Lazy Reference Counting for the Microgrid

Raphael Poss, Clemens Grelck
University of Amsterdam
Amsterdam, NL
{r.c.poss,c.grelk}@uva.nl

Stephan Herhut
Intel Corporation
Santa Clara, CA, USA
stephan.a.herhut@intel.com

Sven-Bodo Scholz
Heriot-Watt University
Edinburgh, UK
s.scholz@hw.ac.uk

Abstract

This papers revisits non-deferred reference counting, a common technique to ensure that potentially shared large heap objects can be reused safely when they are both input and output to computations. Traditionally, thread-safe reference counting exploit implicit memory-based communication of counter data and require means to achieve a globally consistent memory state, either using barriers or locks. Acknowledgeing the distributed nature of upcoming many-core chips, we have developed a novel approach that keeps reference counters at single physical locations and ships the counting operations asynchronously to these locations using hardware primitives, rather than implicitely moving the counter data between threads. Compared to previous methods, our approach does not require full cache coherency.

1. Introduction

All modern programming languages are in one way or another based on the concept of automatic heap management, *i.e.* the implicit reclamation of heap memory that is no longer needed by a running program. The lesson learnt from C/C++ style programming with explicit memory management was that the programmer almost inevitably does it wrong: either memory is reclaimed too early leading to inconsistent program states or too late resulting in space leaks that continuously decrease the effectively available heap memory during the runtime of a program. Functional languages led this development and have always been using automatic heap management, but since the advent of Java in the 90s the concept has quickly gained popularity among imperative and object-oriented programmers alike. And the same story continued with the scripting languages of the previous decade and today.

The prevailing deferred garbage collection techniques have been very much refined since the early days (for a survey see [16]). Nevertheless, common properties have hardly changed. The interface between the application and the heap management is typically concise. In one way or another memory of a certain size can be requested from the underlying heap manager. Allocation is often implemented by simply advancing a high water mark pointer. Only if there is no sufficient memory left, the memory reclamation machinery becomes active. It identifies live data structures (ojects) and releases dead objects for further allocations.

One disadvantage of deferred garbage collectors is that a program cannot determine whether an object is still referenced elsewhere by just looking at it. This rules out reuse of input objects for output in languages with single assignment semantics, a common case in the functional community. Cases where non-sharing can be proven statically are rather rare in practice; typically, the question is undecidable in all but the most simple codes. As a consequence, whenever some object is derived from an existing one, leaving potentially large parts untouched, a fresh object must always be allocated in memory and the unchanged parts copied.

Although this issue has limited impact in languages with small allocation units (*e.g.* LISP cells), it becomes more significant in languages that manipulate a smaller number of large objects. With arrays for example, any functionally sound *update* operation that defines a new array as being identical to an existing one with the exception of few recomputed elements, becomes intolerably expensive. What would be a constant time operation in an imperative environment, becomes linear in the size of the array.

This well-known *aggregate update problem* [10] is one instance where *non-deferred* garbage collection using reference counting (RC) is necessary. We use the functional array language Single Assignment C (SAC [7]) as an example. SAC uses non-deferred RC to determine reusability of arrays (section 2). In this scenario, arrays are equipped with a reference counter updated throughout the array's lifetime. Althout the allocator must then also deal with heap fragmentation, under the assumption of reasonably sized arrays and few allocations cycles the additional runtime overhead in space and time is negligible. The key advantage, however, is that the space of argument values can directly be reused to accommodate result values since the runtime

system can always query the reference counter and decide whether the data is currently referenced elsewhere. This feature is intensively used by the SAC memory management subsystem [8] and has proven to be essential to competitive performance in functional array programming.

Today's prevalence of many-core processors raises new issues. Requests to modify a reference counter may come from concurrent threads. The same applies to queries for a counter's value at any memory reuse opportunity at runtime. In a traditional shared memory system this require cache coherency, fences and mutual exclusion or atomics. The corresponding hardware support is unlikely to scale with increasing core counts on chip. Instead, larger core counts will cause the advent of on-chip memory architectures that are either fully distributed or show different degrees of consistency from the perspectives of individual cores. Our target Microgrid architecture, introduced in section 3, illustrates this view.

To target these upcoming systems, we propose *lazy reference counting*, a scalable implementation technique for non-deferred RC on many-core architectures using weaker consistency semantics. Traditional RC in a cache coherent system requires to move counter data spatially to the location where the need to update or query the counter arises. We propose instead to keep the reference counter in the same chip location during its lifetime. Then rather than updating the counter itself, a task located spatially elsewhere sends asynchronously a request to the home location of the counter. This location in turn sequentialises incoming requests appropriately and performs object reclamation accordingly. A key aspect for scalability is that tasks can execute asynchronously with the RC operations and do not wait for the remote RC operations to complete.

As we explain in section 4, the price that we are willing to pay for this is that the reference counter state only approximates the number of actual references. There is a time delay between the conceptual update of the reference counter from the perspective of the running program and the effective update of the counter in its home location. We think this is acceptable because we are not interested in tracking the exact number of conceptual copies of some object during runtime, but merely in the question whether or not the object can be updated destructively. For example, we care whether the reference counter is 1 or 2, but whether it is 2 or 42 is entirely irrelevant for memory reclamation and reuse.

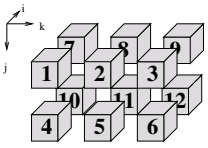
We have specifically developed the proposed technique for compiling SAC to the Microgrid many-core architecture [6]. As we discuss in section 5, the proposed approach imposes extra requirements on the network routing protocol, highlights issues of over-synchronization in the current implementation of the Microgrid and guided the design of architectural solutions. This makes our approach also a rare example of crosslayer design integration.

2. SAC as a use case for reference counting

SAC is a functional, side-effect free variant of C: assignment sequences are interpreted as nested let-expressions, branches as conditional expressions and loops as syntactic sugar for tail-end recursive functions; details can be found in [7]. Despite the radically different underlying execution model (context-free substitution of expressions *vs.* step-wise manipulation of global state), language constructs adopted from C show exactly the same operational behaviour as expected by imperative programmers. This allows programmers to choose their favourite interpretation of SAC code while the compiler exploits side-effect free semantics to better optimize and parallelize evaluation.

2.1. Truly multidimensional arrays

SAC provides native support for truly multidimensional and stateless/functional arrays using a shape-generic style of programming (Figure 1). Any SAC expression evaluates to an array. Arrays may be passed between functions without restrictions. Array primitives can access either array metadata, *e.g.* an array's rank (`dim(`*array*`)`), or individual elements or entire subarrays using the usual square bracket notation: *array*`[`*idxvec*`]`.



dim: 3
shape: [2,2,3]
data: [1,2,3,4,5,6,7,8,9,10,11,12]

42

dim: 0
shape: [ ]
data: [42]

Figure 1. Multidimensional arrays in SAC.

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

Figure 2. Array comprehensions in SAC.

2.2. WITH-loops

All aggregate array operations are specified using WITH-loop expressions, a SAC-specific array comprehension shown in Figure 2. 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. 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.

Figure 3 shows the example of a WITH-loop that defines a 5×6 matrix B using two generators and the default element. Each element of the lower left 3×4 submatrix is defined as the sum of the two elements of the index vector. The right 5×2 submatrix is "copied" from the existing matrix A. The remaining elements, *i.e.* the upper left 2×4 submatrix, are implicitly defined by the default element.

```
B = with {  ([3,0]  <= iv  <  [5,4])  :  iv[0]  +  iv[1];
            ([0,4]  <= iv  <  [5,6])  :  A[iv];    }: genarray(  [5,6],   0)
```

$$B \leftarrow \begin{pmatrix} 0 & 0 & 0 & 0 & A[0,4] & A[0,5] \\ 0 & 0 & 0 & 0 & A[1,4] & A[1,5] \\ 0 & 0 & 0 & 0 & A[2,4] & A[2,5] \\ 3 & 4 & 5 & 6 & A[3,4] & A[3,5] \\ 4 & 5 & 6 & 7 & A[4,4] & A[4,5] \end{pmatrix}$$

Figure 3. Example WITH-loop and result.

2.3. Memory management

We focus here on memory management when compiling WITH-loops to executable code. Assuming that the existing array A in Figure 3 is of the same shape as B, *i.e.* 6×7 elements, A is a *reuse candidate* for B. Whether or not this candiate may actually be reused is only decidable at runtime from its associated reference counter.

If its reference counter is 1, the reference to it is the last one, and the memory can be safely updated destructively. This saves one loop nesting, the one derived from the second generator. In case the candidate cannot be reused, the default code allocates fresh memory to accommodate the new array B and uses three loop nests to initialise all elements from scratch. Finally, A's counter is decremented. Regardless of reuse, B's counter is set to the number of statically inferred references in the rest of the current code block.

The example makes the benefits of memory reuse obvious. Not only does this save a costly invocation of the memory allocator, but it also allows to skip all code that is merely concerned with copying data from an existing to the new array. The interested reader is referred to [8] for a more thorough treatment of the subject.

3. Our target: SVP and the Microgrid

The System Virtualization Platform (SVP) is a machine model and low-level programming interface for the exploitation of general-purpose many-core chips with combinations of shared and distributed memories [12].

The basic construct for exposing concurrency is *bulk creation* of multiple *logical threads* at once. This declares a concurrent unit of work, called a *family*, which the platform can either execute in parallel, sequentially, or a combination of both. A creating thread can then *bulk synchronize* on termination of a family. This fork-join pattern captures concurrency hierarchically, from software component composition down to inner loops. Although an implementation may offer more flexibility, SVP encourages forward-only dataflow compositions so that programs stay deadlock-free and serialization is always possible.

SVP further separates the declaration of concurrency from its scheduling. When code expresses more concurrency than is available, processors automatically switch from space scheduling to time scheduling at the point when all concurrent resources become full. Excess concurrency is automatically removed to prevent overheads. This avoids granularity mismatches on heterogeneous resources.

Space scheduling is further achieved by binding a collection of processors, called a *place*, to a family upon its creation. This can happen at any level, dynamically. Creations can either propagate the current mapping to the child families ("default" place), or restrict to a smaller entity ("local" place). This provides an efficient way to control locality at fine granularities. Creation can also use explicit places provided at a coarse level by a *place allocator* which leases entire clusters to requesting program components [13].

3.1. An SVP implementation: the Microgrid

The Microgrid is a many-core chip architecture which implements the SVP interfaces in hardware [11, 3]. Dedicated circuits next to a RISC pipeline provide bulk creation, synchronization and inter-thread communication. This is in turn controlled by dedicated instructions in the ISA.

For work distribution, programs designate either single cores or clusters of multiple cores as a place, using a uniform addressing scheme which preserves cache locality at any cluster size (Figure 4). Within a cluster, bulk creation and synchronization use a fast *local distribution network*, whereas across clusters a narrow mesh *delegation network* is used for remote family control. This is physically separated from the memory network to avoid interference. The memory network makes caches coherent at family creation and synchronization points only, resulting in weak consistency between sibling threads in a family.

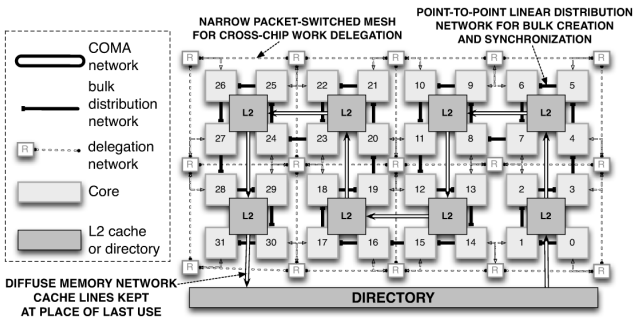


Figure 4. Example 32-core Microgrid tile.

3.2. Exclusive places for shared state

The memory system of the Microgrid chip only provides bulk concistency at family creation and termination. This is efficiently implemented in hardware by flushing pending memory stores at family events. However, this comes at the cost of disallowing implicit memory-based communication between sibling threads that are unrelated via parent-child synchronization, *including all manners of reference counting and mutual exclusion via memory-based structures*.

For cases where bounded non-determinism is desirable in programs, SVP also provides *exclusive places*. These provide the following semantics: all families created at the same exclusive place are scheduled in *some sequential order* with regards to each other, *i.e.* they are guaranteed to run mutually exclusively. To implement this, an **exclusive allocate** request defines the requirement for sequential scheduling. When an allocation requests arrives at a core with the exclusive flag, the hardware create process waits until previous exclusive work has terminated before satisfying the request. The memory system guarantees that writes performed at an exclusive place are visible by subsequent families running at the same place. This way, atomic accesses to shared state can be sent to the same exclusive place. This corresponds to the "secretary" concept of [4]; compared to criticial sections and monitors found in other programming models, exclusive places have an explicit location on chip.

3.3. Compiling SAC to the Microgrid

We have implemented a compiler from SAC to the Microgrid as previous work [6], which we summarize here. There are two forms of concurrency in SAC: functional concurrency in independent expressions and data-parallelism in WITH-loops. Function calls can be translated to SVP families of one thread. This way execution of functions can be parallel if there are resources available, otherwise they are sequentialized automatically. This can be applied recursively. Sharing of arrays happens when the same array is provided to two or more concurrent function calls. Data parallelism in WITH-loops is mapped onto wide SVP families. Any multi-level WITH-loop is compiled into multiple thread programs. When the WITH-loop is reached during execution, this triggers the creation of a tree of families, one level per generator dimension. Hints for SVP "default" and "local" placement are also used to optimize locality of accesses. When possible, code to reuse or allocate memory for the result is placed before a top-level family creation.

4. Our solution: distributed lazy RC

Reference counting in SAC is straightforward and relies on the usual operations: **setrc** upon initial allocation, **incrc**, **decrc** (hereafter considered to be **incrc** with a negative value) to update the value of a reference counter, and **getrc** to enquire its current value. In traditional shared memory systems, all these operations but the first run potentially concurrently and must be implemented using atomic primitives or locks. The low-level requirement are then a cache protocol and memory locking semantics that ensure that the data for the reference counter is globally consistent across the entire memory network. This implies non-trivial synchronization latencies and interference with computation-related memory network traffic.

4.1 Distributed reference counting

The key idea in our approach is to *ship the individual reference counting operations to a computing resource* that holds the counter data, using low-latency fine-grained hardware primitives, instead of shipping the counter to the computing resource where the RC occurs. The immediate advantage in a setting with distributed memory is the reduced bandwidth requirement. Furthermore, by keeping the reference counters in a single location, our approach works with weaker consistency systems such as the Microgrid.

In a first approach, **incrc** and **getrc** are mapped to SVP thread programs. **incrc** takes the address of a reference counter, or a heap object containing a reference counter at a predefined offset, and an increment as arguments, and updates the value of the counter. **getrc** takes the address of a reference counter as argument and returns its current value.

From a compiler's perspective these are the same primitives as used in a non-concurrent setting. The running program, however, now *sends requests to a shared processing resource* instead of performing the RC operations directly. In SVP this resource is an exclusive place (*cf.* subsection 3.2). It sequentialises all incoming RC requests and performs only one RC operation per object at a time. This further requires preservation of creation order across the network: RC operations issued by one thread need to be performed in their issue order; also, if one thread

spawns other threads, all RC operations of the parent thread up to the spawn operation need complete before the RC operations of any child thread are performed. This is to ensure that the caller increments are performed before any callee decrements of the children are enacted.

We further do not require a single exclusive place for all RC operations; we merely require that a single counter is managed by the same exclusive place throughout its lifetime. This means that *each reference counter can be managed by a different place*. The scheme is thus scalable with the number of exclusive places used for RC, up to one per core on our Microgrid implementation. Furthermore, on the Microgrid this mapping is efficient: using a primitive hash function, the address of any reference counter can be transformed into a valid exclusive place identifier in a way that balances load across the entire grid.

4.2 Lazy reference counting

A further key observation is that for most RC operations, the requesting thread is not interested in the result of the operation. The **incrc** requests are issued by a thread to create a conceptual new reference or to release a conceptual reference. As long as either operation is performed eventually and thus freeing of the memory prevented or facilitated, it does not matter when the operation is actually performed.

Given that SVP separates *initiation* of a concurrent activity (through the issuing of a **create** operation) and the actual execution of the corresponding instructions (when the scheduling of threads eventually happens), we can also issue **incrc** asynchronously and let them execute concurrently with the computation until the synchronization required by the observation operations. This overlap provides extra latency hiding. Only at the point of observation, the thread requesting **getrc** requires synchronization as it needs to decide a potential memory reuse. In short, pending asynchronous requests must complete only before **getrc**.

This is what we name *lazy reference counting*: we can postpone RC operations, from the issuing threads' perspective, until a point of potential reuse is reached. This leads to a heap management that is partially deferred, yet provides low-overhead liveness information. Furthermore, with sufficient hardware resources, we can perform RC concurrently with the program's execution.

4.3 Partially ordered communication

Our solution as presented so far requires a global order for all creation requests. However, this architectural requirement has similar costs as requiring a global memory consistency. In constrat, *local ordering* of messages, *i.e.* preservation of order between pairs of processing resources, is significantly cheaper to implement. As only the messages sent from one resource need to have a consistent order, no global state needs to be maintained.

To support architectures that only provide ordered point-to-point communication, such as the Microgrid, we have to extend our approach slightly. In this case, ordering issues arise when sub-computations sharing a RC place are delegated concurrently to different places. We assume that threads, once created, do not migrate across places.

In this context, a caller increments the reference counts of all arguments of a spawned thread before the spawn. These RC operations must complete before the spawned thread potentially performs **decrc**. If both threads run on the same place, this is ensured by the local ordering between the threads' place and the exclusive place used for RC. If, however, the spawned thread is delegated to another place, this ordering is no longer guaranteed. In particular, the RC operations issued by the child thread's place may be performed before those issued by from the parent thread's place. To prevent this, we introduce another RC operation, **flush**. It is implemented by a synchronous request that performs no action. Due to the local order, completion of **flush** implies that all previous RC operations have completed. In particular, if a **flush** is issued after **incrc**, it is safe to delegate a thread after **flush** completes.

The absence of a global order affects **getrc** as well. Its result is no longer an accurate representation of the system's state. Although **getrc** is synchronous, there may still be subsequently issued RC operations pending from other places.

This may lead to *false negatives*, *i.e.* to situations where a reuse opportunity is missed due to an inaccurate counter value. However, *false positives*, *i.e.* a reuse of an object that is still referenced otherwise, cannot occur. Indeed, if the counter has been decreased to 1 at any stage of a computation, this means that only one thread may hold a reference to the corresponding object. Thus, all pending RC operations must belong to that thread as otherwise a thread would emit an RC operation on an object it no longer references, which is impossible. As **getrc** is performed synchronously by the thread that holds the last reference to the object, all other RC operations for that object will complete before **getrc** completes. Thus, if the result is 1 then the enquiring thread at the point of enquiry is holding the single last reference. Consequently, memory reuse is safe.

This extended scheme requires only a minor adaptation in compilers. Before delegating a spawned thread to a different execution resource, we now additionally insert **flush** operations on all array arguments of the thread.

5. Feedback on the Microgrid design

We report on the implementation and evaluation of our proposed scheme separately in [9]. Retrospectively, our results have provided feedback to the Microgrid implemen-

tation. This happened on two levels: a reflection on the on-chip network protocol due to the ordering requirements from the scheme and optimization strategies to the on-chip thread spawning processes to remove unnecessary synchronization.

5.1. Partial ordering of network requests

As explained in subsection 4.3, the minimal requirement for our RC scheme is that requests issued from the same thread are processed in order, although not necessarily in order with requests from different threads. As this was not guaranteed by the abstract SVP model, we had to look at the processes involved on chip. When a thread requests the remote creation of a family at another place, the following happens: on the originating core, a request is *issued* onto the delegation network, then *routed* to the target place, then at the target core, it is *queued* for later processing. Local ordering must thus be ensured at these three levels.

Both issuing and queueing requests use a single queue so the ordering is trivial. All individual cores are connected to the delegation network, which has a mesh topology. Here the Microgrid implements dimension order routing, which also preserves order and satisfies our requirement. In other chips, the work delegation protocol should respect this ordering between execution places. The SVP semantics have since been refined to provision our ordering requirement.

5.2. Hardware concurrency optimizations

While experimenting with our first implementation, we discovered that the Microgrid was preventing the expected latency hiding benefit of asynchronous delegations, by requiring excess synchronization not expressed in the code.

The issue arises from the phase separation of family creation on chip. Any expressed creation implies the following:

1. The creating thread performs the **allocate** operation, provided with a place identifier, which triggers the allocation process. This reserves a family context and at least one thread context on the target place and returns a *family identifier* as result (comprising the address of the target). With remote places this causes a network roundtrip to allocate.

2. Using the family identifier, the creating thread can then *configure* the family context to define the logical thread count. Configuration does not entail synchronization and is optional: by default only one thread is created.

3. Once the family is configured, the thread performs **create** on a family identifier and an initial program counter, and completes when the creation process at the target place has started. With remote places, this also causes a network roundtrip.

4. After creation has synchronized, the creating thread can optionally *send thread function arguments* to the running child family asynchronously.

Once these steps are completed, the child family is running asynchronously with the parent thread, which may choose not to synchronize on termination of its child and instead **detach** from it for truly asynchronous completion[1].

In the case where this targets an exclusive place, the expected asynchrony is unfortunately prevented. Indeed, exclusivity is implemented by reserving a single family context for exclusive delegations on each core. This causes any *allocation* to unnecessarily synchronize on the *completion of all previous exclusive requests*. We illustrate this in Figure 5: 2000 threads are created over 64 cores; each thread performs a **incrc** as its first operation. Due to over-synchronization on allocation, the threads are unnecessarily sequentialized (low pipeline efficiency) up to t≈150kcycles where all RC operations have been queued. After that point the threads can run in parallel (high pipeline efficiency) and the RC operations become truly asynchronous. This behavior runs counter to the desired laziness of our scheme.

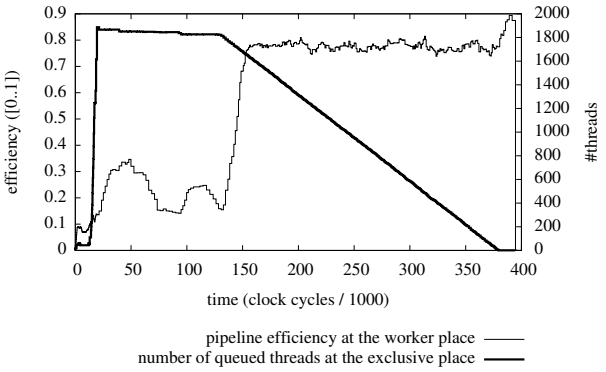


Figure 5. Effect of excess synchronization.

A first approach to overcome this limitation was to adapt the architecture by combining the **allocate**, **create** and **detach** steps into a single operation which only synchronizes on the queueing, not on the execution. This is conceptually possible if the exclusive families do not need configuration, *i.e.* are single-threaded, because then the family context can be pre-allocated. Then a special *detached create* request could be sent, using the family identifier and program counter as input, that simply queues exclusive creation at that family entry and synchronizes on queueing, followed by an asynchronous request to provide the function arguments to the created family. Because partial ordering is guaranteed, the arguments would always arrive after the create that they pertain to. However, this approach requires

[1] **detach** is also an asynchronous request.

a more complex buffering arrangement that provides non-FIFO access to arguments. This is because arguments from different creates may be interleaved in the queue. Hence, an associative FIFO lookup is required based on a per-create tag to avoid deadlocks (any thread on any core can issue creates and arguments out of order). A proper tagging scheme would be also complex to design. Finally, a finite argument buffer could cause deadlock if it became full with arguments unrelated to the first entry in the create buffer. Thus some form of software-directed deadlock prevention would also be required, adding more to the complexity of this solution.

At this point we observed that the fundamental requirement for the desired level of asynchrony was the ability to queue work remotely, at the exclusive place, using a single network message containing all the information required. To achieve this, a seemingly unrelated feature exists on the Microgrid: *system call gates*, originally designed for implementing privileged operating system services like resource allocation and I/O access. Looking at this mechanism, we see that this consists of a combination of two features: an **indcreate** operation, which takes a *service identifier* and an argument value as input and queues requests remotely, and an *indexed create process* at each core, which takes as input requests from its local queue and processes allocation and creation. This is illustrated in Figure 6: a table in memory maps service identifiers (SID) to code. The remote identifier given to **indcreate** contains both the network address of the remote core, the SID, and bits to indicate whether to detach and/or create exclusively. Each core is configured with a base table address in memory, which is then offset by each SID to find a program counter and possible other family configuration data. The extra argument value provided to **indcreate** becomes the index value of the first thread in the created family. This way clients can provide function arguments to the service. By assigning reference counter operations to entries in the service table of the exclusive place, we obtain the desired asynchrony using the input queue of the indirect create process.

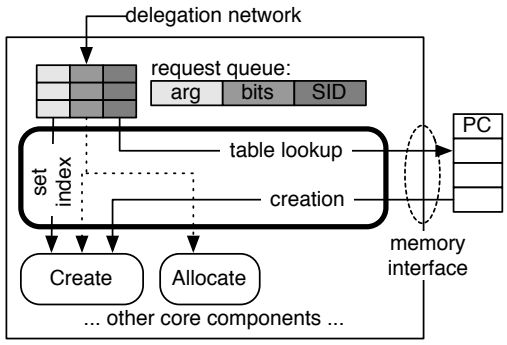


Figure 6. Indirect create process on chip.

While this mechanism was designed independently, our RC scheme helped refine the design. Firstly, the mechanism was originally designed so that both **indcreate** and the indexed create process would execute at the issuing core with the resulting allocation/create being delegated if necessary. Instead, the inverse solution where **indcreate** queues remotely is what we need. Also, the use case requires that client threads must be able to synchronize on the queueing, so as to receive a guarantee that the RC operation will *eventually* be performed *before* any subsequently queued request is serviced. This means that an **indcreate** operation must only synchronize when the remote queueing is acknowledged. While we clarified this requirement in the light of reference counting, it is actually a quite general prerequisite to many software locking schemes.

6. Related Work

Although we are aware of recent work on using non-deferred reference counting in the context of mainstream object oriented languages [14], we did not come across any work of non-deferred reference counting in the context of distributed memory or weakly coherent shared memory multi-cores. 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 [1] for many-cores. In Barrelfish, instead of using a single global kernel and shared state, the operating system is built around communicating network of kernels. Each computing resource is managed by its own kernel and state is replicated using message passing. Similar to our approach, system services are delegated to responsible cores that hold the corresponding state instead of communicating the state. The driving force here, like in our approach, is scalability.

Similar, but on a significantly larger scale, distributed file system have to contend with typically large objects (files) duplicated 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 [2, 15] and the Google File System [5] 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*. Separately to 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 the lazy updates, synchronous reads in our scheme.

7. Conclusion

The primary contribution of this paper is a novel *lazy, non-deferred reference counting scheme* targeting a many-core architecture with shared memory. To reduce communication requirements our scheme ships the reference counting operations to a shared processing resource instead of migrating the counter data to the thread that needs to perform a reference counting operation. As a welcome side effect this allows our scheme to run on systems with weak or no cache coherency, by ensuring that all updates to the same counter are performed by the same processor. To reduce contention on the shared resource and to enable latency tolerance we queue counter updates asynchronously without synchronizing on their completion: synchronization is only required at the point of potential reuse. Finally, we only require only partial network routing order between requests issued by the same thread, instead of a global order between all clients of a reference counter.

We have implemented this scheme successfully in the array programming language SAC. Our implementation targets the SVP programming interface and its many-core hardware implementation, the Microgrid. Our work has helped refine the Microgrid design and suggested optimizations for additional hardware concurrency. This positive feedback on the design of a many-core chip architecture constitutes our secondary contribution.

8. Acknowledgements

The research was performed at the School of Computer Science, University of Hertfordshire, UK, and the Institute for Informatics, University of Amsterdam, NL, under EU research grant FP7/2007/215216 Apple-CORE.

References

[1] A. Baumann, P. Barham, P. E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *Proc. 22nd symposium on Operating systems principles (SOSP'09)*, pages 29–44. ACM, 2009.

[2] D. Borthakur. The Hadoop distributed file system: Architecture and design. 2007.

[3] K. Bousias, L. Guang, C. Jesshope, and M. Lankamp. Implementation and Evaluation of a Microthread Architecture. *Journal of Systems Architecture*, 55(3):149–161, 2009.

[4] E. W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1(2):115–138, June 1971.

[5] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, 2003.

[6] C. Grelck, S. Herhut, C. Jesshope, C. Joslin, M. Lankamp, S.-B. Scholz, and A. Shafarenko. Compiling the Functional Data-Parallel Language SAC for Microgrids of Self-Adaptive Virtual Processors. In *Proc. 14th Workshop on Compilers for Parallel Computing (CPC'09), IBM Research Center, Zürich, Switzerland*, 2009.

[7] C. Grelck and S.-B. Scholz. SAC: A functional array language for efficient multithreaded execution. *International Journal of Parallel Programming*, 34(4):383–427, 2006.

[8] C. Grelck and K. Trojahner. Implicit Memory Management for SaC. In C. Grelck and F. Huch, editors, *Implementation and Application of Functional Languages, 16th International Workshop, IFL'04*, pages 335–348. University of Kiel, Institute of Computer Science and Applied Mathematics, 2004. Technical Report 0408.

[9] S. Herhut, C. Joslin, S.-B. Scholz, R. Poss, and C. Grelck. Concurrent non-deferred reference counting on the microgrid: first experiences. In *Proc. 22nd international conference on Implementation and application of functional languages (IFL'10)*, pages 185–202. Springer-Verlag, 2011.

[10] P. Hudak and A. Bloss. The aggregate update problem in functional programming systems. In *Proc. 12th ACM Symposium on Principles of Programming Languages*, pages 300–314. ACM, January 1985.

[11] C. Jesshope. Operating systems in silicon and the dynamic management of resources in many-core chips. *Parallel Processing Letters*, 18(2):257–274, 2008.

[12] C. Jesshope, M. Hicks, M. Lankamp, R. Poss, and L. Zhang. Making multi-cores mainstream – from security to scalability. In *Advances in Parallel Computing*, volume 18. IOS Press, 2010.

[13] C. Jesshope, J.-M. Philippe, and M. van Tol. An architecture and protocol for the management of resources in ubiquitous and heterogeneous systems based on the SVP model of concurrency. In *Embedded Computer Systems: Architectures, Modeling, and Simulation*, pages 218–228, 2008.

[14] P. G. Joisha. A principled approach to nondeferred reference-counting garbage collection. In *Proc. 4th International Conference on Virtual Execution Environments (VEE'08)*, pages 131–140. ACM, 2008.

[15] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop distributed file system. In *Proc. 26th Symposium on Massive Storage Systems and Technologies (MSST'10)*. IEEE Press, May 2010.

[16] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic Storage Allocation: A Survey and Critical Review. In H. Baker, editor, *Proc. International Workshop on Memory Management (IWMM'95)*, volume 986 of *Lecture Notes in Computer Science*, pages 1–116. Springer-Verlag, 1995.