplified coalescing scheme. On average, frees are 43% cheaper than DLmalloc, leading to an overall reduction in memory management performance overhead across the traces overall.

Analysis of execution performance across the benchmark applications shows that the SMA algorithm performs worst when a trace consists of a large number of very small allocations (i.e. 2-4 words such as in a2p, gcc, cvt and tar). Small memory allocations cause SMA to have to regularly split up larger regions to service these requests, while freeing these small regions causes multiple coalesces.

[5]Three pointers can be held in a single 32 bit word, therefore each size class pool can defer coalescing of up to three elements. Doubling this to six elements did not significantly improve performance.

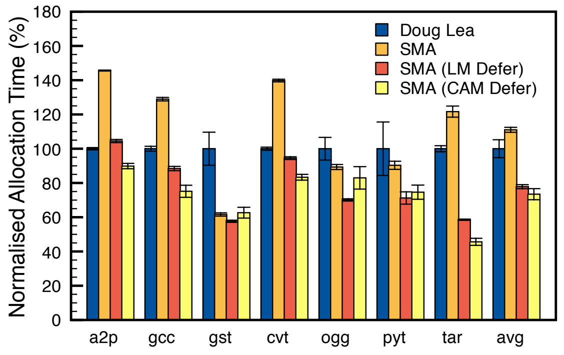

Figure 4. 4kB local memory allocation performance

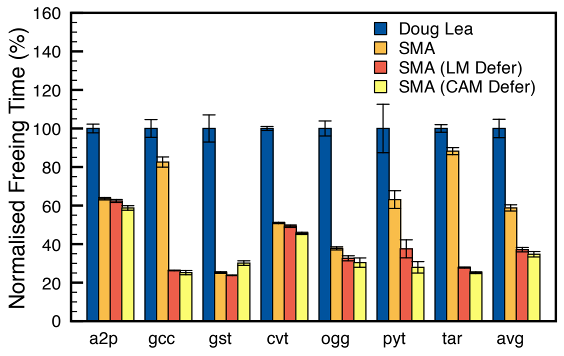

Figure 5. 4kB local memory free performance

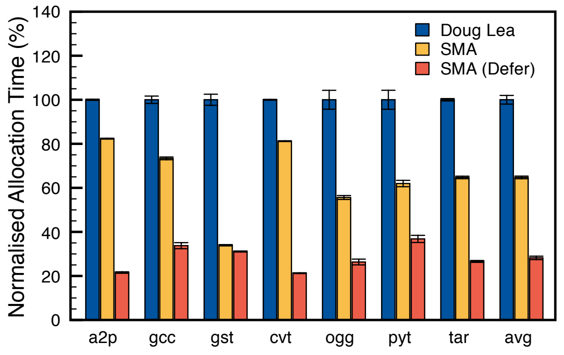

Figure 6. 16kB shared memory allocation performance

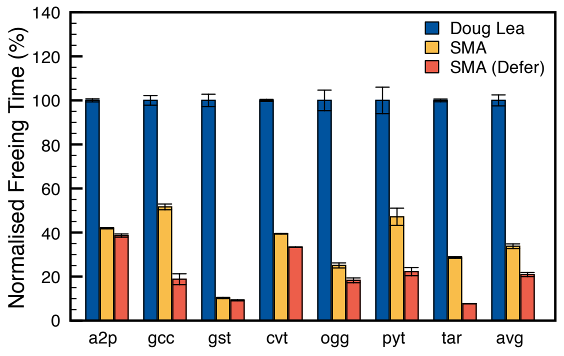

Figure 7. 16kB shared memory free performance

Deferred coalescing significantly reduces the overhead of small region allocation in SMA. Most traces show 20-60% improvement when using either the CAM or LM schemes. The improvement is slightly better for the CAM scheme as it can defer coalescing of any memory region size, while the LM scheme can only defer coalescing for sub-block sized regions. However, the difference was more marginal than we expected, suggesting that typical applications request such a wide variation of sizes that the most benefit is seen over the small subset of sizes which are guaranteed to be reused. Overall, SMA (LM Defer) and SMA (CAM Defer) perform 25% and 29% better than DLmalloc respectively when allocating memory, and about 64% better when freeing memory.

The overall number of cycles required to allocate or free a region of memory in the shared 16kB scratch-pad is much greater than that in local memory, due to the significantly higher access latencies of this memory. Since much of the time is spent blocked waiting for memory accesses, the micro-engine cores can switch to another unblocked hardware thread to reclaim these wasted cycles. The actual number of cycles spent doing useful work for these allocators is comparable to that of the local memory allocators.

The allocation performance of SMA is superior to DLmalloc when managing this shared scratch-pad memory (Figure 6), requiring on average 39% fewer CPU cycles to service a memory allocation request compared with DLmalloc. Deferred coalescing shows an even greater performance improvement for the shared memory implementation, with an average of 73% fewer cycles compared to DLmalloc. This is due both to the deferred coalescing and the lower latency for accessing the memory regions in the deferred coalesce pool as it is stored in local memory, as opposed to shared memory. The number of cycles required for an average free operation using SMA is 70% lower than DLmalloc, with deferred coalescing improving this further to 74% fewer cycles than DLmalloc (Figure 7).

The increase in performance of SMA compared with DLmalloc is much larger when managing the high latency shared IXP memory than the low latency local IXP memory. This implies that SMA accesses memory less frequently than DLmalloc, and should therefore scale better as memory latency continues to increase relative to CPU clock speed. It is also interesting to note that with deferred coalescing, both allocating and freeing from shared scratch-pad memory using SMA is less costly than a single access to external DRAM memory, which costs 200 cycles on average in this architecture.

5.4 Memory Wastage

Memory allocators contribute to a loss of potentially usable memory through a number of factors: boundary tags; internal fragmentation; data structures storing state about the managed memory; and external fragmentation. Figure 8 shows that SMA, on average, wastes 40% less memory than DLmalloc when managing the 4kB local memory. Figure 9 shows that this reduction is mostly due to the reduced memory state required by SMA. Deferred coalescing increases the state memory requirements, causing a slight increase in overall memory wastage (the LM implementation seems to buck this trend due to a reduction in external fragmentation, likely due to favourable memory layout caused by the delayed coalescing). When managing the 16kB shared memory (Figure 10) SMA has, on average, a 15% reduction in memory wastage, compared to DLmalloc (12% with deferred coalescing). Figure 11 shows that the main cause of memory wastage under SMA is internal fragmentation. This is largely down to SMA's limit of power-of-two sizes for small memory regions, which causes non-power-of-two small sized allocations to be potentially rounded up by a significant percentage. Examining the trace files shows that those traces which performed well under SMA mainly requested sizes close, or equal to a power of two, while those which did not, performed less well.

In order to investigate the scalability of SMA to scratch-pad memories, which are larger than those available on the IXP processor, we modelled the SMA algorithm in Python and applied this model to the management of memories in a range from 1kB

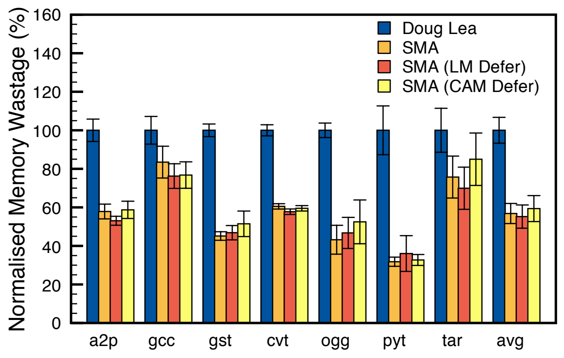

Figure 8. 4kB local memory wasted

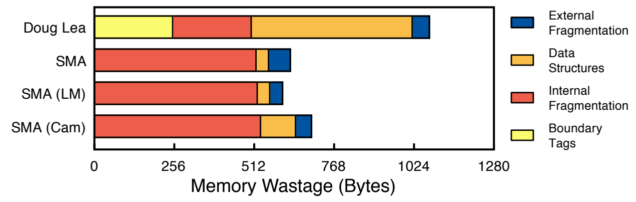

Figure 9. 4kB local memory wastage causes

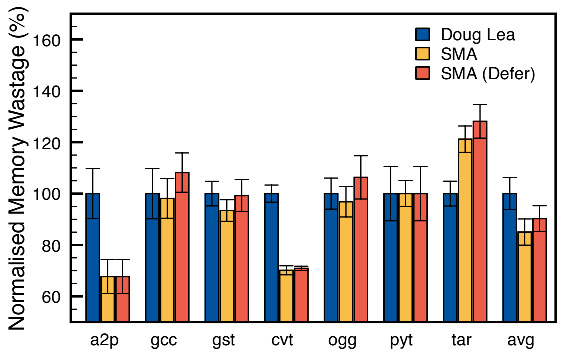

Figure 10. 16kB shared memory wasted

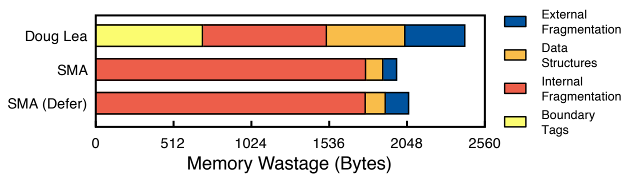

Figure 11. 16kB shared memory wastage causes

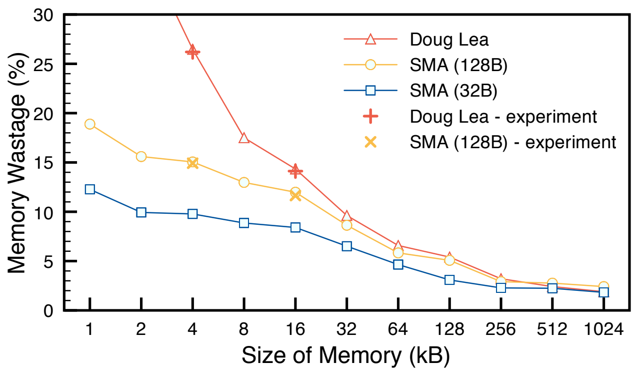

Figure 12. Memory wastage across memory sizes.

to 1MB. Traces were collected in a similar manner to those used in the previous experiments (with the 4kB and 16kB traces being the same as those used before). Figure 12 shows the overall percentage of memory lost due to the SMA and DLmalloc allocation algorithms across different memory sizes. The experimental results from the previous graphs are overlaid to show the minimal difference between modelled and experimental results.

The line marked *SMA (128B)* corresponds to the above experiments where a block size of 128 bytes was used. Once the size of memory reaches 128kB, the state memory reduction provided by SMA when compared to DLMalloc, becomes negligible compared to overall memory size. However, by reducing the block size, the amount of internal fragmentation caused by SMA can be reduced, as fewer sizes need to be rounded up to a power-of-two. If the block size is reduced to 32 bytes (*SMA (32B)*), SMA wastes less memory overall, and does not reach the percentage of memory wasted by DLmalloc until the memory is half a megabyte in size. Although we have yet to perform experiments with SMA using 32 byte blocks in hardware, we would expect the increase in the number of overall blocks, required to represent memory, would cause a slight slowdown in SMA execution performance.

5.5 Concurrent Access Scalability

In order to examine the scalability of the SMA algorithm, we implemented the two concurrent SMA algorithms described in Section 4.4: *SMA Coarse*, which protects memory state with a single lock; and *SMA Fine*, which protects each size class with an individual lock. Both of these concurrent implementations manage the 16kB IXP scratch-pad memory shared across the micro-engine cores. Since SMA (Defer) stores the pool of deferred coalesce regions in local memory, these free regions are inaccessible to other cores. Under certain situations this isolation of memory regions can lead to premature memory exhaustion, especially in a producer consumer relationship where one core allocates data objects and another core frees these objects. However, these pools are bounded and limited to storing small-sized memory regions. Therefore, the maximum amount of memory which can be isolated is 360 bytes per core (3 elements of each small size class).

The same memory traces were used for these concurrency measurements as the single threaded case, however, each of the operations in a single trace was randomly assigned to an available core. Memory regions are not necessarily freed by the same core which allocated them, simulating a producer / consumer relationship for some of the data. All cores do nothing other than memory allocation and free operations repeatedly, with no work executed between memory management operations. This would be very unusual in a real world application and simulates the worst case contention.

Figure 13 shows the average speed-up in the time required for each algorithm to complete all the memory operations in a given trace, as the number of threads executing in parallel increases. Both locking mechanisms incur less than 6% overhead compared to the non-locking implementation of SMA in a single threaded environment. The use of a test and set operation to implement per-size class locking means that SMA Fine incurs less than 1% additional single thread overhead compared with SMA Coarse, while scaling significantly better. Without deferred coalescing, SMA Coarse realises only a 1.6x speedup with four times as many cores executing. The fact that SMA Coarse can scale at all is due to the fact that large allocations use a lock free algorithm, and so are not serialised by the single coarse lock. SMA Fine scales better, with a 2x performance increase at four cores.

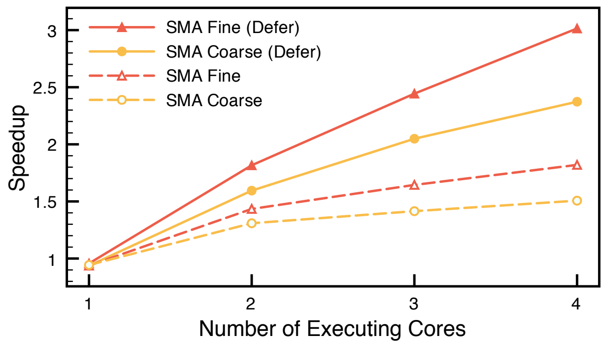

Figure 13. SMA concurrency scaling

With deferred coalescing incorporated, not only does the average performance increase, but the scalability of the SMA algorithm also improves. Deferred coalescing helps lower overall lock contention, since requests which can be serviced from the deferred coalesce pool do not have to synchronise with other threads. This leads to a 2.4x speed up for SMA Coarse and a 3x speed up for SMA Fine at four cores.

6. Conclusions

In this paper we have proposed SMA, a lightweight dynamic heap memory management algorithm targeted at small scratch-pad memory management. This algorithm reduces the size of data structures required to manage memory by: initially representing memory coarsely, only resorting to fine-grained management where necessary; use of coded bit-maps for compact data-structures; and elimination of boundary tags in favour of coalesce tags that can be hidden in unused memory. Code complexity and execution times are low thanks to simple bitmap based algorithms.

Our current implementation of SMA suffers from higher internal fragmentation than DLmalloc, therefore, initial future work will focus on reducing this overhead, most likely through removal of the power-of-two restriction on small memory sizes. Further work will concentrate on hiding the disjointed memory spaces of SMA managed scratch-pad memory and main memory, through the use of a virtual machine based runtime system which can use code annotations to inform its data placement decisions.

The propagation of scratch-pad memories to more complex and general purpose architectures will increase the need for effective automatic management of this memory. Our experience with SMA suggests that dynamic management of scratch-pad memory is both practical and beneficial.

7. Acknowledgments

We would like to thank Doug Lea for his open source implementation of DLmalloc. We thank the anonymous reviewers for their insightful comments. We are also grateful to the Carnegie Trust for the Universities of Scotland who funded this work.

References

[1] M. Adiletta, M. Rosenbluth, D. Bernstein, G. Wolrich, and H. Wilkinson. The Next Generation of Intel IXP Network Processors. *Intel Tech. Journal*, 6(3), 2002.

[2] O. Avissar, R. Barua, and D. Stewart. An optimal memory allocation scheme for scratch-pad-based embedded systems. *ACM Trans. on Embedded Comp. Sys.*, 1(1), 2002.

[3] R. Banakar, S. Steinke, B.S. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. *10th Int'l Symp. on Hardware/Software codesign*, 2002.

[4] E.D. Berger, K.S. McKinley, R.D. Blumofe, and P.R. Wilson. Hoard: a scalable memory allocator for multithreaded applications. *ACM SIGPLAN Notices*, 35(11):117–128, 2000.

[5] D Brash. Arm-v6 architecture. White Paper, 2002.

[6] JM Chang and EF Gehringer. A high performance memory allocator for object-oriented systems. *Computers, IEEE Transactions on*, 45(3):357–366, 1996.

[7] K. D. Cooper and T.J. Harvey. Compiler-controlled memory. In *8th Int'l conf. on Architectural support for programming languages and operating systems*, 1998.

[8] A. Dominguez, S. Udayakumaran, and R. Barua. Heap data allocation to scratch-pad memory in embedded systems. *Journal of Embedded Computing*, 1(4), 2005.

[9] E.G. Hallnor and S.K. Reinhardt. A fully associative software-managed cache design. *SIGARCH Comput. Archit. News*, 28 (2), 2000.

[10] J.D. Hiser and J.W. Davidson. EMBARC: an efficient memory bank assignment algorithm for retargetable compilers. *ACM SIGPLAN Notices*, 39(7), 2004.

[11] Poul-Henning Kamp. Malloc(3) revisited. In *Proc. of the Annual Technical Conf. on USENIX*, 1998.

[12] M. Kandemir, J. Ramanujam, J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh. Dynamic management of scratch-pad memory space. In *38th conf. on Design automation*, 2001.

[13] K.C. Knowlton. A fast storage allocator. *Communications of the ACM*, 8(10):623–624, 1965.

[14] D.E. Knuth. *The Art of Computer Programming. Vol. 1: Fundamental Algorithms*. Addison-Wesley, 1973.

[15] D.E. Knuth, J.H. Morris Jr, and V.R. Pratt. Fast Pattern Matching in Strings. *SIAM Journal on Computing*, 1977.

[16] D Krishnaswamy, R Stevens, R Hasbun, J Revilla, and C Hagan. The Intel PXA800F wireless Internet-on-a-chip architecture and design. In *IEEE Custom Integrated Circuits*, 2003.

[17] D. Lea. A memory allocator, 1996. http://gee.cs.oswego.edu /dl/html/malloc.html.

[18] P.R. Panda, N.D. Dutt, and A. Nicolau. On-chip vs. off-chip memory: the data partitioning problem in embedded processor-based systems. *ACM Trans. on Design Automation of Electronic Systems*, 5(3), 2000.

[19] D. Pham, S. Asano, M. Bolliger, M.N Day, H.P Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, et al. The design and implementation of a first-generation CELL processor. *IEEE Solid-State Circuits Conference*, 2005.

[20] J. Scott, L.H. Lee, J. Arends, and B. Moyer. Designing the Low-Power MCORE Architecture. *Power Driven Microarchitecture Workshop*, 1998.

[21] Tammo Spalink, Scott Karlin, Larry Peterson, and Yitzchak Gottlieb. Building a robust software-based router using network processors. *SIGOPS Oper. Syst. Rev.*, 35(5), 2001.

[22] P.R. Wilson, M.S. Johnstone, M. Neely, and D. Boles. Dynamic Storage Allocation: A Survey and Critical Review. *Int'l Workshop on Memory Mgmt.*, 1995.

[23] W.A. Wulf and S.A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Comp. Arch. News*, 1995.