FP7-215216

Architecture Paradigms and Programming Languages for Efficient programming of multiple COREs

Specific Targeted Research Project (STReP) THEME ICT-1-3.4

Data Parallelism and Functional Parallelism on the Microgrid using Single Assignement C

Deliverable D4.4, Issue 1.0

Workpackage WP4

Author(s):	Sven-Bodo Scholz, Stephan Herhut, Carl Joslin		
Reviewer(s):	Chris Jesshope		
WP/Task No.:	WP4	**Number of pages:**	19
Issue date:	2011-07-31	**Dissemination level:**	Public

Purpose: In this deliverable we report on the outcome of our efforts to combine our previous work on SaC compilation systems for data-parallel and functional concurrency on the Microgrid into a single joint back-end. We give an overview of the challenges we faced and the approaches we have developed. In particular we focus on resource management and concurrency mapping techniques. Furthermore, we give a brief outlook on future research directions.
Results: In the context of resource management, we have implemented novel heap management and liveness analysis techniques. We show that solutions that are viable for either data-parallel or function concurrency in isolation are not applicable in a joint setting. We show how our previous approaches have to be extended to ensure scalability and runtime performance in a combined setting. For concurrency mapping, we have evaluated multiple schemes for functional concurrency, none of which leads to satisfying results in the general case. However, we have learned valuable lessons and do have a clearer understanding now of how a successful mapping strategy might look like.
Conclusion: Handling functional concurrency efficiently is complex and poses many challenges not present in the data-parallel case. Yet, we have developed runtime solutions that allow us to efficiently execute programs that contain both. We have developed novel techniques tailored not only to the Microgrid but massively parallel architectures in general. Some questions, in particular means to find an efficient, scalable, yet adaptive mapping of functional concurrency to hardware resources, require further research.

Approved by the project coordinator: Yes **Date of delivery to the EC:** 2011-07-31

Document history

When	Who	Comments
2010/12/3	Sven-Bodo Scholz	Initial version
2010/12/20	Stephan Herhut	added section on scheduling issues
2010/2/14	Carl Joslin	added section on new reference counting
2010/6/17	Sven-Bodo Scholz	added Quicksort results and discussion
2010/7/27	Sven-Bodo Scholz	added FFT results and discussion

SEVENTH FRAMEWORK PROGRAMME

Project co-funded by the European Commission within the 7th Framework Programme (2007-11).

Table of Contents

1 Introduction

From a language designer and compiler implementer's perspective, the Microgrid architecture [2] is a radical detour from existing, well understood hardware designs. Other than well studied multi-core processors and SMP machines, the Microgrid puts concurrent execution at the heart of its performance promise. The architectural design is geared towards parallel execution, and little of the precious silicon has been spent on sequential performance.

A similar point can be made for current, throughput oriented hardware like the Cell's streaming processors or AMD and NVIDIA's GPGPU offerings. Although these processors too are designed with massively parallel execution in mind, their execution model is rather restricted. In either case, basically only SIMD style executions are permitted. Other than these approaches, the Microgrid promises a relaxed, fine grained concurrency model, enabling a liberal mapping of a program's inherent concurrency to the underlying hardware.

As often, with liberty comes the responsibility for one's actions. The main challenges faced during the Apple-CORE project and, as a result, the solutions we propose, deal to a lesser extend with exhibiting concurrency in the first place. Instead, we have spend most of our effort on responsible concurrency. Much like citizens in an increasingly overpopulated world, threads in a massively parallel architecture like the Microgrid contend heavily for resources. Ensuring that resources are appropriately shared to the benefit of the whole system is a hard problem. Keeping management overheads down, too, adds further complication. Our work in the context of Apple-CORE has focussed on finding answers to both.

To keep the project manageable, we have early on decided to address data-parallel concurrency and functional concurrency, often referred to as task parallelism, as well, separately. On a conceptual level, data-parallel and functional concurrency seem similar. Yet, the way language constructs of a high-level language like SaC [12] are mapped to the low-level `create` operation in SL [10] is mostly orthogonal. Data-parallel operations basically lead to a nesting of few `create` operations that each spawn many threads [4] whereas functional concurrency typically results in many, deeply nested `create` operations that spawn few threads. As a result, the compilation technology involved is rather specific to the kind of concurrency targeted. For instance, the allocation strategies to map the available concurrency of the algorithm to the bounded concurrency resources provided by the hardware differ significantly. Whereas data-parallel concurrency is usually constrained by the number of available hardware threads, functional concurrency often suffers from exhaustion of thread families. Code generation has to take these effects into account.

Even for aspects where a solution was not expected to depend on the kind of concurrency, a staged approach seemed reasonable. For example in heap management, although the structure of the exhibited concurrency vastly differs between data-parallel and functional concurrency, the essential problem of safe concurrent access to a shared heap remains the same. Thus, we anticipated to reuse a single heap manager in either case. Yet, our experience has shown that subtle differences in the lifetime of threads and data can have a huge effect. As we have reported in [6], we had to make a significant investment into an extended, fully thread safe SaC runtime system. In particular, we have developed an asynchronous, distributed heap manager and a novel asynchronous non-deferred reference counting technique [8]. The automated extraction of functional concurrency itself, however, was left for the final year of Apple-CORE.

First experiments with different mapping strategies quickly unveiled the complexity of the task at hands. The mapping of extracted concurrency to the hardware generally is essential for good runtime performance on the Microgrid. For functional concurrency, however, the existing mapping strategies implemented by the hardware scheduler of the Microgrid do not suffice and much finer control in software is required. We report on our findings in Section 3.

Another key issue that we have addressed during the last year of APPLE-Core was the integration of our work on functional and data-parallel concurrency. We found that supporting functional concurrency inside of data-parallel sections and vice versa puts significant strain on the runtime system. In particular, reference counting based non-deferred garbage collections turned out to be a major constraint on scalability [8]. Section 4 presents our refined approach that now scales well even in the context of mixed concurrency.

We close this report with an overview of the implementation status in Section 5 and final conclusions in Section 6.

2 Combining Functional Paralllism and Data Parallelism

Before we detail the various improvements during the final year of Apple-CORE, we first give a short recap of the implementation of functional and data-parallel concurrency in SaC. Essentially, we have focussed on three key aspects:

1. mapping a program's inherent concurrency to the Microgrid hardware,
2. managing a massively shared heap, and
3. efficient non-deferred reference counting.

The first aspect, finding appropriate mappings of concurrency to hardware resources, so far was researched mainly for data-parallel workloads. We early on found that mappings with maximum width and minimum depth made optimal use of the Microgrid's resources [4]. Width in this context refers to the number of threads issued by each single `create` statement whereas depth refers to the maximum nesting of `create` operations. A sufficiently large width ensures good use of the hardware threads on a Microgrid. Furthermore, resource usage due to a mapping's width can be well controlled. The depth, on the other hand, directly relates to the number of thread families required. This resource, however, is usually scarce. We have implemented a range of optimisations to reshape the concurrency mapping towards the wide and shallow optimum [13].

For the second aspect, managing a massively shared heap, we have proposed a range of approaches tailored to specific usage scenarios. In the data-parallel setting, we were able to exploit static knowledge about the lifetime of threads and data to implement a very lightweight heap management strategy. Essentially we exploit the fact that the lifetime of local data is often aligned with the allocating thread's lifetime. Thread local stacks [7] use the hardware managed thread local storage on the Microgrid to store such short-lived local data structures. Like for function stacks, allocation boils down to a pointer increment. At the end of a thread's lifetime, the memory is automatically reclaimed by the hardware.

In the setting of functional concurrency, however, the alignment of lifetimes can no longer be guaranteed. Thus, thread local stacks do not apply. As an alternative, we have developed a distributed, lock-free heap manager that allows cheap local allocation and uses a lightweight garbage collection scheme to defer free operations until they are safe to perform [13]. Other than thread-local stacks, this approach uses fully software controlled memory invalidation, a feature that was specifically added to the Microgrid architecture. This is yet another example of successful hardware/software co-design in the context of Apple-CORE.

To combine functional and data-parallel concurrency, we have devised a hybrid scheme that supports both thread local stacks and the distributed heap manager at the same time. However, as our distributed heap manager requires software managed invalidation of memory, the hybrid scheme simulates hardware invalidation of thread local stacks in software. Although this introduces a slight overhead, the characteristics and runtime behaviour of each component of the hybrid approach remains largely unchanged.

The third and last aspect we have examined is the implementation of efficient reference counting schemes for non-deferred garbage collection on a massively parallel architecture like the Microgrid. We provide an in depth discussion of the various schemes in Section 4. For the data-parallel scenario, which generally exhibits only moderate reference counting loads, we have devised a distributed scheme that exploits the latency hiding capabilities of the Microgrid to perform reference counting operations concurrently with the actual workload [8]. The key insight that lead to this novel approach is that a certain amount of imprecision of liveness information can be tolerated even in non-deferred garbage collection. Most heap management operations do not require fully up-to-date liveness information. Only update-in-place optimisations, which are critical in functional languages like SaC [5], require this level of precision. We exploit this fact to offload reference counting operations to a dedicated hardware resource by means of asynchronous message passing. Only for update-in-place operations, we enforce a consistent global state.

The above list of aspects does not include automated concurrency extraction from program texts. We have left the extraction of concurrency from the program source mainly explicit. In the case of data-parallel concurrency, the programming model of SaC already encourages the programmer to write applications in a data-parallel fashion. Thus, programs implemented in SaC usually exhibit sufficient levels of explicit data-parallelism to yield efficient utilisation of the Microgrid's hardware resources. Even though the parallelism is explicit in the program text, the programmer hardly notices. From a programmer's perspective, SaC's data-parallel construct, the with-loop, is just a loop construct and often its uses are well hidden in SaC's standard library.

The integration of functional concurrency uses a different strategy. Ultimately, we hope to infer suitable function applications from the control flow. Currently, to aid experimentation, we use explicit annotations. Other than the data-parallel case, these annotations solely serve the purpose of expressing concurrency and are not deeply embedded into SaC's programming model. A detailed description can be found in [6].

3 Managing Resource Mapping Strategies

A key challenge in mapping concurrent programs to the Microgrid is finding an appropriate mapping of the concurrency as exhibited by the program to the hardware resources. In the case of data-parallel concurrency, we leave most of the mapping and scheduling to the hardware. In particular, we do not assign computations to specific cores or places. This is motivated twofold. Firstly, by leaving the exact mapping open, we ensure that the generated executables can tolerate a range of resource constraints. These constraints can either be due to different implementations of the Microgrid with varying concurrency resources, or due to multiple applications sharing a single host. As the hardware and its embedded scheduler know about the specific constraints of the executing machinery, they can adapt the program at runtime. This, of course, could be achieved by a machine specific runtime system in software, as well. But, secondly, scheduling in hardware induces significantly less runtime overhead. This is particularly the case for the Microgrid where thousands of threads need to be orchestrated.

The generated SL code does, however, contain annotations that describe which threads should be distributed globally across the entire Microgrid and which threads are intended to be executed on the current core. If a `create` operation is annotated for distributed execution, the hardware scheduler will distribute threads across all cores, beginning with the core where the `create` operation was issued. For data-parallel workloads, this usually is the intended behaviour and we have found this means of thread distribution to be sufficient. In the case of functional concurrency, however, this scheduling strategy is not adequate.

As an example, consider the compilation of a SaC implementation of quicksort, a classical divide and conquer style algorithm. Figure 1 shows SaC pseudo code for an implementation. As can be seen, while the list given as argument is non-empty, it is split into two sub lists based on a pivot element. These lists are then sorted using a recursive application of `quicksort`. Finally, the result is computed as the concatenation of the two sorted sub lists. A seemingly obvious means to exploit functional concurrency in this algorithm is to spawn both recursive descends as concurrent threads. To this effect, we use two spawn statements in Lines 7 and 8. Each creates a new thread that will execute the function `quicksort` on one of the sub lists. We will come back to the additional *target* argument shortly.

Each `spawn` operation in SaC is mapped to a corresponding `create` operation in SL [6]. Consequently, each recursive call to `quicksort` will create two hardware threads on the Microgrid. If we were to use the same hardware scheduling as for the data-parallel case, all threads, regardless of distribution annotations, would end up on the same core. For local `create` operations this is

```
1  function quicksort( list ):
   if empty( list ):
3          return [];
   else:
5          pivot = list[0];
           filterL, filterR = filter( list, pivot );
7          qsortL = spawn( target ) quicksort( filterL );
           qsortR = spawn( target ) quicksort( filterR );
9          return qsortL ++ [pivot] ++ qsortR;
```

Figure 1: Pseudo code of a quick sort implementation in SaC.

obvious, as they are intended to spawn threads on the same core. In the case of distributed `create` operations, this is an artefact of the chosen scheduling: Each family is spread across all cores of the Microgrid, beginning with the core where the corresponding `create` operation originated. Note that even changing this scheme to start with the core next to the current core would not be good enough. Still, both threads spawned by one instance of `quicksort` would use the same core.

To overcome this limitation and still benefit from automated hardware scheduling, we have developed a scheme that aims at balancing the created threads across the Microgrid (see FSBR strategy in [9]). The idea is to automatically create threads at the core with the least workload. As a dynamic measure of a core's workload, we use the number of currently active families. As the Microgrid consists of conceptually idependently managed cores, an implementation of this scheme inevitably consists of two phases. In the first phase, a message is sent around the Microgrid's ring network to generate a snapshort of the current state of the system. In particular, we record the number of active families for each core. In a second phase, we select the core with the least load and send a delegation message to that core.

Compared to a traditional create, which only requires a single delegation message sent directly to the target core, this balanced scheme produced signigicant additonal overhead. To reduce this overhead, we have implemented a fast mode for boot strapping an empty Microgrid. If during the initial enquiry phase we find a core that is not yet computing, we abort further traversal of the ring and directly create the threads at that core. Still, as Figure 2 shows, the runtime overhead of dynamic load balancing is too costly for quick sort. The runtime improvement due to additional concurrency resources is outweighed by the increased distribution overhead due to longer communication latencies.

To reduce the overhead due to distribution, we have implemented a second version of quick sort that limits the exposed concurrency. This implementation is guided by the idea that, due to the incurred overheads, distribution of small workloads is not viable. Therefore, our enhanced implementation only spawns further threads if the remaining list to be sorted has more than 10 elements. Figure 3 shows the resulting runtimes. The algorithm now shows at least limited scaling up to 32 cores. However, from 64 cores onwards the distribution overhead shadows gains due to conrrent execution even for lists larger than 10 elements.

Ultimaltely, we came to the conclusion that fully automatic scheduling, although effective for data-parallel loads, is too costly in the functional concurrency setting. As before, the lack of known structure of the concurrency tree makes dynamic runtime approaches difficult. Predicting the concurrency behaviour of a program is hard and, as our results show, dynamic one-fits-all solutions are too expensive. Yet, using a fully explicit mapping approach would inhibit scalability of binary programs across different implementations of the Micrgrid with varying numbers of processing cores.

To achieve better performance but retain scalability of binary programs, we have evaluated an approach that compromises on both. We have extended the `spawm` operation in SaC by an additional target arguments, as shown in Lines 7 and 8 of Figure 1. The target qualifier is passed to the SL runtime layer as a hint on how to distribute the corresponding `create` operation. The following targets are available:

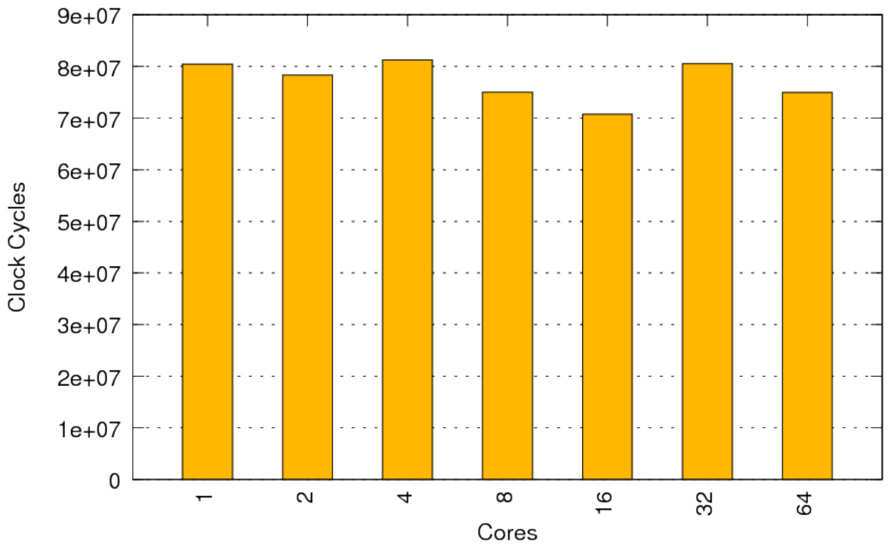


Figure 2: Runtime behaviour of quick sort implementation when using the Microgrid's hardware scheduler (balanced) without explicit distribution annotations.

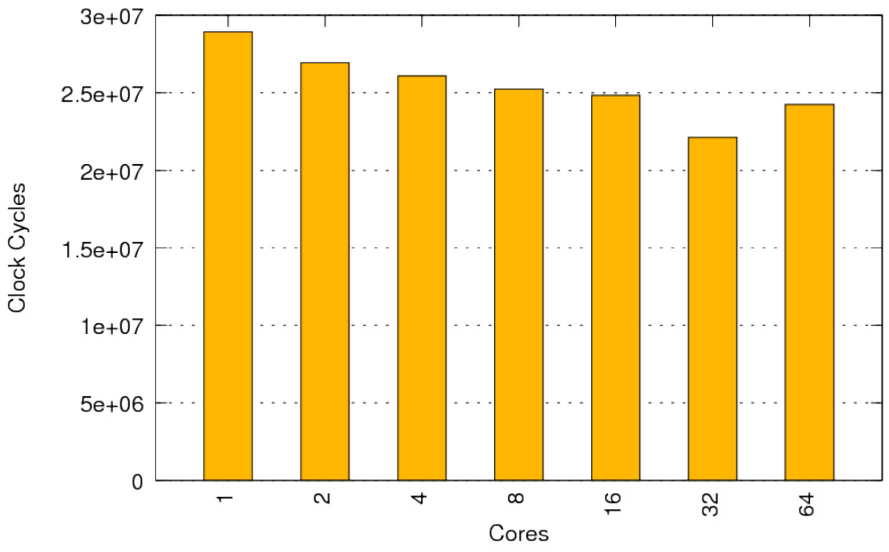


Figure 3: Runtime behaviour of our quick sort implementation with concurrency throttling when using the Microgrid's hardware scheduler (balanced) without explicit distribution annotations.

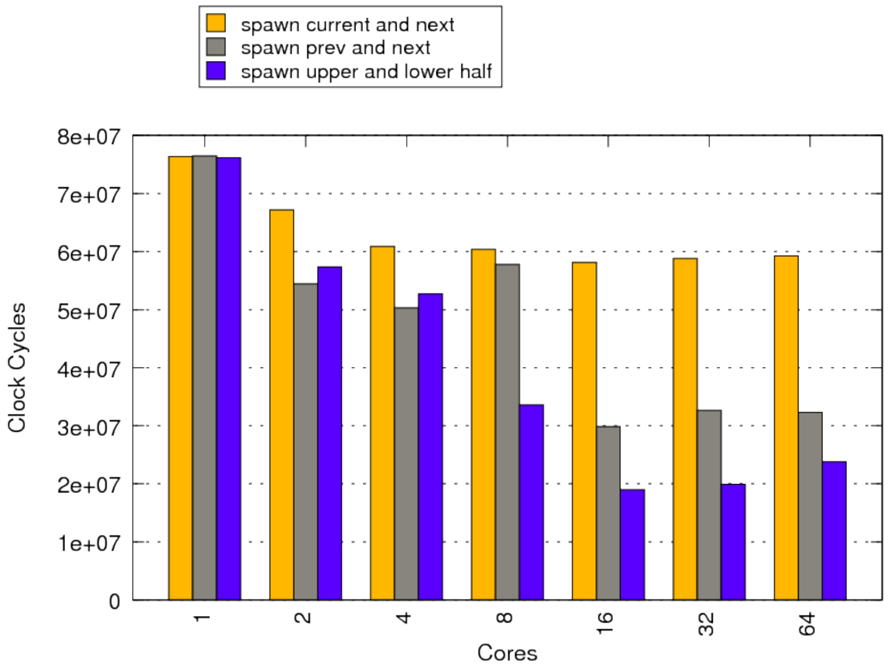


Figure 4: Runtime behaviour of quick sort when using the Microgrid's hardware scheduler with explicit distribution annotations.

NEXT_N_GCORE(N) use the core N hops down the ring network. If N is zero, the current core is used.

PREV_N_GCORE(N) use the core N hops up the ring network. If N is zero, the current core is used.

NEXT_HALF_GCORE use the core half way around the ring network.

These annotations still make no assumptions on the number of cores and thus allow the hardware scheduler to scale binary applications to the resources available on a specific execution machinery. Yet, they provide sufficient information to allow the scheduler to distribute work more efficiently. Figure 4 shows the execution times of the same benchmark as in Figure 2 using different target annotations. The first strategy we have used, *current and next*, only provides limited scaling. It, however, yields better overall runtimes compared to the fully automated distribution. The limited scaling can be explained by the under utilisation of the available processing cores. Figure 5 shows a simulation of the expected thread distribution[1] of quick sort using the *current and next* scheme. As can be seen, the resulting distribution after 12 recursive descends is highly unbalanced and only involves half of the available cores.

The *prev and next* scheme, which spawns the two subcomputations within each instance of quick sort on the previous and next core on the ring, respectively, fares better. Figure 4 shows good scaling up to 16 cores, where the runtimes even out. The simulated thread distribution if Figure 6 unveils a much improved distribution over our previous scheme. After 10 recursive descends, the computation has been distributed across the entire Microgrid. Other than the previous distribution, our new scheme distributes threads towards both sides of the first core. However, due to the nature of the scheme, only every second core is being used.

Lastly, we have evaluated the *upper and lower half* scheme. It distributes one spawn to the next available core and the other spawn to the core half around the ring network. As can be seen in

[1]Our simulation only considers target annotations but does not take resource constraints into account. On a real implementation of the Microgrid, not all threads might actually be created. Nonetheless, our model is well suited to predict maximum distribution.

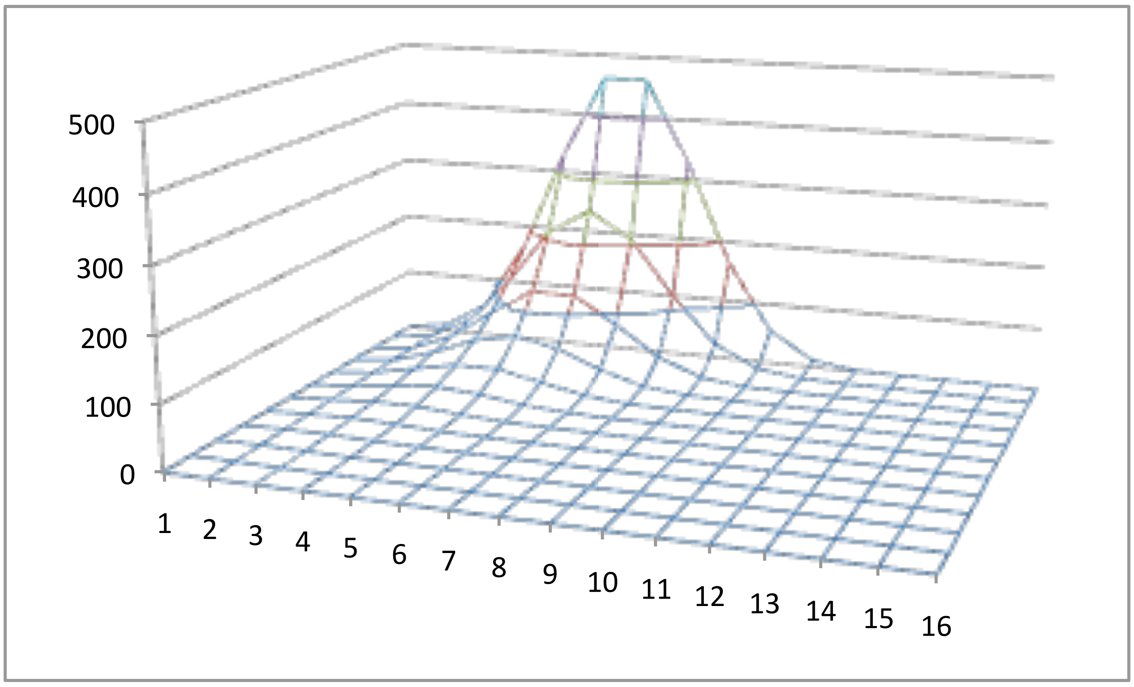


Figure 5: Simulated thread distribution using the *current and next* scheme on a Microgrid with 16 cores. The x-axis denotes the core whereas the y-axis gives the number of active threads. We have plotted time along the z-axis.

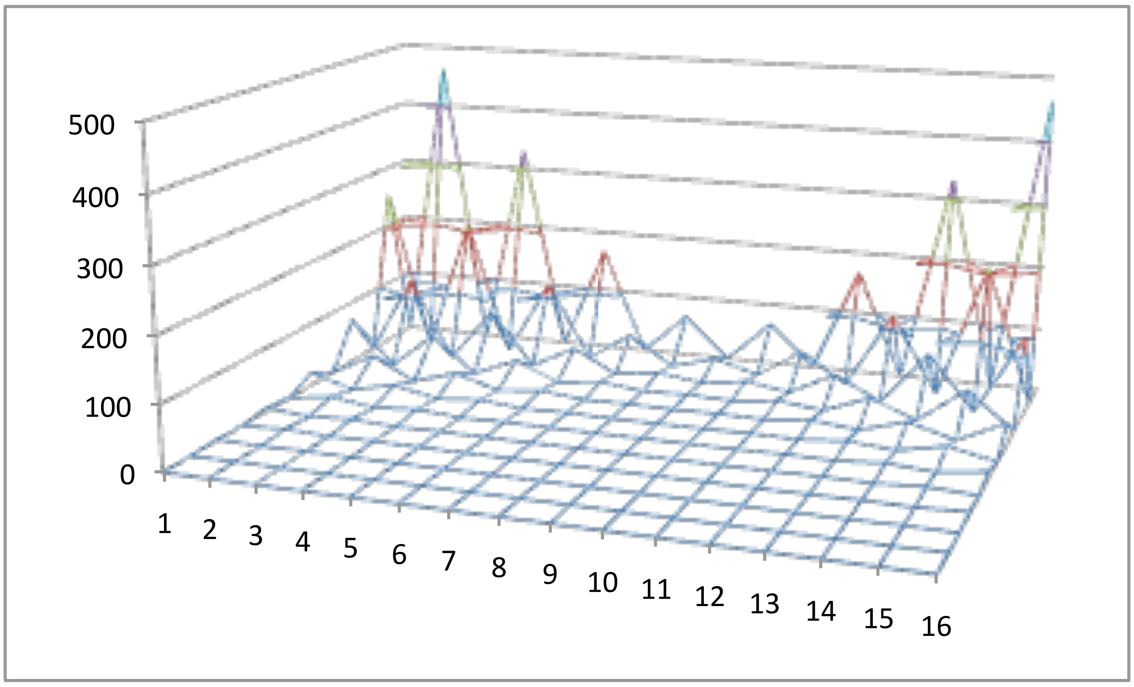


Figure 6: Simulated thread distribution using the *prev and next* scheme on a Microgrid with 16 cores. The x-axis denotes the core whereas the y-axis gives the number of active threads. We have plotted time along the z-axis.

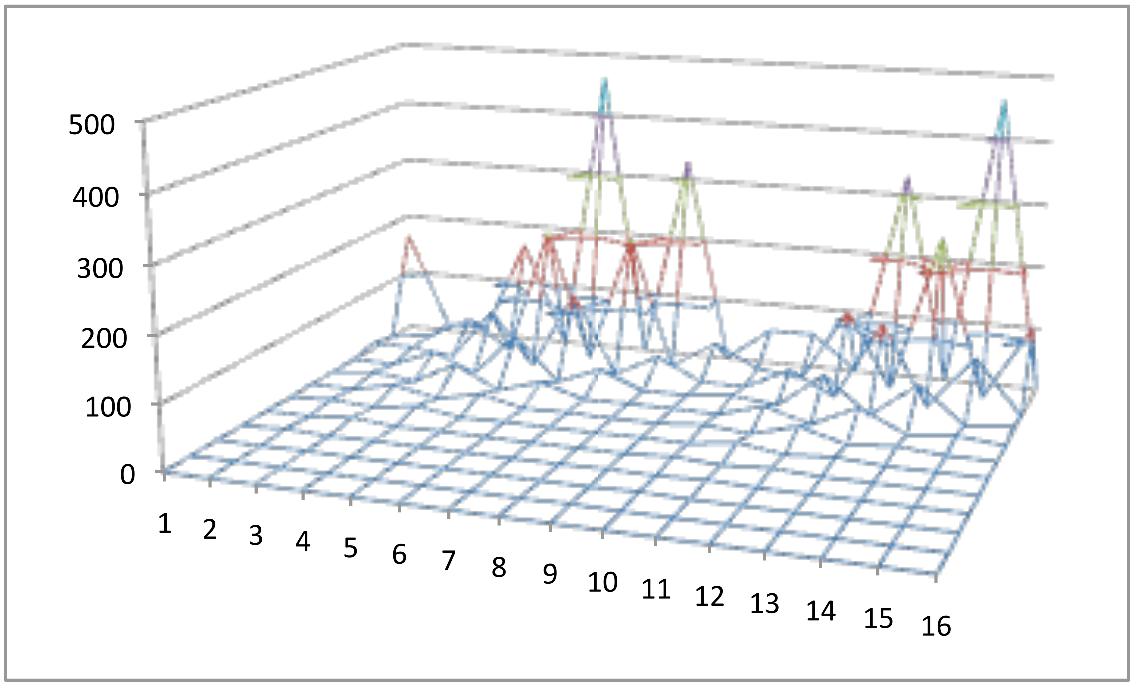


Figure 7: Simulated thread distribution using the *upper and lower half* scheme on a Microgrid with 16 cores. The x-axis denotes the core whereas the y-axis gives the number of active threads. We have plotted time along the z-axis.

Figure 4, this scheme performs slightly worse than the *prev and next* scheme on smaller Microgrids with less than 8 cores. However, for larger Micrgrids, it scales better. The cause is its more efficient distribution scheme, as shown in Figure 7. After ten recursive descends, most cores are active. This scheme produces two hot spots, whose centers are opposite on the ring network.

Comparing the results for our explicit mapping scheme in Figure 7 with the results from the automated balancing scheme in Figure 3 shows that the explicit approach scales better yet has significantly longer overall runtime. Whereas the automated approach completes in in about 30 million cycles on one core, our best explicit approach needs 16 cores to achieve similar runtimes. This suggests that we have traded sequential performance for scalability. However, the implementation in Figure 3 achieves its good sequential performance by limiting the recursive unfolding of concurrency to cases where the expected workload is big enough. To allow for a fair comparison, we have implemented the same approach using our best explicit mapping strategy, i.e., a *lower and upper half* mapping. Figure 8 presents our findings. As can be seen, using concurrency throttling, our explicit approach nearly reaches the sequential runtime of its counterpart using automated mapping. With increasing core counts, the overheads of the balanced approach start to show. Our explicit mapping scales linearly for up to 4 cores and reaches its peak performance at 16 cores.

Even though the explicit distribution using the *lower and upper half* strategy works well for less than 8 cores, its tendency to produce two hot spots fires back for larger core counts. Figure 9 shows the expected thread distribution for the *upper and lower half* scheme on 64 cores. Basically the pattern is the same as in Figure 7, just that now the two hotspots are far apart. Overall, though, the same number of cores is used.

As these experiments show, ultimately resource mapping will have to take the number of cores available on a Microgrid into account. We are currently evaluating further approaches that, although using a mapping depending on the number of cores, still allow for scalability of binary programs. Simply computing a mapping at runtime is too costly, as the hardware implementation of the balanced scheme has shown. Using higher-order annotations in spawn operations that are parameterised over the number of cores might be a solution. However, for performance reasons, such a solution needs to be implemented in the systems language SL and the hardware, as well.

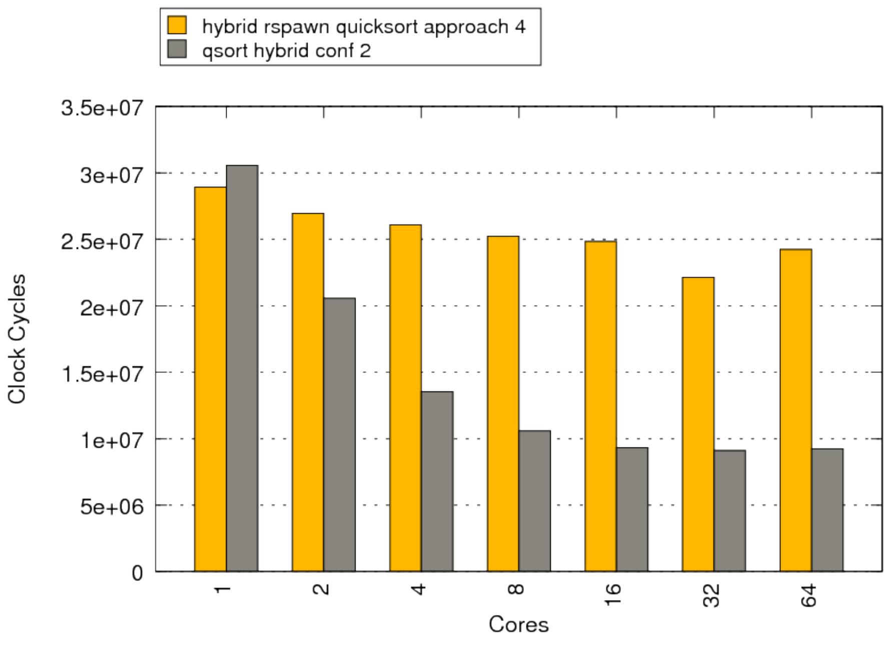


Figure 8: Runtime results of our quick sort implementation using concurrency throttling using automated balanced mapping (*hybrid rspawn quicksort approach 4*) and explicit lower and upper half mapping (*qsort hybrid conf 2*).

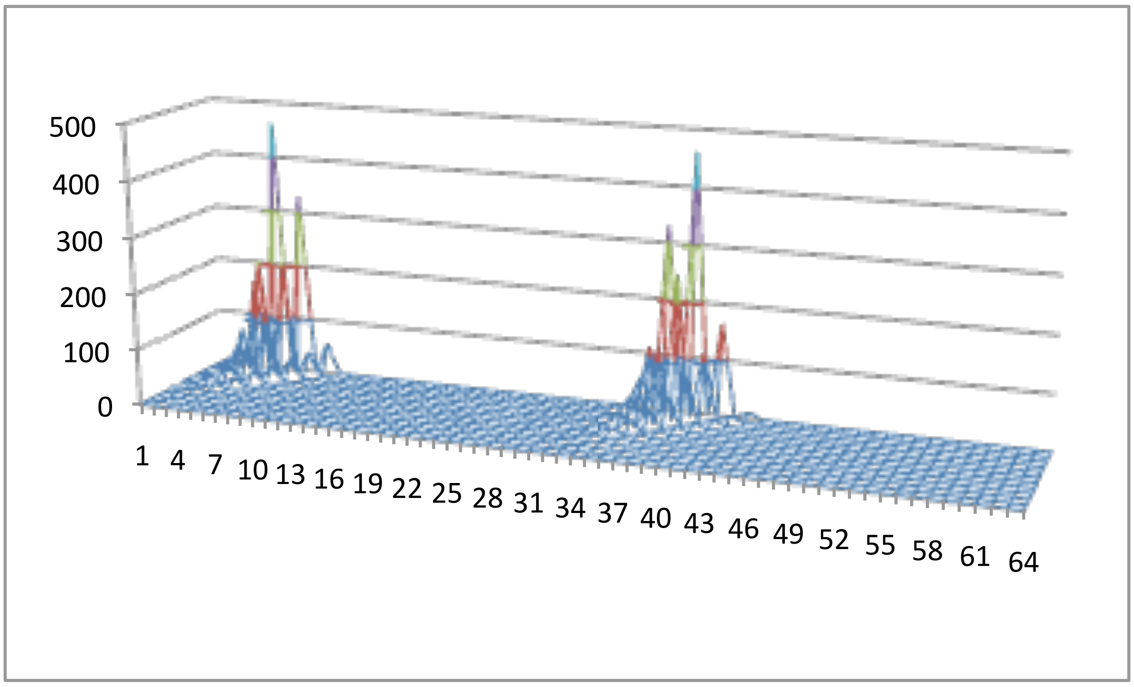


Figure 9: Simulated thread distribution using the *upper and lower half* scheme on a Microgrid with 64 cores. The x-axis denotes the core whereas the y-axis gives the number of active threads. We have plotted time along the z-axis.

4 Reference Counting Strategies For Combined FP and DP

In functional programming languages, data is usually treated as immutable values. Consequently, to update data, even partially, requires a copy of the original. In classical functional languages with is usually small data structures, the resulting runtime overhead is tolerable. For a language like SaC, however, where large array values are the predominant kind of data, copying values for partial updates is not an option.

Therefore, SaC uses reference counting to accurately track the liveness of data objects at runtime. If an array that is to be updated is not referenced otherwise, it can be updated in place without violating the immutability requirement. As an additional bonus, liveness information can be used for automatic heap management, thereby avoiding the overheads of garbage collection.

The benefits of reference counting come at a price. By its very nature, reference counting is sequential. It operates on shared state and usually updating that state requires a globally consistent view. For few numbers of threads, this sequentialisation can be tolerated as the parallel workload of each thread outweighs sequential components. In the context of the Microgrid, however, synchronisation costs due to maintaining the global reference counting state quickly become a bottle neck.

To overcome this hurdle, we have developed asynchronous non-deferred reference counting [8, 6]. Our novel approach exploits the latency hiding capabilities of the Microgrid to interleave reference counting operations and the actual workload. Reference counting operations thereby are asynchronously delegated to a separate execution resource, allowing the actual compute cores to continue execution of workloads. Only if the global reference counting state needs to be queried, we synchronize the querying core with the reference counting resource. This, of course, still leads to a sequentialisation of execution. However, only one core is involved and such operations a relatively rare.

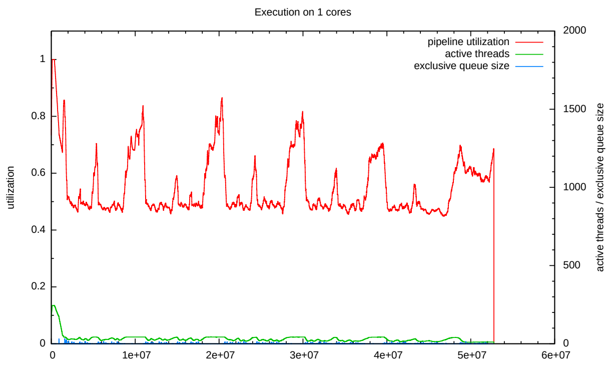
Figure 10 demonstrates the runtime behaviour of our approach for a two dimensional FFT on 256 element vectors. We have plotted three different metrics over time, which is measured in clock cycles. The first metric is the overall pipeline efficiency, given by the red line in percent (cf. left y-axis); it measures how many cycles, on average, are spent computing the actual workload as opposed to waiting for results or resources to become abvailable. On a single core, the utilisation always remains above 50% and ocassionally spikes above 80%. These spikes coinside with reduced reference counting loads, depicted by the blue line. We measure the reference counting load by observing the number of threads that await processing by the dedicated reference counting core. The correspondign scale is given by the right y-axis.

The last metric we observe it the number of active threads computing the workload, given in green. For our FFT example, we start out by using the maxiumum number of available threads, and consecutively create just enough threads to keep the machine occupied. Overall, on a single core, the computation of FFT requires just above 50 million cycles.

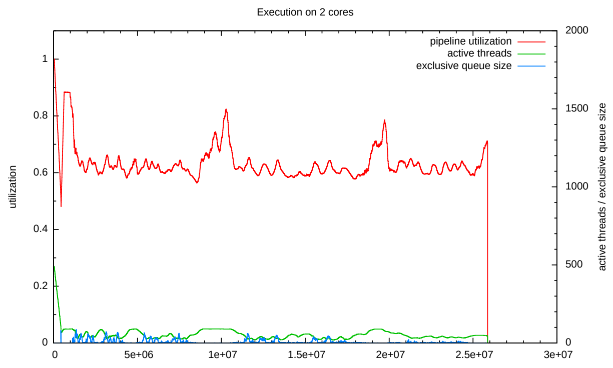
To study the scalability of our asynchronous reference counting approach, we have measured the same benchmark for varying numbers of cores. Graphs (b) and (c) exhibit nearly linear scaling for two and four cores. The overall runtime is reduced to 25 and 14 million cycles, respectively. However, from 8 cores onwards (cf. Graphs (d) to (g)) overall runtime stagnates at around 10 million cycles.

The reason for this stagnation is to be found in the increasing concurrent reference counting load. As an indicator, consider the queue size for pending reference counting operations at the reference counting resource (blue line). For up to four cores, the queue length periodically increases but over time returns back to an empty state. Thus, the reference counting core is able to cope with its workload before the other cores run out of actual work to compute. From 8 cores onwards, this is no longer the case. The reference counting queue is nearly constantly filled and the reference counting operations can no longer be completed in time. Instead, cores that compute the actual workload have to wait until space in the queue becomes available. This is reflected in the decreasing pipeline utilisation rate (red line).

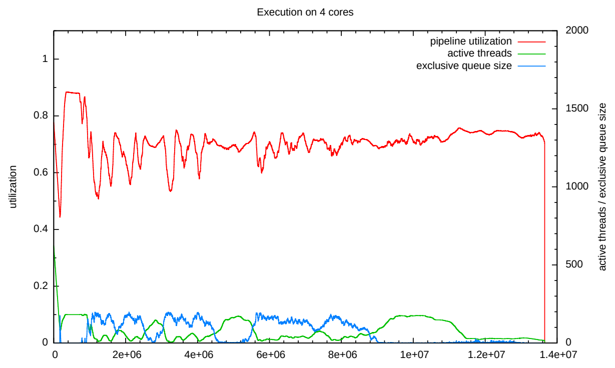
Thus, even though our approach is able to tolerate some reference counting load, ultimately



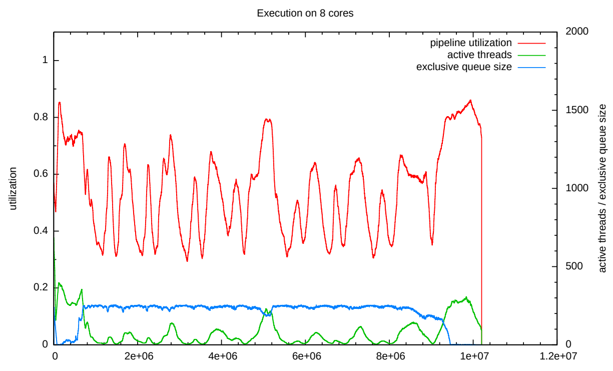
(a) one core



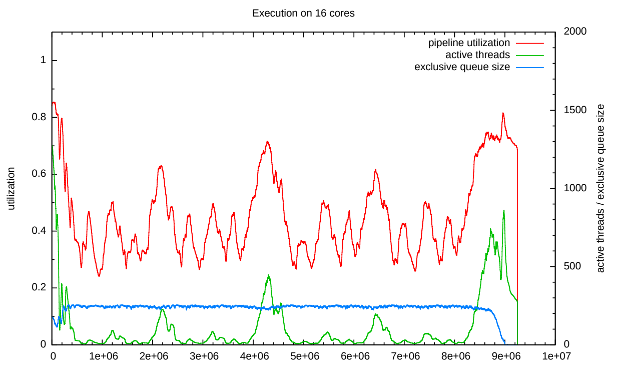
(b) two cores



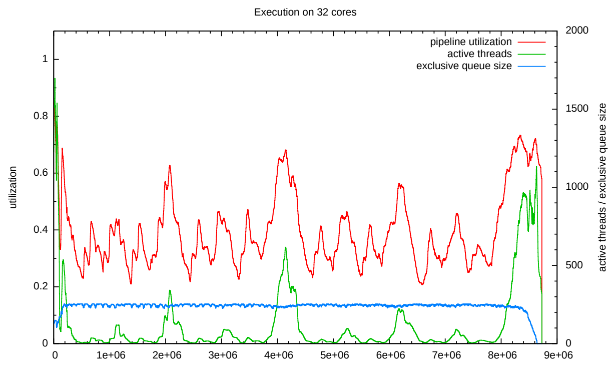
(c) four cores



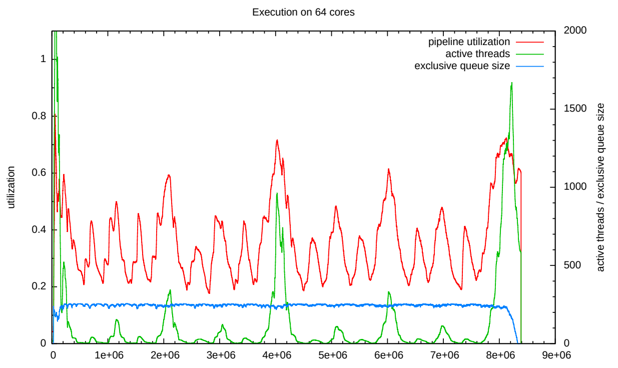
(d) 8 cores



(e) 16 cores



(f) 32 cores



(g) 64 cores

Figure 10: Runtime behaviour of a 2-dimensional 256 element FFT running on varying numbers of cores using asynchronous deferred reference counting.

```
int f( int[64000] vect, int idx, int load)
{
  res = vect[idx];
  for( i = 0; i < load; i++) {
    res += 1;
  }
  return( res);
}

int run( int[64000] vect, int size, int load)
{
  res = with {
          ([0] <= [i] < [size]) : f( vect, i, load);
        } : modarray( vect);
  return( res[10]);
}
```

Figure 11: An artificial example demonstrating the Amdahl effect from asyncronous reference counting in massive sharing in SaC.

Amdahl's law [1] hits us and the sequential component due to reference counting dominates. Consequently, the only solution that allows for true scalability is to reduce the reference counting load.

The effect we observe on FFT can be demonstrated in a nutshell using the example shown in Figure 11. All that is done here is to call a function `f` within a massive sharing situation and to pass the vector `vect` as an argument. This requires all threads to increment the reference count of `vect` roughly at the same time. Subsequently, within the body of the function `f` the reference count of `vect` is decremented after the variable has been read from. Finally, the parameter load allows us to control some artficial load. Given this setup, it is to be expected, that for small loads, the massive reference counting on `vect` dominates the overall runtime.

However, the data parallel setting here combined with the side-effect free nature of SaC code gives rise to a simple yet very effective optimisation. Whenever we pass a variable into a data parallel context, we know for sure that this data structure cannot be aliased and, thus, cannot leave the data-parallel context at all. As a consequence, we can, at least conceptually, surpress *all* refrerence counting on such variables within the data parallel context and treat the reference within that context as a single reference *outside* of the data parallel context.

Doing so, of course weakens the accuracy of liveness information gained. Even though the last reference to a variable passed into a data parallel section might lie inside of that section, we will only find out after the section has completed. Thus, we might miss an opportunity for updating a variable in place and introduce an additional copy. Although this at first glance seems to have a dramatic impact on runtime, actually little harm is done in practice. If an array update happens within a data-parallel construct, all threads but one will have to copy the array in either case. Thus, the runtime overhead already is high. High enough, in fact, to mask the additional copy. Furthermore, as our experience shows, such cases are rather rare.

With this observation in mind, we have created a bi-modal reference counting strategy. Upon allocation of a data structure, we start reference counting in asyncronous mode as needed for functional concurrency. Whenever we enter a data parallel context, we switch into a no-reference-counting mode for all data structures that are passed into the data parallel section (like `vect` in the example of Figure 11). After the data parallel section has been completed, we switch back to the asynchronous mode.

We have implemented this scheme in our research compiler. To infer whether a data structure is shared and thus has its reference counting suspended, we use a tag in the reference counting descriptor itself. On entry into a data parallel section, all shared data is tagged and on exit the

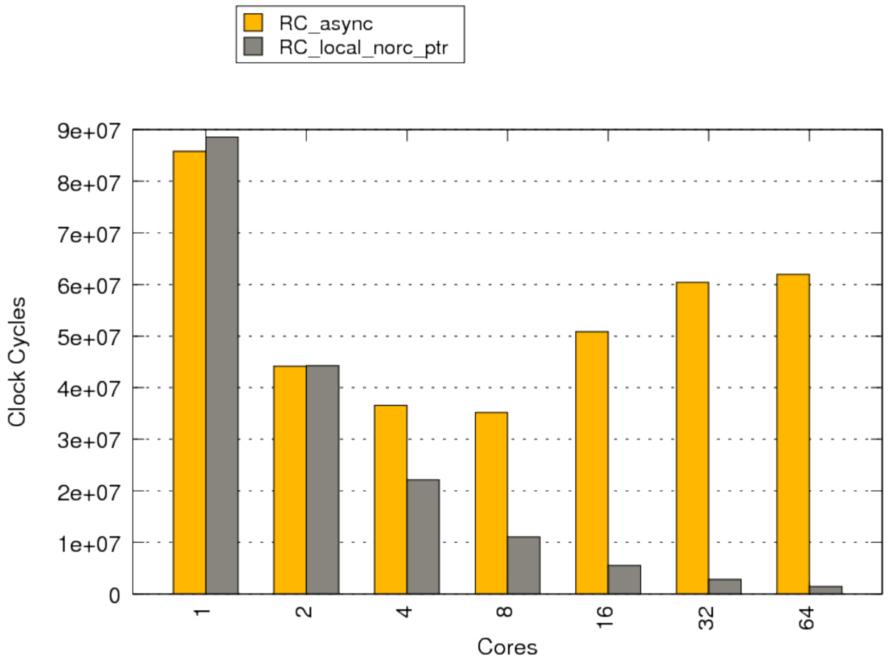


Figure 12: Runtime comparison for the artificial example from Figure 11 for asynchronous versus bi-model reference counting.

tag is reset to its previous value. A complication arises if the data structure is not only shared within the data-parallel section but between different threads emanating from spawn operations, as well. In this case, updating the tag in the reference counter, which is a shared resource, is not safe. Therefore, we create copies of the descriptors, thereby resolving the sharing. As we know that the data-parallel section has no impact on the value of the reference counter, it is safe to simply discard the copies after the data parallel section. Copying the descriptor also resolves potential sharing due to aliasing.

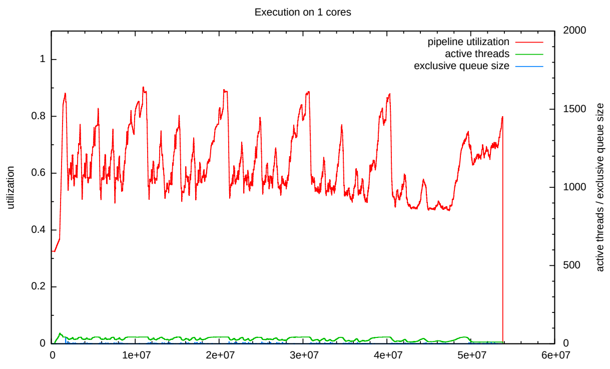
Figure 12 shows the runtime effect of the two reference counting strategies: asyncronous (labelled async) and the bimodal technique (labelled async-norc). As to be expected, the purely asychronous approach does not scale well. Beyond 2 cores, scaling is limited. Even worse, the additional communication overhead due to the increased number of hops on the ring network, combined with the contention on the reference counting queue even negatively impacts runtimes. Execution on 16 or more cores is slower than on a dual-core Microgrid.

With our optimisation in place, the runtime on a single core is slightly higher. We attribute this to the overhead of checking the reference counting mode of data structures during reference counting operations. However, our artificial examples now scales linearly with the number of available cores. As only the data structure referenced by `vect` is reference counted and as this data structure is passed into the data parallel section, no reference counting happens during concurrent execution. Amdahl's law no longer inhibits scaling.

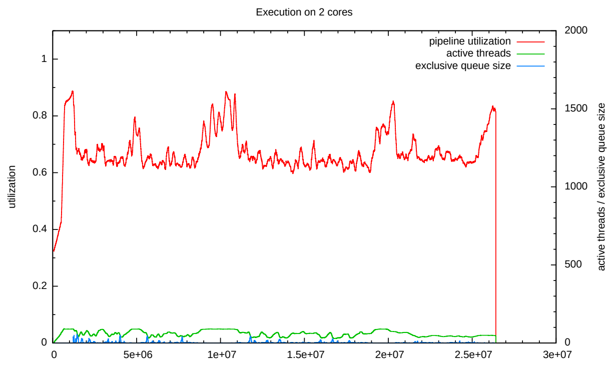
To verify our optimisation in a real world scenario, we have rerun our FFT example using the bi-modal reference counting approach. As Figure 13 shows, our implementation now scales beyond four cores. For 8 cores, we achieve an overall runtime of 8 million cycles. Yet, we still observe a high reference counting load, which from 16 cores onward inhibits further scaling.

A post mortem of the code reveals the cause: To compute the FFT of each vector in the 2 dimensional FFT, we use the Cooley-Tukey algorithm [3] that computes FFT using a divide and conquer strategy. A mayor step in the algorithm is to split the input vector into two vectors holding the elements at odd or even index positions. This two data structures are then passed on into two recursive applications of the algorithm.

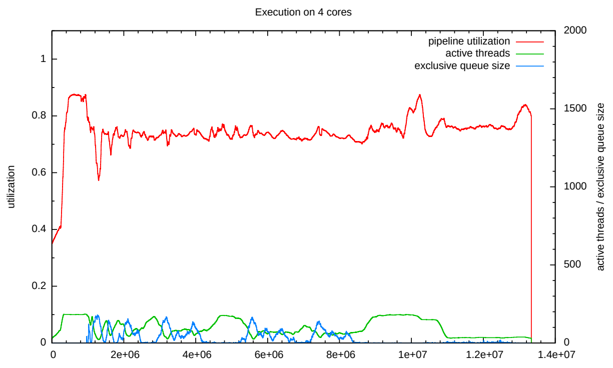
Although we have eliminated the reference counting operations on shared data structures in



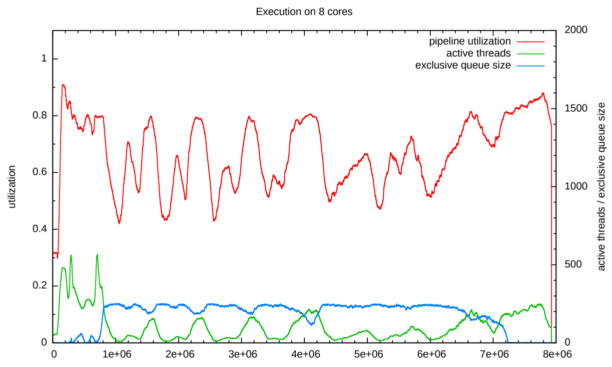
(a) one core



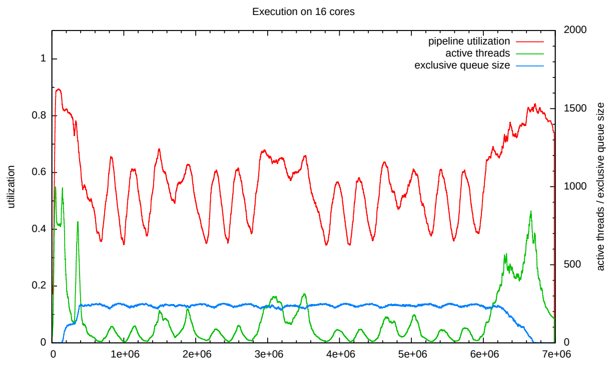
(b) two cores



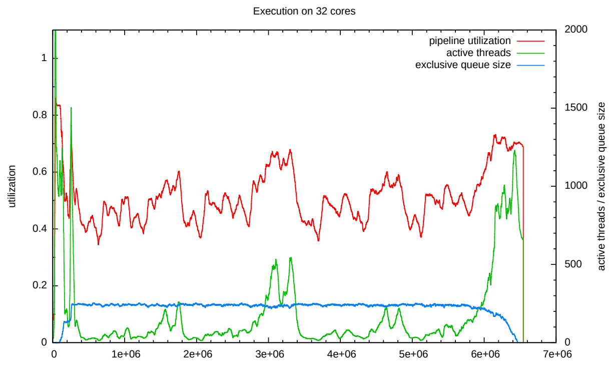
(c) four cores



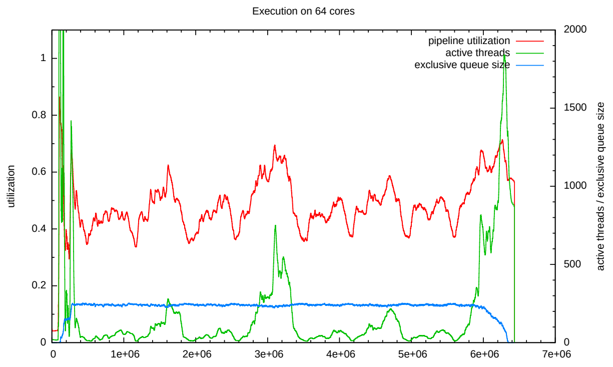
(d) 8 cores



(e) 16 cores



(f) 32 cores



(g) 64 cores

Figure 13: Runtime behaviour of a 2-dimensional 256 element FFT running on varying numbers of cores using bi-modal deferred reference counting.

data-parallel sections, locally allocated data structures still cause reference counting operations to be issued if those data structures are passed into functions. Other than the example shown in Figure 11 that only uses shared data structures, the FFT code passes local data structures into function applications. Thus, in our bi-modal scheme, each function application leads to two asynchronous reference counting operations: One caller increment and the corresponding callee decrement. The operations, furthermore, cannot be elided. Other than for shared data structures, we cannot statically decide on the lifetime of locally allocated data as those may be aliased.

Still, there is potential for optimisation. If function calls are not spawned, we know that they will be executed sequentially on the same core as the calling function was executed. With this knowledge, it safe to use a local reference counter for those data structures, in the same way as we do during sequential compilation of SAC. However, as soon as we spawn a function, all data passed as arguments needs to switch to the asynchronous distributed reference counting style, as otherwise global consistency of the reference counting state cannot be guaranteed.

We have implemented this approach, which we call *tri-modal* reference counting. For all local data, we allocate a local descriptor which holds the reference counting information at runtime. If we enter a data-parallel section, we tag the descriptor to inhibit further reference counting operations. If we encounter a spawn operation, we need to split the reference counting data between the two threads such that each can safely continue. To achieve this, we use a hierarchical approach. We create an asynchronous master counter at the dedicated reference counting resource, which counts the number of local reference counting descriptors a data structure has. Furthermore, we allocate a secondary local reference counting descriptor for the newly spawned thread. The master counter therefore initially has the value two.

To keep the information in sync, we link both local descriptors to the asynchronous master descriptor. The local descriptors are then used for reference counting as before. If, however, the local counter drops to zero, we now have to enquire with the master counter whether further local counters exist. Likewise, to acertain whether data can be updated destructively, we have to ensure that both the local and master counter have a value of one.

If we need to further split a reference counter for which a master already exists, it suffices to increment the master counter and create one copy.

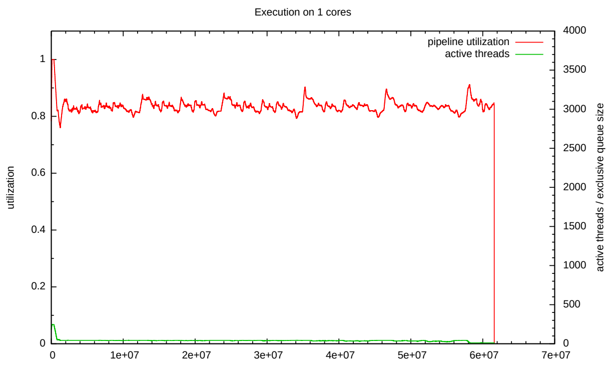
Figure 14 presents the runtime behaviour of FFT when using our tri-modal reference counting approach. As can be seen, the asynchronous reference counting load (blue line) is drastically reduced. Most reference counting is either elided or performed locally. This has a benenficial effect on the overall pipeline utilisation, as well. As we no longer have to enqueue remote reference counting operations, the corresponding delays vanish, too.

Yet, the runtime on a single core has increased from just above 50 million cycles to just over 60 million cycles. We attribute this to management overheads in the tri-modal scheme: For each reference counting operation, we have to identify which of three modes is currently active.

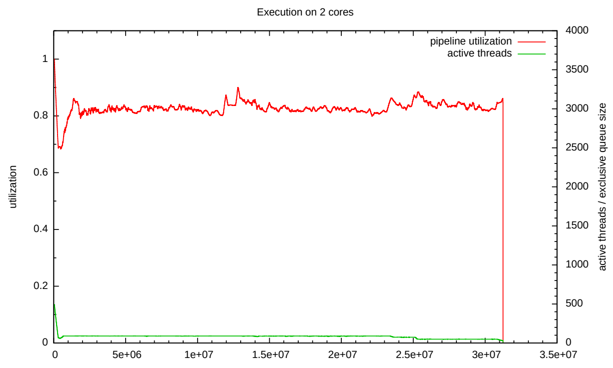
When executing on multiple cores, this overhead is masked by better scalability. Our implementation now scales nearly linearly for up to 32 cores, yielding a total runtime of just above 2 million cycles, which equates to nearly a 30 fold speedup. Ultimately, however, scaling is limited by the size of the workload. Beyond 32 cores, the pipeline utilisation drops as fewer work is available.

Although our tri-modal approach overcomes the limitations of the purely asynchronous approach, there is still room for optimisations. Currently, we need to copy descriptors every time new threads are spawned due to functional concurrency. These allocations, although small in size, put a strain on the heap allocator. The allocation itself, thereby poses less of a challenge. The distributed, lock-free allocator we use copes well with high loads. As it allocates data locally and uses an efficient arena based allocation strategy, allocations of small data structures are cheap. The overhead arises when those data structures are no longer needed. To prevent locking, we use a deferred freeing strategy. When some data is freed that was allocated in a different thread's heap, we only mark the data as garbage. During a consecutive allocation by that thread, we pick up the garbage and free it.

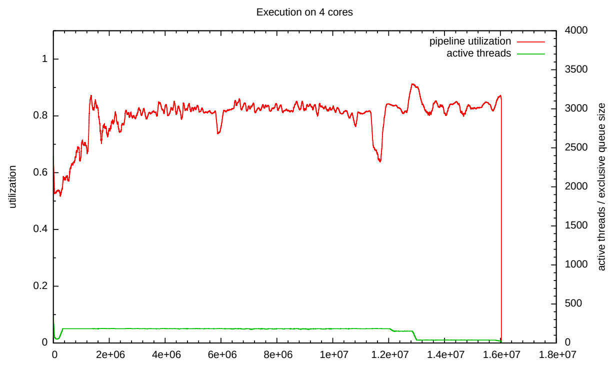
Garbage collection in our approach inevitably requires a traversal of the heap, as we cannot update management structures at free time to avoid locking. The more items have been allocated, the longer this traversal takes. This is where the additional heap manager load hits.



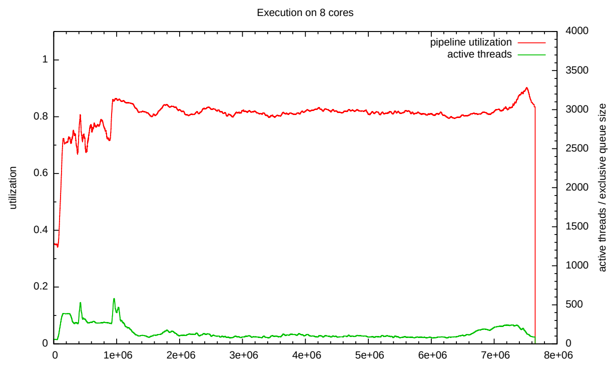
(a) one core



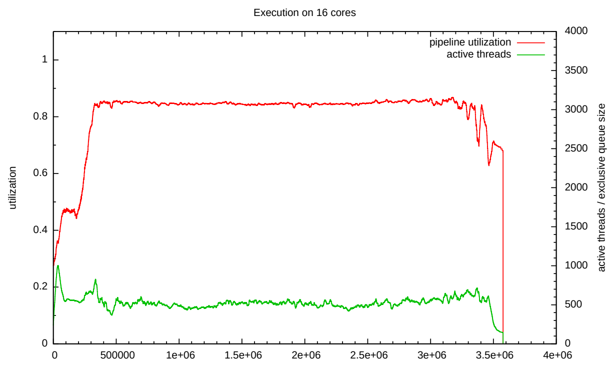
(b) two cores



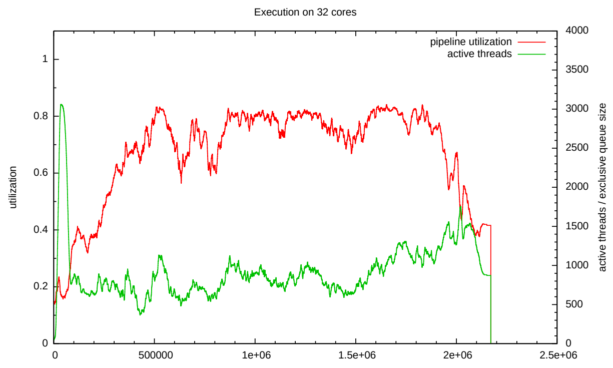
(c) four cores



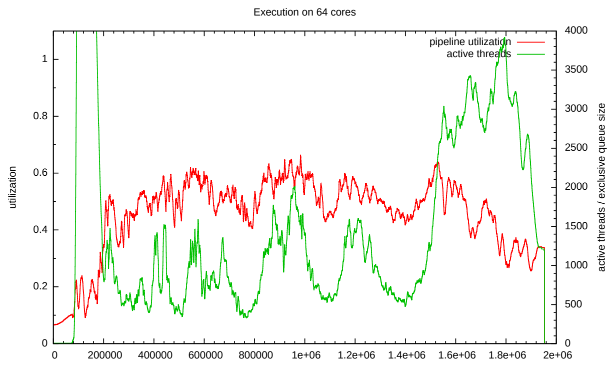
(d) 8 cores



(e) 16 cores



(f) 32 cores



(g) 64 cores

Figure 14: Runtime behaviour of a 2-dimensional 256 element FFT running on varying numbers of cores using multi-modal deferred reference counting.

We have designed an alternative scheme that uses the pointer to the reference counting descriptor for tagging instead of the descriptor, itself. Copying a pointer can be achieved in a local variable or even register and thus has very little runtime overhead. Furthermore, as we only need two bits to enumerate all three states, we only loose a maximum of 8 bytes per descriptor due to the required alignment. Overall, we expect this approach to be very lighweight. Unfortunately, we cannot present runtime results, yet, as our implementation has not yet sufficiently stabilised.

5 Implementation Status

We have implemented all the techniques described in this report and they are available as part of the latest release of the SaC compiler `sac2c` or the latest Microgrid tool chain, respectively. In particular, the latest release includes

1. the lock-free, distributed heap manager with deferred garbage collection with support for thread-local stacks,
2. support for various thread distribution techniques and means to exploit them from SaC programs, and
3. an implementation of the trimodal distributed, non-deferred reference counting.

Whereas the heap manager is enabled by default, thread distribution and reference counting techniques are exposed to the user. Thread distribution can be either influenced by using a target argument to `spawn` operations in SaC programs or using a `sac2c` command line option. Usually the former approach would be chosen to expose algorithm specific distribution strategies to the runtime system. However, if no such sophisticated approach is required, we allow the user to override annotations using the `mutc_force_spawn_flags` command line parameter. It expects as argument a valid SL place expression, which is then used for all `spawn` operations.

The active reference counting technique is configured using the `sac2c` options file `sac2crc`. We have opted for a more persistent configuration in this case as the trimodal reference counting approach, which is enabled by default, is superior in most cases and thus should usually not be changed. We provide this option mainly to allow for experimentation. To change the reference counting mode, the parameter `RC_METHOD` must be set to one of the following values:

local Traditional reference counting assuming locality.

norc No reference counting at all.

async Asynchronous reference counting. RC is done through messages to a seperate core. Re-use and freeing implies synchronisation.

local_norc_desc Switches from local to norc and back. Implies no re-use or freeing of non-local variables in a concurrent context. Implements the mode via a flag in the descriptor.

local_norc_ptr Same as previous but encodes the mode in the lowest bits of the pointer to the descriptor.

async_norc_copy_desc Switches from async to norc and back. Whenever going to norc, the descriptor is being copied.

async_norc_two_descs Same as previous but instead of copying we always allocate two descriptors.

async_norc_ptr Same as previous but instead of having two descriptors. We encode the mode in the pointer.

local_pasync_norc_copy_desc Very similar to local_norc_desc. The only difference is that when spawning a thread we create a copy of the descriptor and use the original one as asynchronous parent.

local_async_norc_ptr Very similar to async_norc_ptr. The only difference is that we start in local and we try to get back there whenever possible.

The full tool-chain is available for download from `http://www.apple-core.info/resources/`. Note that as of this writing, the pointer based referenc counting modes (identified by the *ptr* suffix above) are not yet sufficiently stable for production use. Updates to the SaC tool chain beyond the scope of Apple-CORE are available from the project's website at `http://www.sac-home.org`.

6 Conclusions

During the last year of Apple-CORE, we have concentrated on improving the support for functional concurrency in SaC and its interplay with the existing data-parallel framework. We came to realise that functional concurrency due to its lack of structure requires different support, both in hard- and software, than the data-parallel concurrency we have studied first. In particular, we had to make major investments into an efficient runtime system for massively parallel architectures, which we believe applies well beyond the Microgrid.

As our results show [11], these investments have paid off. We achieve significant performance improvements over the old runtime system and hardware implementation. Due to our simulation based infrastructure, we were only able to investigate small kernels and benchmarks. Scaling this up to full applications would yield further insight and is expected to expose further optimisation potential.

Another realisation is the power that hardware/software co-design brings to language designers and compiler writers. The solutions we propose in this report, i.e., the heap manager, the non-deferred reference counting approach and the scheduling and mapping strategies, all required changes to the hardware. Either these changes enabled the approach in the first place (heap manager) or greatly enhanced its performance (reference counting). This great power, however, comes at a price. Implementing the heap manager required changes to the entire tool chain, from the simulator, via runtime libraries, all the way to the high-level languages. The same is true for the reference counting approach. This has bound more development resources than we had anticipated.

The loser in the struggle for resources, in the end, unfortunately was the research on automatic extraction of functional concurrency from existing SaC programs. Our current solution requires explicit programmer annotations. We still believe in the power of automated solutions and that such solutions are feasible. Having a manual solution in place, research towards an automated one is now under way. Finding a solution, however, will remain a task outside of the scope Apple-CORE.

References

[1] G. M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings of AFIPS*, volume 30, page 483, Washington, DC, USA, 1967.

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

[3] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex fourier series. *Math. Comp.*, (19):297–301, 1965.

[4] Clemens Grelck, Stephan Herhut, Chris Jesshope, Carl Joslin, Mike Lankamp, Sven-Bodo Scholz, and Alex Shafarenko. Compiling the Functional Data-Parallel Language sac for Microgrids of Self-Adaptive Virtual Processors. In *14th Workshop on Compilers for Parallel Computing (CPC'09), IBM Research Center, Zurich, Switzerland*, 2009.

[5] Clemens Grelck and Kai Trojahner. Implicit Memory Management for SaC. In Clemens Grelck and Frank 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.

[6] Stephan Herhut, Carl Joslin, and Sven-Bodo Scholz. Functional concurrency on the microgrid using single assignment c – a status report. Technical report, Deliverable 4.3 of the EU Framework 7 Apple-CORE project, 2010.

[7] Stephan Herhut, Carl Joslin, and Sven-Bodo Scholz. Thread-Local Stacks, a Light-Weight Alternative to Thread-Local Heaps. In *15th Workshop on Compilers for Parallel Computing (CPC'10)*. Vienna University of Technology, Vienna, Austria, 2010.

[8] Stephan Herhut, Carl Joslin, Sven-Bodo Scholz, Raphael Poss, and Clemens Grelck. Concurrent Non-Deferred Reference Counting on the Microgrid: First Experiences. In J. Haage and M. Morazán, editors, *22nd International Symposium on Implementation and Application of Functional Languages (IFL'10), Alphen a/d Rijn, Netherlands, Revised Selected Papers*, volume 6647 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Heidelberg, New York, 2011. to appear.

[9] Raphael Poss and Mike Lankamp. Family allocation and places. `https://notes.svp-home.org/mgsim12.pdf`.

[10] Raphael Poss, Mike Lankamp, Thomas Bernard, and C.R. Jesshope. SL language reference. `https://mac-chris.science.uva.nl/csa/notes/www/book5.pdf`.

[11] Daniel Rolls and Chris Jesshope. Final report of benchmark evaluations in different programming paradigms. Technical report, Deliverable 2.3 of the EU Framework 7 Apple-CORE project, 2010.

[12] S.-B. Scholz. Single Assignment C — efficient support for high-level array operations in a functional setting. *Journal of Functional Programming*, 13(6):1005–1059, 2003.

[13] Sven-Bodo Scholz, Stephan Herhut, and Carl Joslin. Implementation of a first SaC to μTC compiler. `http://www.apple-core.info/wp-content/apple-core/2008/01/d42.pdf`. Apple-CORE Deliverable D4.2.