Concurrent Non-deferred Reference Counting on the Microgrid: First Experiences[*]

Stephan Herhut[1], Carl Joslin[1], Sven-Bodo Scholz[1],
Raphael Poss[2], and Clemens Grelck[2]

[1] University of Hertfordshire, United Kingdom
{s.a.herhut,c.a.joslin,s.scholz}@herts.ac.uk
[2] University of Amsterdam, Netherlands
{r.c.poss,c.grelck}@uva.nl

Abstract. We present a first evaluation of our novel approach for non-deferred reference counting on the Microgrid many-core architecture. Non-deferred reference counting is a fundamental building block of implicit heap management of functional array languages in general and Single Assignment C in particular. Existing lock-free approaches for multi-core and SMP settings do not scale well for large numbers of cores in emerging many-core platforms. We, instead, employ a dedicated core for reference counting and use asynchronous messaging to emit reference counting operations. This novel approach decouples computational workload from reference-counting overhead. Experiments using cycle-accurate simulation of a realistic Microgrid show that, by exploiting asynchronism, we are able to tolerate even worst-case reference counting loads reasonably well. Scalability is essentially limited only by the combined sequential runtime of all reference counting operations, in accordance with Amdahl's law. Even though developed in the context of Single Assignment C and the Microgrid, our approach is applicable to a wide range of languages and platforms.

1 Introduction

Functional programming languages are particularly suitable for concurrent execution due to their side-effect-free nature. However, when run on a conventional von Neumann architecture, side effects can ultimately not be avoided. Values from the functional world need to be manifested in heap memory. Managing this heap brings back some of the challenges that imperative programming faces when it comes to concurrent execution. Fortunately, in the context of functional languages, the added complexity remains confined to the programming language's runtime system. Commonly, heap management is implemented by means of deferred garbage collection: heap usage is monitored, and, whenever a high-water mark is reached, program execution is interrupted and dead objects in the heap are identified and removed. Following Amdahl's law, performing garbage collection sequentially would introduce a serious detriment to scalability. Parallel

garbage collectors [16,5] alleviate this problem to some extent, but their scaling is limited by the inevitable locking of heap objects during collection.

The functional array language SAC [17,11] uses a different approach to heap management, namely non-deferred reference counting [4]. Each heap object is accompanied by a counter that maintains the number of live references during the object's life time and de-allocates the heap object as soon as no references are left. While memory management overhead is nicely parallelised alongside an application itself, this technique suffers from a similar problem as deferred garbage collection: updating the reference counter of a heap object requires exclusive access. Unfortunately, locking can quickly over-sequentialise a data parallel program, leaving most threads waiting for some lock to become available. Lock contention has a progressively detrimental effect on runtime performance as concurrency increases. Furthermore, performing reference-counting operations on different cores requires the reference counter to be communicated between cores. Most emerging many-core architectures support only weak memory consistency. Thus, accessing a reference counter from different cores involves explicit invalidation of caches and synchronisation of memory to ensure a consistent global view of reference counters. Both aspects are very costly.

With only a small number of threads, as in current mainstream multicores, locks can be avoided by managing thread-local copies of reference counters [12]. However, the runtime cost for maintaining thread-local reference counters and collating them into a globally consistent view whenever necessary is linear in the number of threads. The same holds for memory overhead. On the Microgrid architecture [3] (or any other many-core system) with its large number of cores and thousands of hardware threads, one reference counter per object per thread is not viable as the inflicted overhead would quickly outgrow the actual workload.

In this paper, we present an alternative approach to lock-free concurrent reference counting that can indeed be efficiently implemented on many-core architectures in general and specifically on the Microgrid. We make use of two specific features: *exclusive places* and *delegation.* An *exclusive place* is a dedicated hardware resource that is guaranteed to run a thread to completion without interleaving. This ensures single-threaded access to heap objects if all such attempts originate from the same exclusive place. *Delegation* allows any thread running on any core to delegate the execution of code to another place, be it exclusive or not. Such delegation requests can be performed synchronous, *i.e.*, the delegating thread waits for completion of the delegated task, or asynchronous, *i.e.*, the delegating thread directly continues execution. The use of exclusive places and delegation for reference counting is based on two observations. Firstly, reference-counting operations of different threads can be interleaved, as long as each thread's reference counting operations remain in order. Secondly, reference counting operations do not need to be executed before a thread can continue; it suffices if they are executed eventually. Thus, on the Microgrid we delegate all reference-counting operations on some heap object to a single exclusive place. Since the delegation mechanism does guarantee the order of requests, all interleavings are safe.

Yet, using just delegation has the same drawbacks as using locks would have: If multiple threads issue a reference-counting request, one thread will be blocked until the other thread is serviced. In contrast to locks, however, delegation can alleviate this effect: as the result of the reference-counting operation is not required for either thread to continue, reference-counting requests can be executed asynchronously with the main computation. Hence, we make use of asynchronous delegation. As first experiments show, asynchronous delegation allows us to hide the latency of reference counting operations for a large range of workloads and varying numbers of processing cores.

The remainder of this paper is structured as follows. The next section gives a brief introduction to SAC and motivates the use of reference counting for heap management. An overview of the Microgrid architecture is given in section 3. Next, we present our distributed approach for non-deferred reference counting in section 4. In section 5 we discuss experimental results. We discuss related work in section 6 before we conclude in section 7.

2   SAC and Non-deferred Reference Counting

Single Assignment C, or SAC for short, is a data-parallel, purely functional programming language with a strong emphasis on processing truly multidimensional arrays. While on the syntactic level SAC very much resembles ANSI C, the semantics of SAC is based on the principle of context-free substitution of expressions rather than the step-wise manipulation of state. This choice facilitates far-reaching compiler-directed program transformations for optimisation [10] and parallelisation [8]. We refer the interested reader mainly to [11] for a thorough introduction to the design rationale of SAC, the ambivalence of functional and imperative interpretation of C-like code and the essence of code transformation in the SAC compiler.

It is a design principle of SAC not to provide aggregate array operations as built-in operations, but rather SAC features a versatile array comprehension construct to define such aggregate operations in SAC itself. Figure 1 shows a

$$
\begin{aligned}
\textbf{with} \quad &\{\,(\,lower\_bound_1 \leq idxvec < upper\_bound_1\,) \;\; : \;\; exp_1\,; \\
&\;\;\; \ldots \\
&\;\;\, (\,lower\_bound_n \leq idxvec < upper\_bound_n\,) \;\; : \;\; exp_n\,; \\
\}\; &: \;\textbf{genarray}\,(\,shape\,,\;\; default\,)
\end{aligned}
$$

Fig. 1. The WITH-loop: array comprehensions in SAC. Here, *lower_bound* and *upper_bound* denote expressions that must evaluate to integer vectors of equal length. They define a rectangular (generally multidimensional) index set. The identifier *idxvec* represents elements of this set, similar to loop variables in for-loops. However, no order is defined on these index sets, making the WITH-loop a truly data-parallel construct. An index set specification is called a *generator* and it is associated with an arbitrary SAC expression. It creates a mapping between index vectors and values, in other words an array.

```
     int [6 ,7] , int fun (int [6 ,7] A, int [6 ,7] B)
2    {
       tmp = foo ( A);
4      C = with {
             ([3 ,0] <= iv < [6 ,4]) : bar( A, iv );
6            ([0 ,4] <= iv < [6 ,7]) : B[ iv ];
             }: genarray ( [6 ,7] , tmp );
8      return ( C, foo ( C));
     }
```

Fig. 2. Example WITH-loop

simplified form of WITH-loop. Essentially, a WITH-loop maps expressions on a multi-dimensional index space to define the elements of a multi-dimensional array.

Figure 2 shows an example SAC function named `fun` featuring a WITH-loop that defines a 6×7 matrix C using two generators and the default element. Each element of the lower left 3×4 submatrix is defined by the application of function `bar` to the argument array A and the index vector `iv`. The right 6×3 submatrix is "copied" from the corresponding elements of the argument matrix B while all remaining elements, *i.e.* the upper left 3×4 submatrix are defined by the default value `tmp`. Note that the function `fun` has two return values, C and `foo(C)`. We assume that both functions `foo` and `bar` are defined elsewhere in the code. We use a contrived example here to expose most relevant reference counting related features in a relatively short and simple program fragment.

As with any other functional language, automatic memory management is a core feature of SAC. Still, the setting substantially differs from that of most functional languages that are based on algebraic data types. Whereas deeply nested, pointer-interconnected structures made up of large numbers of relatively small entities prevail in main-stream functional languages, SAC programs rather deal with a much smaller number of mostly very large data structures that in turn are either not nested at all or are characterised by a small nesting level. As a consequence, conventional deferred garbage collection techniques are not suitable for SAC, and we use non-deferred reference counting instead. This choice has two essential advantages: Large chunks of memory can be reclaimed as early as possible and not only once heap space is exhausted. Moreover, suitable operations can immediately reuse the memory of argument arrays for storing result arrays. If the elements of a result array are actually identical with those of the reused argument array, any copying of data from argument to result array can be avoided entirely.

Figure 3 shows pseudo C code compiled from the example in Figure 2. We focus on memory management aspects of compiled code and keep the generation of efficiently executable C loop nestings from SAC WITH-loops opaque; the interested reader is referred to [9] for details. The argument arrays A and B carry reference counters that have at least the value 1 as in the calling context of the

```
1    int [6 ,7] , int [6 ,7] fun (int [6 ,7] A, int [6 ,7] B)
     {
3      incrc ( A, 1);

5      tmp = foo ( A);

7      if (getrc (B) == 1) {
         C = B;
9        for (iv = [0 ,0] to [3 ,4]) { C[iv] = tmp;                   }
         for (iv = [3 ,0] to [6 ,4]) { incrc ( A, 1);
11                                     C[iv] = bar ( A, iv); }
       }
13     else {
         C = malloc (...);
15       for (iv = [0 ,0] to [3 ,4]) { C[iv] = tmp;                   }
         for (iv = [3 ,0] to [6 ,4]) { incrc ( A, 1);
17                                     C[iv] = bar ( A, iv); }
         for (iv = [0 ,4] to [6 ,7]) { C[iv] = B[iv];                 }
19     }

21     incrc ( C, 1);
       incrc ( A, −1);
23     incrc ( B, −1);

25     return ( C, foo ( C));
     }
```

Fig. 3. Pseudo C code generated from example in Figure 2

function `fun` the corresponding arrays must appear in argument position. The values can be higher, of course, if the arrays are also referenced elsewhere.

Our reference counting scheme implements a *caller-increments/callee decrements* policy. So, at the end of the computation of `fun` each reference counter must have a value one less than at call time. At compile time we count the number of references of `A` and `B`. `A` appears twice in the body of `fun`, once in the first application of `foo` and again in the WITH-loop. Thus, as `fun` has only received one conceptual reference from the caller (the caller increment), we have to increment the reference counter of `A` by one to cater for the second reference. This is encoded by means of the pseudo operation `incrc`. `B` only appears in the WITH-loop, hence we leave the reference counter as is. Following the caller-increments/callee decrements principle, the reference counter of `A` will be decremented during the evaluation of `foo`.

The SaC compiler generates two code variants for the WITH-loop, one that reuses the argument array `B` for storing the result array `C` and one that allocates fresh memory. The decision which code to execute is taken at runtime by querying the value of `B`'s reference counter (`getrc`). Note that this choice can generally not be made at compile time as the number of references to `B` outside the current

function context is unknown and depends among others on the call site of `foo`. If the reuse is successful, we not only avoid a costly memory allocation, but can also leave out all the code that is merely concerned with copying values from the argument array to the result array. Within the WITH-loop, we need to add code that increments the reference counter of `A` in each iteration because, following our guiding principle, the function `bar` will decrease the reference counter of its argument and we must avoid the premature de-allocation of `A`.

After the reuse conditional, we increment the reference counter of C as we have two occurrences of `C` in the subsequent code. Note that the reference counter of `C` initially will be 1. In the reuse case this is obvious, and in the non-reuse case the reference counter is initialised to this value. In contrast, both the reference counters of `A` and `B` are decremented since these arrays are no longer needed after completion of the WITH-loop. Whether or not they are also de-allocated solely depends on the existence of further references outside the context of `foo`. The interested reader is referred to [13] for a more thorough discussion of reference counting in SAC.

3   The Microgrid and the SVP Concurrency Model

The Microgrid is a customisable many-core chip architecture. It is based on single-issue, in-order RISC cores. Cores are hardware multi-threaded and capable of context switching between threads at each pipeline cycle. A context switch is triggered by any long latency instruction such as memory accesses, floating point operations or synchronisations and, thus, hides the instruction's latency and prevents pipeline stalls. Cores are clustered in rings we call *places* to allow for efficient thread mapping and communication of inter-thread dependencies. Figure 4 shows a block diagram of the Microgrid configuration we used in our experiments.

The Microgrid is actually a hardware implementation of a more general fine-grained concurrency model named SVP for *Self-adaptive Virtual Processor* [14].
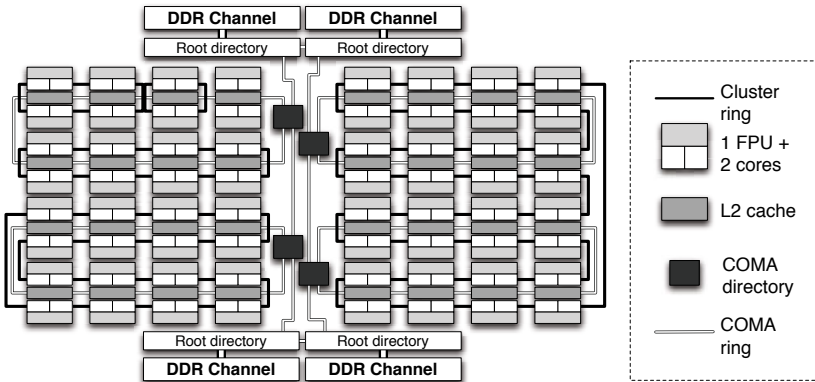

Fig. 4. The layout of a 128 core Microgrid

The SVP model is based on the concept of *thread families*. A family consists of one or more threads that execute the same procedure. Still, each thread has an independent control flow, optionally dependent on a unique index within the family. Any thread can create subordinate families, which take arguments in a similar way as functions do. The arguments to a family of threads are passed on to each thread upon creation and are read-only within the subordinate family. Families of threads can either be created detached or non-detached. In the latter case the parent thread waits for termination of the child family and signals resource reclamation; in the former case the child family terminates independently and resources are released implicitly and immediately.

Furthermore, families of threads can be created at a specific place. This is called *delegation*. The SVP model distinguishes two kinds of places. On *generic places* all families run concurrently up to exhaustion of cores and hardware threads. On *exclusive places* each family runs to completion without interleaving with other families. Cores in the SVP model have access to a distributed memory. While the address space is shared, there is no implicit consistency between memory writes and reads performed by different threads. Consistency is only guaranteed between parent and child threads at the points of creation and synchronisation.

As mentioned before, the Microgrid is effectively a hardware implementation of the SVP concurrency model. On the Microgrid thread family parameters are stored in a hardware family table while information about each individual thread is stored in a hardware thread table. Both are configurable, a typical Microgrid supports up to 32 families and 256 threads per core. If a family of threads is created across multiple cores then a family table entry is allocated on each of the cores. The Microgrid has slightly stronger memory consistency than required by the SVP model. It allows a program to explicitly request system-wide consistency if needed at the expense of increased on-chip network traffic.

The delegation network implements deterministic routing: exclusive families created by a single thread execute in the order they were issued. Similarly, exclusive families issued by a parent thread before a child thread is created are started and complete before any of the child thread's exclusive families. The processing order of exclusive creates issued by unrelated threads is non-deterministic.

Exclusion is negotiated by dedicating a single family context to all exclusive delegations. This way all exclusive delegations are made sequential. A queueing mechanism is required in hardware to avoid having the creating thread wait when the exclusive place is busy.

4   Our Approach: Asynchronous Reference Counting

Our approach for asynchronous non-deferred reference counting is based on two key observations. First, if communicating reference counting state is costly, it may be cheaper to communicate a reference counting operation to the core where the current state of the reference counter resides as opposed to communicating the reference counting state like in classical approaches. On the Microgrid, this

can be achieved with minimal overhead using delegation. The idea is to spawn a dedicated thread for the reference counting operation only and delegate that thread to a reference counting core that is statically assigned to the heap object whose reference counter is to be manipulated. By using exclusive creates, we ensure exclusive access to the reference counting state, efficiently managed in hardware, including the queueing of waiting threads.

Second, although it is important to keep the reference counter accurate during program execution, most threads do not actually require knowledge of the current state of the reference counter. That state is only required for reuse operations, which are relatively infrequent during data-parallel operations. This observation motivates us to perform reference counting operations asynchronously. As we already perform reference counting by separate threads, spawning these threads asynchronously is rather simple. Even more, delegation is ordered between cores on the Microgrid. Thus, even though operations are performed asynchronously, they still remain in order, ensuring a valid, yet slightly delayed reference counting state at all times.

For reuse operations, however, a synchronous approach is required. In this case, the issuing thread is actually interested in the current state of a heap object's reference counter. Thus, the issuing thread has to wait for the reference counting operation to finish. Again, as reuse decisions are relatively infrequent with respect to workload connected to the subject of the decision, such delay can be tolerated, in particular as long as other threads are still ready to compute.

We use a fixed assignment of reference counting places to memory addresses. Each heap object's reference counter is created at its corresponding place and remains there until the object is freed. Furthermore, we use only two operations: An asynchronous `incrc` operation and a synchronous `getrc` operation. The former expects a heap object and an offset as arguments; it asynchronously updates the heap object's reference counter by delegating a thread using a detached create to the exclusive place that is assigned to the heap object. That thread, once having gained exclusive access, increments the heap object's reference counter by the given offset (positive or negative).

Apart from updating a heap object's reference counter, the `incrc` operations also takes care of deallocating no longer needed heap objects. As soon as the reference counter drops to zero, the `incrc` operation notifies the heap manager that the object is no longer needed. This operation, as well, is performed asynchronously to ensure that the reference counting place as soon as possible becomes available again for other pending reference counting operations. For the `getrc` operation we use a similar approach. However, instead of a detached create, we use a synchronous create that allows the issuing thread for the reference counting operation to complete. Furthermore, the `getrc` operation only expects a heap object as argument; it yields the current value of the reference counter of that heap object as its result.

Note here that the returned state is accurate with respect to the inquiring thread's timeline. However, there may still be other pending reference counting requests. Thus, the `getrc` operation might produce *false negatives* in that it

returns a reference counter greater than one, although the object actually is no longer referenced by any other thread. The opposite, a *false positive* where the `getrc` operation returns a reference counting state of one although other threads still access the object can be excluded. As reference counting operations by a single thread are always processed in order, the returned value is accurate with respect to that thread, *i.e.* the inquiring thread holds only a single reference. All other threads then can no longer hold any references, as their local number of references must have dropped to zero. Otherwise, the global reference counting state would need to be at least two.

5   Evaluation

To evaluate our approach, we have first conducted a study using a synthetic benchmark to characterise the scaling behaviour and to quantify the impact of reference counting on runtime behaviour on many-core architectures. Using a synthetic benchmark rather than some real-world computational kernel allows us to study the behaviour of our approach in a controlled setting. However, to show the applicability of our technology outside of the clean room, we include a two dimensional FFT kernel in our experiments.

We have produced our measurements using revision 4196 of the cycle accurate Microgrid simulator and revision 3.2 of the Microgrid toolchain[1]. Our specific platform illustrated in Figure 4 consists of 128 cores, arranged in 8 places of 1, 1, 2, 4, 8, 16, 32 and 64 cores. Each core has split 8K/8K L1 caches for instructions and data, two cores share an FPU and four cores share a 64K unified L2 cache. These are connected to a cache-only memory architecture (cache lines migrated to point of use), with 4 directories and 4 DDR channels to backing store. Each core supports up to 256 hardware threads. Timings are scaled to simulate 1.2GHz cores and DDR3-2400 channels.

We have recorded full traces of the processor states during simulation and have post-processed these traces to compute pipeline utilisation and resource usage. The benchmark code itself was compiled using the Microgrid back-end of the SaC research compiler `sac2c` revision 17128[2]. For concurrent execution, we have sampled the state of the Microgrid only for the runtime of the relevant data-parallel operation(s). For dedicated sequential execution, we present whole program figures as the standard back-end of the SaC compiler does not support this feature. This difference, however, does not affect the validity of our results.

5.1   Synthetic Kernel

Our synthetic kernel is shown in Figure 5. Given an input vector vect, the WITH-loop computes a new vector of the same length. Each element of the result is

---

[1] The Microgrid simulator and toolchain are available on request from the CSA group at the Institute for Informatics of the University of Amsterdam.

[2] The SaC research compiler `sac2c` is freely available for non-commercial use from http://www.sac-home.org.

```
      int work ( int i , int [.] vect )
2     {
        r = 0;
4       for ( j =0; j<vect [[ i ]]; j++) {
          r = r +1;
6       }
        return ( r );
8     }

10    result = with {
                ([0] <= [ i ] <= shape ( vect )): work (i ,vect );
12              }: genarray ( shape ( vect ), 0);
```

Fig. 5. Source code of the synthetic benchmark used for evaluating non-deferred reference counting on the Microgrid

computed by concurrently applying the function work to the current index i and the input vector vect. The function work is given in lines 1–8; it encodes a selection of the i-th element from the argument vector vect. In order to model different workloads, selection is implemented by means of a for-loop that consecutively increments a counter, starting at zero, until the value at the i-th position in vect is reached. Thus, the total runtime of work is largely determined by the values in the vector vect.

From a reference counting perspective, the above benchmark encodes a worst case scenario. All threads created due to the WITH-loop first emit two reference counting operations: The caller increment issued before the application of work and the callee decrement issued directly after the read from vect in line 4. Note here that in SaC other than in C there is only a single read operation from vect. Due to the purely functional semantics of SaC, it is a valid optimisation to store the read operation's result in a local variable, which is then used for consecutive checks of the termination condition of the for-loop.

We have designed our synthetic kernel with two tunable parameters: tuning the length of the input vector vect allows us to influence the number of created threads and thus the number of reference counting operations emitted. As each thread contains one call to work with vect as argument, we get two reference counting operations per thread. As the first argument of work, the index i, is a scalar, it is not heap allocated and thus not reference counted.

The second tunable parameter is the value used for the elements of vect. Each thread performs three operations per loop iteration. Thus, the overall workload per thread can be computed as roughly three times the value of the elements of vect. For our experiments, we use a single number for all vector elements, thereby encoding a uniform workload.

In our program, the main computation only results in a single thread family. Hence, the family table can be kept small at the computing place. Due to a current limitation of the architecture, which prevents asynchronous queueing of

detached families, we use two-level creates to implement asynchronous creates at exclusive places. In this scheme the work thread detaches a thread at a non-exclusive proxy place; the detached thread in turn issues a synchronous create at an exclusive place. We thus simulate queueing in software by allowing multiple families at the proxy place to wait simultaneously for the exclusive place. This is less efficient than hardware queueing, but offers an advantage for our evaluation: the number of active threads at the proxy place indicates the number of pending requests to the exclusive place. With this scheme, we can easily tune the number of family entries at the proxy place used for reference counting to compensate for contention. To benchmark our program we use 256 family entries per core.

Figures 6 and 7 show the results for a vector of 512 elements and a value of 100. This corresponds to a total of 1024 reference counting operations and a workload of about 150k instructions. The left hand side of both figures presents the results for the executables as produced by the SaC compiler. On the right hand side, we have repeated the same measurements with hand patched executables where all reference counting operations have been removed.

We have first measured runtime and pipeline efficiency for a fully sequential version of the benchmark that does not expose any concurrency. Furthermore, we have used classical reference counting by direct manipulation of a reference counter in memory for these measurements. As direct comparison of Figure 6a and Figure 6b shows, reference counting in the sequential case introduces an overhead of about 200k cycles. Apart from the actual cost of the reference counting operations, we mainly attribute this overhead to the reduced pipeline efficiency. The Microgrid does not feature sophisticated branch predictors or memory pre-fetching stages. Instead, it relies on concurrency to hide the latencies of branches and memory loads.

As expected, adding concurrency and our asynchronous approach to reference counting therefore leads to improved runtime behaviour even on a single core. Figures 6c and 6d both show significant improvement over their sequential counterparts. By offloading reference counting to a dedicated asynchronous core, we have reduced the incurred overhead to less than 50k cycles.

An interesting artefact is the relatively low pipeline utilisation at the beginning of the data-parallel section when reference counting is enabled. We attribute this to queueing effects. Each thread has to successfully enqueue two reference counting operations before it can start computing its workload. It seems that due to the scheduling chosen by the architecture most threads only manage to enqueue their first request before the request queue is full. Thus, threads initially have to wait for reference counting operations to complete.

The version of our benchmark without reference counting exhibits almost linear scaling with increasing numbers of cores as Figures 6f to 7h show. For the runtimes with reference counting enabled, as shown on the left hand side, however, scaling is less favourable and it hits a limit at about 4 cores. Furthermore, we can observe that the number of pending reference counting operations (the dotted lines in the figure) increases with the number of cores. This effect culminates in Figure 7a where for the first time the reference counting queue remains
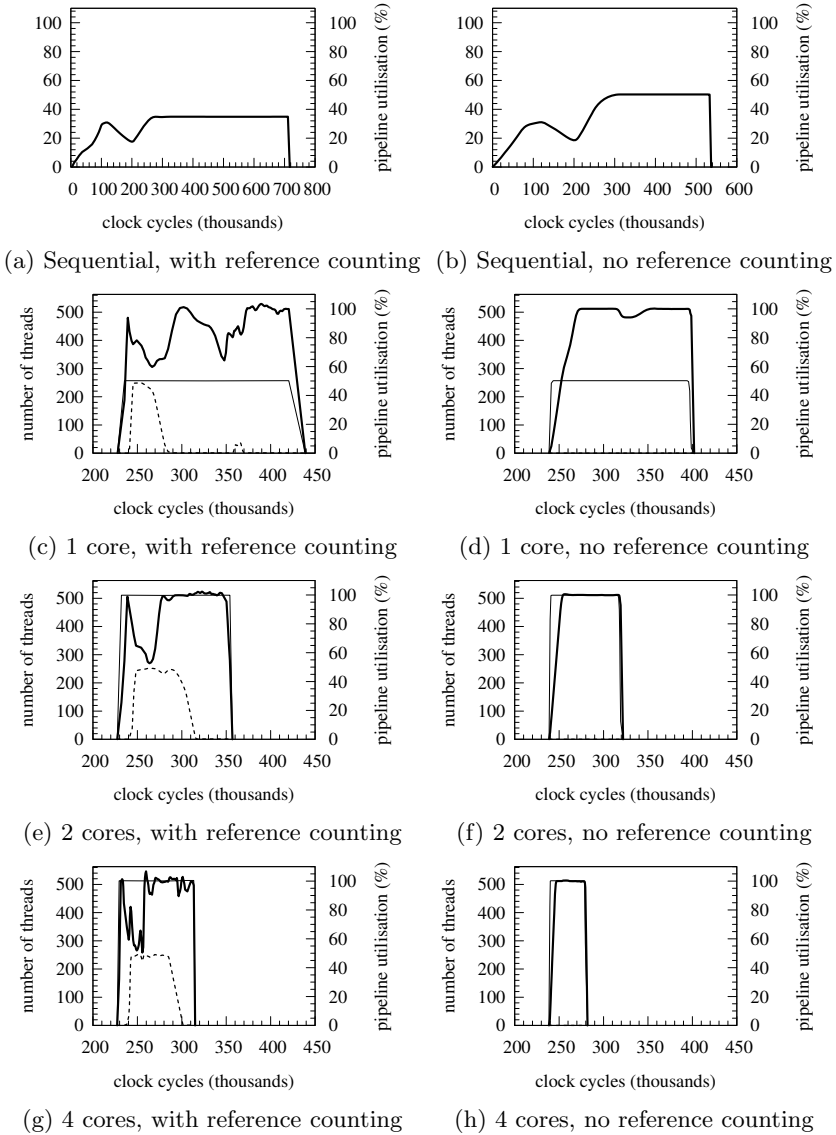
(a) Sequential, with reference counting

(b) Sequential, no reference counting

(c) 1 core, with reference counting

(d) 1 core, no reference counting

(e) 2 cores, with reference counting

(f) 2 cores, no reference counting

(g) 4 cores, with reference counting

(h) 4 cores, no reference counting

Fig. 6. Benchmark results for a vector of 512 elements and a workload of 100 additions. Thin lines show the number of threads computing the actual workload; dotted lines represent the number of pending reference counting operations; thick lines give the average pipeline utilisation across cores that compute the workload. We start with fully sequential code and then continue with parallel code for 1, 2 and 4 cores. In the left column we show results for the code generated by our compiler, in the right column for code where we manually removed all reference counting operations. We continue this in Figure 7 for larger numbers of cores.
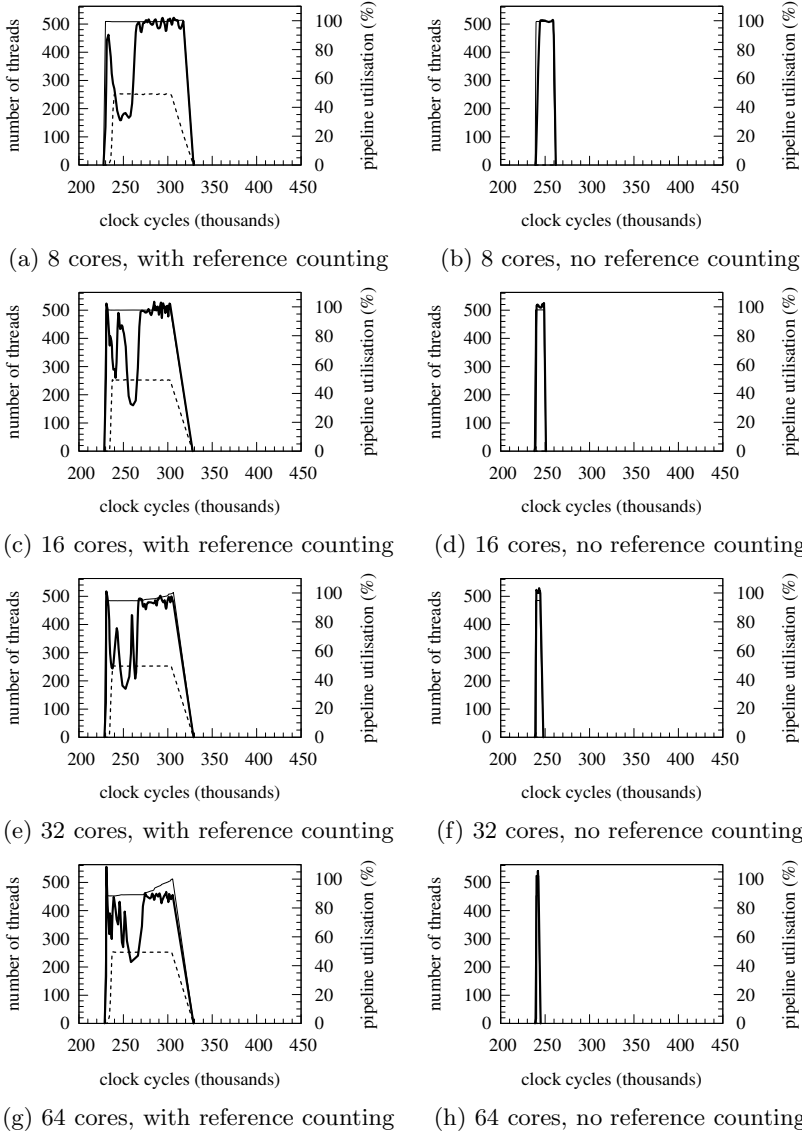
(a) 8 cores, with reference counting

(b) 8 cores, no reference counting

(c) 16 cores, with reference counting

(d) 16 cores, no reference counting

(e) 32 cores, with reference counting

(f) 32 cores, no reference counting

(g) 64 cores, with reference counting

(h) 64 cores, no reference counting

Fig. 7. Benchmark results for a vector of 512 elements and a workload of 100 additions. Thin lines show the number of threads computing the actual workload; dotted lines represent the number of pending reference counting operations; thick lines give the average pipeline utilisation across cores that compute the workload. Continuing from Figure 6 we show results for using 8, 16, 32 and 64 cores. In the left column we show results for the code generated by our compiler, in the right column for code where we manually removed all reference counting operations.

fully loaded during the entire data-parallel section; the reference counting operations dominate the overall execution. This is a consequence of the necessity to perform all reference counting operations sequentially. Our software implementation of the asynchronous reference counting operations requires roughly 100 cycles each leading to 100k cycles in total. This observation enables us to predict the best possible speedup by means of Amdahl's law: it equates to the ratio between workload and reference counting time which, in our example, are 300 and 100 cycles per element, respectively.

This ratio of 1.5 in fact is the limiting factor for the speedups observed as shown in Figure 8. To confirm our explanation, we repeated the same experiment for larger vectors and with varying ratios between workload and reference counting times. Figure 8 shows three different experiments in total. Besides our initial experiment, it also contains an experiment on a 4096-element vector with the same ratio of 1.5, and an experiment with a ratio of 15 (1000 iterations).
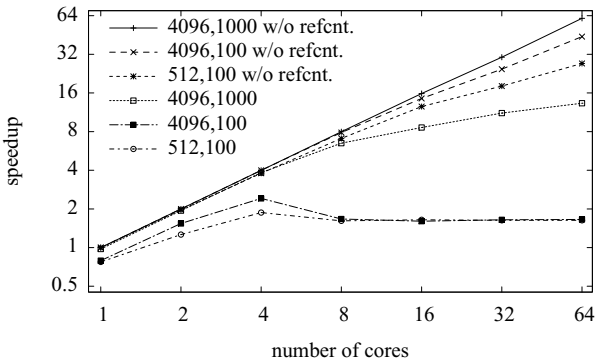

Fig. 8. Observed speed up for varying numbers of threads and workload sizes compared to the runtimes on a single core with reference counting operations removed

A first observation is that increasing the number of elements and, thus, the overall load has no impact on speedups even though it entails an eightfold increase in the number of threads created. Increasing the number of iterations to 1000, and with it the ratio between workload and reference counting time to a factor of roughly 15, directly impacts scaling. Even more, we can see that our predicted speedup factor is reached when using 64 cores. This demonstrates nicely that the architecture is capable of hiding all the workload (roughly 12M cycles) behind the sequential program fragment due to reference counting (roughly 800k cycles). For completeness, we have included the results for running the benchmark with reference counting operations removed, as well. For 100 iterations we observe close to linear scaling while 1000 iterations scale perfectly linear.

5.2 2-Dimensional FFT

As a representative of a real-world computational kernel, we chose 2-dimensional FFT. In fact, our code is a stripped-down version of the NAS benchmark FT [1],
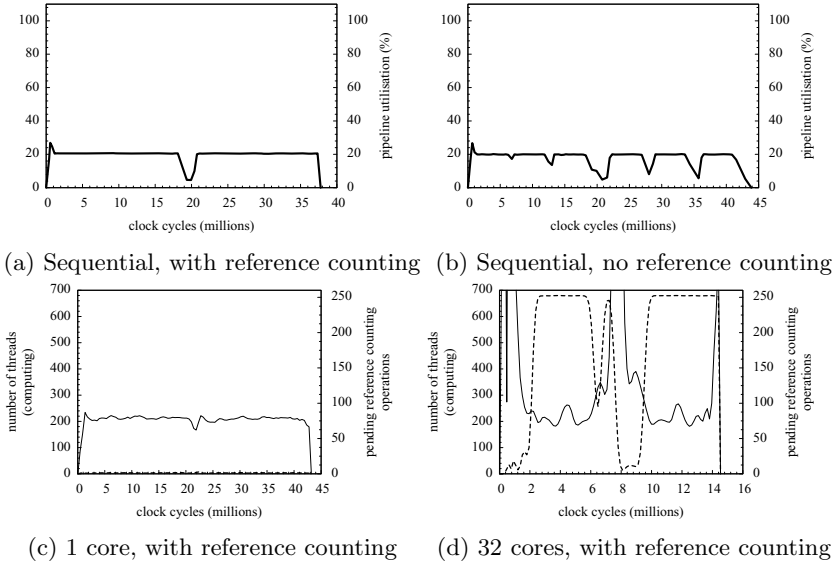
(a) Sequential, with reference counting (b) Sequential, no reference counting

(c) 1 core, with reference counting (d) 32 cores, with reference counting

Fig. 9. Results for a two dimensional FFT on a 128×128 complex matrix

which implements 3-dimensional FFT. We restrict ourselves here to the 2-dimensional case and a relatively small problem size due to the computational complexity of cycle-accurate simulation of the Microgrid. In essence, our kernel transforms a matrix of complex numbers by applying 1-dimensional FFTs to each row vector, then transposing the result matrix and again applying 1-dimensional FFTs to each row vector, i.e. the former column vectors. We implement the 1-dimensional FFTs using the Danielson-Lanczos algorithm, which recursively decomposes the argument vector into the vectors of even- and odd-indexed elements. A detailed discussion of the SAC implementation of this benchmark can be found in [7].

Figure 9 shows our experimental results for running the 2d-FFT kernel on a matrix of 128×128 double precision complex numbers. The baseline performance for fully sequential execution with classical reference counting is given in Figure 9a. The three phase nature of our implementation can be nicely observed in the pipeline utilisation graph. During the first phase, the computation of FFT on the rows of the input, we achieve an efficiency of just above 20%. This is followed by a short phase, the transpose operation, with efficiency dropping below 10% before it goes back to 20% for the second round of FFT on the columns. The low pipeline utilisation is to be expected as FFT and to an even larger degree transpose operations are memory bound.

Note here that the fully sequential version without reference counting is actually slower than the version with reference counting. We attribute this to the unexploited memory reuse potential when reference counting is disabled. The

same effect can be observed across 2, 4, 8, 16, 32 and 64 cores. An implementation using our concurrent reference counting scheme consistently outperforms the version without any form of reference counting. We omit the details here due to space limitations.

Figure 9c shows the runtime behaviour of a concurrent implementation running on a single core. The observed reference counting behaviour greatly differs from our synthetic benchmark. Instead of a high initial reference counting load that tails off during program runtime, we observe continuous, yet low frequency reference counting throughout the runtime of the benchmark.

The changed pattern relates well to the different reference counting distribution in FFT. Whereas our synthetic benchmark first issues all reference counting operations and then computes the workload, FFT starts with an increment of the argument due to the initial function call but then immediately processes some workload before further reference counting operations are emitted. Thus, reference counting operations and the computation of the actual workload are better interleaved, resulting in a lower pressure on the reference counting queue.

As expected, the pressure on the queue grows with increasing numbers of cores until a maximum is reached at 32 cores, shown in Figure 9d. At this stage, we can observe a constant reference counting load and further scaling becomes constrained by Amdahl's law. This finding matches our previous experience with the synthetic benchmark. As before, the architecture is able to hide the 45 million cycles of workload in 14 million cycles of reference counting operations.

6   Related Work

Although we are aware of recent work on using non-deferred reference counting in the context of object-oriented languages [15], we did not come across any work of non-deferred reference counting in the context of distributed shared memory systems. However, the underlying principles of our approach, *i.e.* shipping computation to data and exploiting asynchronous communication for latency hiding, have been applied to related problems in distributed systems before.

One example in this setting is the multi kernel paradigm adopted by the Barrelfish operating system [2] for multi- and many-core systems. In Barrelfish, instead of using a single global kernel and shared state, the operating system is built around a communicating network of kernels. Each computing resource is managed by its own kernel and state is replicated using message passing. Similar to our approach, operating system services are delegated to responsible cores that hold the corresponding state instead of communicating the state. The motivation here, like for us, is scalability.

Similar, but on a significantly larger scale, distributed file systems have to contend with typically large objects (files) replicated in storage across several applications (clients). Usually, metadata and directories are maintained separately from the data, with tables that keep track of which clients currently hold a copy of each file. Storage reclamation after path deletion can only occur when the last client has dropped its replica of the corresponding file. The Hadoop distributed file system [18] and the Google File System [6] are particular examples of distributed

file systems that employ a scheme closely related to our approach. In both, objects are file data blocks and are distributed across a set of *data nodes*. Separate from these, *name nodes* hold the metadata and reference information. When data nodes duplicate data or create new data they must inform the name node of the existence of new copies through *heartbeat messages*. Client applications can enquire through a name node to know how many copies of a data block exist. On each name node, heartbeats are handled in order but asynchronously, except when an application requests a *flush-and-sync* of pending heartbeats. This is similar to our asynchronous updates/synchronous read scheme.

7   Conclusion

We have presented a novel approach for concurrent non-deferred reference counting on many-core architectures. Instead of locks and exclusive regions, we employ exclusive processing units for reference counting. We communicate the reference counting operation to the core where the associated state is stored rather than communicating state between cores. In a many-core setting, this greatly reduces reference counting related overheads. Furthermore, we use asynchronous communication where possible to hide the latencies involved.

As a first evaluation shows, our approach is able to tolerate even worst-case reference counting scenarios. The scaling behaviour of our synthetic benchmark is dominated by the combined sequential runtime of the inflicted reference counting operations. According to Amdahl's law, this is the best possible behaviour we can expect as reference counting operations must be performed sequentially. Nonetheless, there is room for improvement. The current implementation of our approach encodes the asynchronous communication protocol between cores in software. For a purely hardware based solution, we expect the sequential runtimes of reference counting operations to be reduced by at least a factor of 4. This would directly reflect in a four-fold increase in the expected maximum speedups.

Yet, ultimately reference counting remains the bottleneck. In particular for data-parallel operations as investigated here, the bursts of reference counting operations typical for SPMD style code may dominate runtime behaviour. Our future research in this context, therefore, concentrates on further reducing the number of reference counting operations on shared data in data-parallel codes.

We believe our approach is well suited for task-parallelism, as well. The less structured interleaving of reference counting operations and workloads found in task-parallel applications should allow for even better exploitation of asynchronism for hiding reference counting overheads. However, an extension of our approach to support less structured settings remains future work.

References

1. Bailey, D., et al.: The NAS Parallel Benchmarks. International Journal of Super-computer Applications 5(3), 63–73 (1991)
2. Baumann, A., Barham, P., Dagand, P.E., et al.: The multikernel: a new OS architecture for scalable multicore systems. In: 22nd Symposium on Operating Systems Principles (SOSP 2009), pp. 29–44. ACM, New York (2009)
3. Bousias, K., Guang, L., Jesshope, C., Lankamp, M.: Implementation and Evaluation of a Microthread Architecture. J. Systems Architecture 55(3), 149–161 (2009)
4. Collins, G.E.: A Method for Overlapping and Erasure of Lists. Communications of the ACM 3(12), 655–657 (1960)
5. Doligez, D., Leroy, X.: A Concurrent, Generational Garbage Collector for a Multithreaded Implementation of ML. In: POPL 1993: 20th Symposium on Principles of Programming Languages, pp. 113–123. ACM, New York (1993)
6. Ghemawat, S., Gobioff, H., Leung, S.T.: The Google file system. SIGOPS Oper. Syst. Rev. 37(5), 29–43 (2003)
7. Grelck, C., Scholz, S.B.: Towards an Efficient Functional Implementation of the NAS Benchmark FT. In: Malyshkin, V.E. (ed.) PaCT 2003. LNCS, vol. 2763, pp. 230–235. Springer, Heidelberg (2003)
8. Grelck, C.: Shared memory multiprocessor support for functional array processing in SAC. Journal of Functional Programming 15(3), 353–401 (2005)
9. Grelck, C., Kreye, D., Scholz, S.B.: On Code Generation for Multi-Generator WITH-Loops in SAC. In: Koopman, P., Clack, C. (eds.) IFL 1999. LNCS, vol. 1868, pp. 77–94. Springer, Heidelberg (2000)
10. Grelck, C., Scholz, S.B.: Merging compositions of array skeletons in SAC. Journal of Parallel Computing 32(7+8), 507–522 (2006)
11. Grelck, C., Scholz, S.B.: SAC: A Functional Array Language for Efficient Multithreaded Execution. Int. Journal of Parallel Programming 34(4), 383–427 (2006)
12. Grelck, C., Scholz, S.B.: Efficient Heap Management for Declarative Data Parallel Programming on Multicores. In: 3rd Workshop on Declarative Aspects of Multicore Programming (DAMP 2008), San Francisco, USA, pp. 17–31. ACM Press, New York (2008)
13. Grelck, C., Trojahner, K.: Implicit Memory Management for SaC. In: Grelck, C., Huch, F. (eds.) IFL 2004, pp. 335–348 (2004); University of Kiel, Institute of Computer Science and Applied Mathematics technical report 0408
14. Jesshope, C.: A model for the design and programming of multi-cores. Advances in Parallel Computing, High Performance Computing and Grids in Action (16), 37–55 (2008)
15. Joisha, P.G.: A principled approach to nondeferred reference-counting garbage collection. In: 4th International Conference on Virtual Execution Environments (VEE 2008), pp. 131–140. ACM, New York (2008)
16. Marlow, S., Harris, T., James, R.P., Peyton Jones, S.: Parallel Generational-Copying Garbage Collection with a Block-Structured Heap. In: ISMM 2008: 7th International Symposium on Memory Management, pp. 11–20. ACM, New York (2008)
17. Scholz, S.B.: Single Assignment C: Efficient Support for High-Level Array Operations in a Functional Setting. J. Functional Programming 13(6), 1005–1059 (2003)
18. Shvachko, K., Kuang, H., Radia, S., Chansler, R.: The Hadoop distributed file system. In: 26th Symposium on Massive Storage Systems and Technologies (MSST 2010). IEEE Press, Incline Village, USA (May 2010)