Folding of Tagged Single Assignment Values for Memory-Efficient Parallelism

Dragoş Sbîrlea[1], Kathleen Knobe[2], and Vivek Sarkar[1]

[1] Department of Computer Science, Rice University
{dragos,vsarkar}@rice.edu
[2] Intel Corporation
kath.knobe@intel.com

Abstract. The dynamic-single-assignment property for shared data accesses can establish data race freedom and determinism in parallel programs. However, memory management is a well known challenge in making dynamic-single-assignment practical, especially when objects can be accessed through tags that can be computed by any step.

In this paper, we propose a new memory management approach based on user-specified *folding functions* that map logical dynamic-single - assignment (DSA) tags into dynamic-multiple-assignment (DMA) tags. We also compare folding with *get-counts*, an approach in which the user specifies a reference count for each single-assignment value. The context for our work is parallel programming models in which shared data accesses are coordinated by put/get operations on tagged DSA data structures. These models include dataflow programs with I-structures, functional subsets of parallel programs based on tuple spaces (notably, Linda), and programs written in the Concurrent Collections (CnC) coordination language. Our conclusion, based on experimental evaluation of five CnC programs, is that folding and get-counts can offer significant memory efficiency improvements, and that folding can handle cases that the get-counts cannot.

1 Introduction

The multicore revolution has increased the urgency for developing programming models that deliver scalable parallelism with minimal effort by programmers. The use of shared data structures by parallel tasks has proved to be a two-edged sword in pursuing this goal. On the one hand, a shared address space can reduce the semantic gap between a sequential program and its parallel version. On the other, uncoordinated accesses to shared data structures are a notorious source of bugs that arise from data races and other sources of nondeterminism leading to the *programmability wall*.

One approach to addressing the drawbacks of shared data structures is to enforce a *dynamic-single-assignment property* for shared data accesses, since it in turn can establish data race freedom and determinism in parallel programs. Thus, the context for our work is parallel programming models for multicore and many-core processors in which all shared data accesses are performed through put/get

C. Kaklamanis et al. (Eds.): Euro-Par 2012, LNCS 7484, pp. 601–613, 2012.

operations on dynamic-single-assignment data structures indexed using associative tags (keys). These models include dataflow programs with I-structures [1], functional subsets of parallel programs based on tuple spaces (notably, Linda [7]), and programs written in the Concurrent Collections (CnC) coordination language [3].

However, past experiences with implementations of functional languages have shown that memory management can be challenging with the dynamic-single-assignment property. It becomes even more challenging when objects can be accessed through user-computable tags, since standard reference-based garbage collection cannot be applied in that case. In this paper, we propose a new memory management approach based on user-specified *folding functions* that map logical dynamic-single-assignment (DSA) tags into dynamic-multiple-assignment (DMA) tags. We also compare folding with *get-counts*, an approach in which the user supplies a function that maps tags to integers indicating the number of gets that will occur on the item Both approaches are *fail-safe* i.e., an exception is thrown if the program performs accesses that are inconsistent with the folding functions or get-counts.

There has been a lot of past work focused on converting a multiple-assignment program to dynamic single assignment form so as to simplify program optimization and transformation. An early paper [6] described several applications of dynamic single assignment, such as conversion of a program to a set of recurrence equations, scalar expansion, array expansion [5], program verification and parallel program construction. In contrast, folding addresses the dual problem of converting a dynamic single assignment program to multiple-assignment form with reduced memory requirements. Based on the well known challenges in transforming static single assignment form to multiple assignment form [2], it is natural to expect that translating out of dynamic single assignment form will be a challenging problem too, especially when the original non-DSA program is unavailable. To the best of our knowledge, this paper is the first to propose a user-specified "folding" approach to address this problem.

In summary, this paper includes the following contributions:

- *Basic folding* (Section 2.1), a novel memory management technique for accesses to associative dynamic-single-assignment data structures (item collections). This technique relies on user-specified folding functions with fail-safe checks for correctness at runtime.
- *Update-in-place memory reuse* (Section 2.2), an extension that allows the user to specify *GetForUpdate* operations that allow an input item to be rewritten as an output. This approach can be used both with folding functions and get-counts, and includes fail-safe checks as well.
- *Extended folding* (Section 2.5), an extension to basic folding for items that are written but never read.
- *A design and implementation* (Section 3) of the above folding and get-count techniques for the CnC model.
- *Empirical results* (Section 4) that show that folding and get-counts can offer significant improvements in memory efficiency over the baseline version without these techniques.

2 Folding of Dynamic Single Assignment Values

2.1 Basic Folding

The intuition behind folding is as follows: if we know that two values have non-overlapping lifetimes, we can assign them to the same physical storage thereby reducing the maximum memory requirement for the application. Following the terminology used in the CnC model, we refer to the associative dynamic-single-assignment (DSA) data structures assumed in this work as *item collections*, to keys as *tags*, values as *items*, and computational tasks as *steps*. The two operations supported by item collections are *put(tag, item)* and *get(tag)*. The DSA property requires that dynamically at most one *put()* operation be performed for a given tag. Further, each *get()* operation is assumed to be blocking i.e., it only returns a value after a *put()* operation has been performed with that tag.

Definition 1 (Folding function). *A folding function f transforms a* logical tag t_1 *to a* physical tag, $f(t_1)$. *Thus, the logical* put(t_1, i_1) *operation is transformed into a* physical put(t_1, f(t_1), i_1) *operation, where $f(t_1)$ is the physical location used to store the item and the original tag t_1 is stored as an auxiliary value. Likewise, the logical* get(t_1) *operation is transformed into a* physical get(t_1, f(t_1)) *operation.*

Thus, the folding function maps DSA tags to dynamic multiple assignment (DMA) tags which are associative indices into a physical store. When a new item i_2 is mapped to the same physical store location as a previous item i_1 (because $f(t_1) = f(t_2)$), the space of i_1 is freed. Example executions of a program that computes the n-th Fibonacci element are in Figures 1 and 2 (without and with folding, respectively). Item n can fold over item $n-2$. The folding function used is: $fold(n) = (n+1)\%2 + 1$.

This use of a folding function is called basic folding. As discussed later in Section 2.3, a runtime error may be thrown if the folding function is specified incorrectly, but a *get()* operation will never return an incorrect logical value.

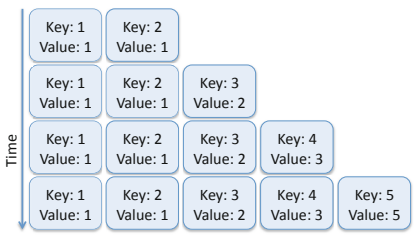

Fig. 1. Item collection content for a baseline execution of Fibonacci

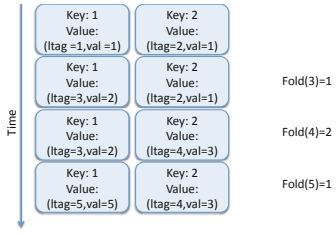

Fig. 2. Item collection content for a folding execution of Fibonacci

We now identify the conditions under which folding is legal. As an example, consider the following sequence of logical *get()* and *put()* operations: "*put*(t_1, i_1); *get*(t_1); *put*(t_2, i_2); *get*(t_1)". In this case, it would be illegal to fold items i_1 and i_2 on the same location because they have interfering live ranges [11]. To ensure safety for folding two items, they must have disjoint lifetimes in any possible schedule of the program.

Definition 2 (Item lifetime). *The* lifetime of an item *in a program execution is the interval between the execution point at which the item is produced by a* put() *operation and the execution point of the last* get() *operation performed on the item. If there are no* get() *operations, the lifetime begins and ends at the* put().

Definition 3 (Legal program). *A legal program is one that always completes execution with all* get() *operations having successfully completed, for all possible schedules.*

Definition 4 (Correct folding transformation). *A folding transformation for a legal program P specified by folding function f is correct if, for every input I, an execution of P with input I and folding function f is also legal (no blocked gets()) and results in the same result for each* get() *operation as the original execution of program P without folding.*

Theorem 1 (Folding correctness requirement). *For a folding transformation of a legal program to be correct, the folding function must not fold together any two items whose lifetimes may overlap.* [Proof omitted due to space limitations.]

2.2 Folding with Update-in-Place Memory Reuse

Basic folding ensures that memory can be reclaimed after the end of a computational step that performs the last logical *get()* operation on an item. However, many steps have the following computational structure: "i_1 = *get*(t_1); *allocate* i_2; i_2.*set*($G(i_1)$); *put*(t_2, i_2);". With basic folding, both i_1 and i_2 will be assumed to be simultaneously live and will contribute to the maximum memory requirement for the program. However, if function G can be implemented as an *update-in-place* function, then i_1's storage can be reused for i_2 if get(t_1) is the last get operation performed with logical tag t_1. To enable this optimization, we allow the user to use a *getForUpdate()* operation instead of *get()*, as an indication that this is the last *get()* operation for the given tag in any schedule, thereby making it possible for item i_1 to be updated in place to obtain item i_2. Figure 3 is an example. As with the folding function, the correctness of a *getForUpdate()* operation will also be checked at runtime so as to guarantee fail-safe behavior (see Section 2.3).

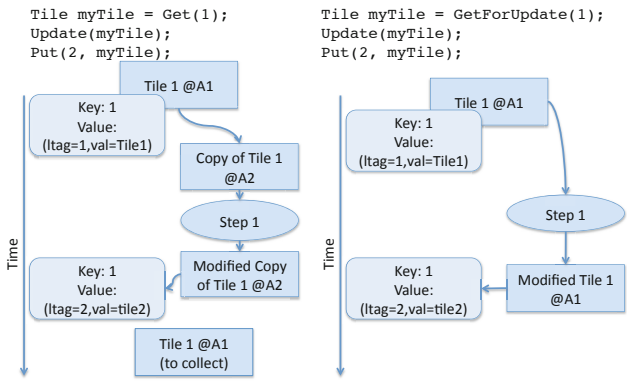

Fig. 3. Left: With a get() call, the item memory is copied before being returned to the step, which can modify it and put() it with some other tag. This leaves the old item memory to be collected when an item folds over its entry in the store. Right: With getForUpdate, the copy is not performed and no memory will need to be collected, as it is reused by the new item.

2.3 Error Detection

The folding error detection mechanisms are based on the assumption that the original program is legal (Definition 3) without the folding optimization. We define any behavior of a legal program in the presence of folding that differs from the behavior of a non-folded execution as an error.

For example, a *get()* that returns an incorrect value would constitute an error. This could happen, if the content of the physical store location corresponding to a particular tag is returned without checking that the logical tag of the item in that location corresponds to the logical tag of the item we are trying to get. If the item in the store does not have the same logical tag, we need to wait for it to be produced. However, if the item was previously produced and some other item was erroneously folded over it, we will never find the item. Without an error checking mechanism, the program may finish with blocked steps instead of the correct non-folded behavior.

To enable detection of such errors, we define a debug mode for folding, in which a boolean flag is stored for each tag that is *put()* during execution. Using this flag, we can differentiate between items that are not present in the physical store because some other item was folded over them and items that have not been produced as yet. A *get()* performed on previously overwritten items should throw an exception reporting an incorrect folding function, but a *get()* should block until the item is produced if that is not the case. In debug mode the system also detects dynamic single assignment violations (on every put, if the boolean flag for that logical tag was previously *put()*, we report an exception) with or without the presence of folding.

2.4 Programmability Benefits of Folding

To illustrate the benefits of folding with error detection, consider a common technique used by performance-oriented C programmers where storage is reused instead of calling free() followed by malloc(). This approach can be especially error-prone for parallel programs, because the overlap in lifetime between the initial and subsequent values may be schedule-dependent. With folding, a similar reuse of memory could be achieved in a fail-safe manner by folding the two logical items and using the getForUpdate mechanism for memory reuse.

As a concrete example, consider the classic two-buffer approach used by iterative algorithms in which one buffer is used as an input and the other as the output, and their roles are swapped in each sequential iteration. With our folding approach, the programmer can think in terms of allocating a new DSA output buffer in each iteration, and a folding function can effectively perform the swap. This approach was used in our implementation of a Routing simulation application (see Section 4) where the routing tables for one iteration are built using the routing tables of the previous one, and a folding function was specified as follows:

```
public final Object fold(point tag) {
   int i, j, k; //i: node id, j: iteration id; k: repetition #
   i = p.get(0); j = p.get(1); k = p.get(2);
   return new point(i, j%2, k);
}
```

2.5 Extended Folding: Folding with Ordering

Items with empty lifetimes pose an interesting research challenge for folding. Consider a program that expects to produce and consume items in order as follows: "*Step1: [put(t_1, i_1)] Step2:[get(t_1);put(t_2, i_2)] Step3:[get(t_2); put(t_3, i_3)] Step4:[get(t_3)]* ". In such a case, it might seem reasonable to fold t_1, t_2, and t_3 to the same physical location. However, if (say) *get(t_2)* is not performed for some reason, there is no way (if using blocking-get synchronization only) to ensure that *put(t_2, i_2)* completes before *put(t_3, i_3)*, thereby making the folding incorrect (because *get(t_3)* may never find t_3 as it has been folded over).

This is an instance of the more general problem caused by optional *get()* calls but in this particular case there is a way to solve the problem. We propose an extension to folding that allows folding of items that may never be consumed. Such items can appear when control dependent gets are used, for example with short-circuit boolean operations such as "get(t_1) && get(t_2)". We observe that items that are never read have an empty lifetime and can be optimized away from the physical store. However, this may not be known at the time of the *put()* operation, but may be known when a subsequent *put()* is performed on the same physical location.

We can express this by allowing the presence of an additional user function that acts like a "compare age" operation. If an item that is being put maps to

a physical location where another item resides and should be declared dead, the function returns *true* ("newer"), and the new item is stored. Otherwise, if the new item is known to never be read, it returns returns *false* ("older"), the incoming item is not stored and the old item is retained.

To perform the age comparison, the function needs two parameters: the tag of the item being put currently and the tag of the old item that exists in the location in the physical store where the new item would be inserted. The programmer has to identify if the tag of the current item in the item collection means that all of the steps that could access the incoming item have executed and did not access the incoming item. If this is the case, then the incoming item can safely be discarded. The Rician Denoising benchmark (see Section 4) uses this extension.

3 Implementation

We have implemented folding as an extension to the Habanero Java CnC runtime [3]. The Java key-value data structure used to implement item collections is now indexed by DMA tags instead of DSA tags. When an item is put() with DSA tag t_1 its corresponding DMA location in the store is determined by identifying $pt_1 = f(t_1)$, where f is the folding function. Then, the physical store is accessed to see if there is any entry at that physical location. If there is none, we create it, and label it with the logical tag t_1. If there is, we need to hold a lock on the physical store location while the following operations are performed. First, we update the logical tag of the physical store entry to the logical tag of the item that has just been put. Then, we go through the list of steps waiting on that particular physical store location and, for each step that is waiting for the current item mark it as ready for execution. The marked marked continue their execution by performing a *get()* that will succeed because the desired item is already in the physical store.

When a get() on item with DSA tag t_1 is performed, its DMA tag is determined by identifying $pt_1 = f(t_1)$. If the entry does not exist, it is created, inserted in the physical store and the step is added to its list of waiting steps. If the entry does not correspond to the logical tag of the item, it registers itself to wait also. Compared to a non-folding execution, the only extras step needed for insertion is the application of the folding function (which does not need synchronization and has minimal overhead). The bigger overhead is in the *put()* , where the list of waiting steps has to be checked linearly to unblock only the steps that are waiting for the new item and this happens while holding the lock. We chose to have the overhead in the *put()* and not *get()* as the *get()* is usually performed multiple times on a single item and our approach leads to less contention.

Both the get-counts and folding policies only remove items from item collections, so that there is no object reference pointing to them; the Java garbage collection subsequently reclaims the memory.

4 Results

The following results were obtained on a 16 core Xeon system with 16GB RAM, running Habanero Java implementation of Concurrent Collections [3] on a 64 bit Java 1.6, using 16 workers for the work-stealing CnC runtime and Java default garbage collection mechanism. In this section we compare the performance and memory footprint of the following CnC memory management policies:

1. *Baseline*: non-collecting CnC (items are never removed from item collections) leading to memory leaks, but also no folding overhead.
2. *Get-counts*: memory management in which the user specifies a reference count for selected items, the count is decremented on every get() operation on a specified item, and the item is freed when the count becomes zero.
3. *Folding*: the folding runtime described in Section 3. We used the ordering extension described in Section 2.5 as needed and the tables contain the "Ordered" specifier where this happened.

For each policy used, we obtained the following measurements:

1. Execution Time - We performed thirty repetitions of the program in the same JVM instance, and reported the average, as advocated in [8].
2. Memory at end - the program footprint after the CnC graph finishes execution. With this metric, get-counts has an advantage because it removes items immediately, where as folding waits for the birth of another item, so at the end folding usually has more live items. In contrast, folding saves some work by taking a lazy approach to freeing items.
3. Items at end - similar to the previous metric, but expressed in items.

We evaluated the impact of folding and get-counts on the following applications:

1. *Microbenchmark* showing the difference in scalability between get-counts and folding with the number of reads per item.
2. *N-body simulation* for performance analysis.
3. *Routing simulation* as an application in which get-counts might lead to leaks because items have a number of accesses unknown at creation time, but folding works without needing the Ordered extension.
4. *Rician denoising* as example of an application in which folding with ordering can safely be used, but get-counts leads to leaks because some items have data-dependent accesses whose number is unknown.
5. *Cholesky factorization* as an example of memory reuse via the getForUpdate optimization.

Microbenchmark: Scalability with read/write ratio This benchmark varies the reads to write ratio to analyze the performance of the two collection mechanisms. Because folding performs most of the synchronization on *put()* as opposed to get-counts, which performs most of the synchronization on *get()*, we checked if the best performing policy might be get-counts for low read/write ratio. However, as shown in Figure 4, the folding version runs faster than both get-counts and baseline CnC even for a ratio of 1. Some applications may have a read/write ratio lower than one; performance for this case is analysed later using the Rician Denoising application.

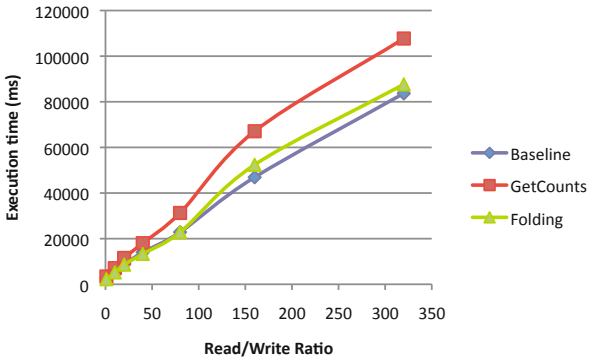

Fig. 4. Performance with read/write ratio (16 core Xeon)

N-Body Simulation. We implemented the O(N^2) algorithm for N-body simulation with both get-counts and folding and the results are shown in Table 1. The folding policy performs well because this benchmark has a small step granularity, large number of items and thus more contention on the item collections. The fact that folding has less synchronization of gets (in this application there are 10 gets per item) leads to a consistent (1.3×) performance improvement compared to get-counts. The get-counts footprint is smaller because folding can only reduce the footprint to the maximum footprint of the program during its execution, and in this case, that footprint is 20 items, which is also the maximum theoretical footprint for get-counts.

Table 1. Experimental results for NBody (5 bodies, 100000 timesteps)

CnC policy	Time (s)	Memory at end (bytes)	(items)
Baseline	16.9	277.0 MB	1,000,005
Get-Counts	18.1	3.6 KB	10
Folding	**13.0**	7.0 KB	20

Table 2. Experimental results for Routing, with reliable links

CnC policy	Time (s)	Memory at end (bytes)	(items)
Baseline	**21**	61.0MB	102000
Get-Counts	25	10.7KB	1000
Folding	**21**	1.3MB	2000

Routing Simulation The routing simulation benchmark has unknown number of gets on each item, making it a challenge for the get-counts approach. It simulates the convergence of min-distance routing protocols such as IS-IS [10] and OPSF [9]. As links might go down, when a routing table is being built, we cannot know how many gets will be performed on that node. In such cases, the get-count will never reach zero and the item will become a memory leak. To see how the number of leaked items varies with the chance of links failing we varied the chance of a message not getting through from 0 to 10%, as shown in Figure 5: at only 1% failure rate half the items are leaked. Even in the absence of link failure, folding shows a 16% performance improvement over get-counts (Table 2).

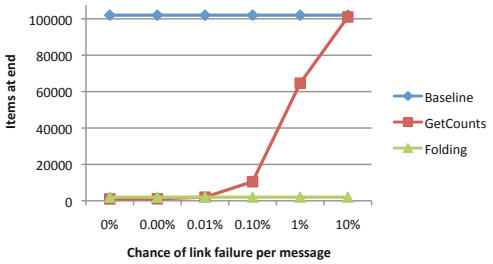

Fig. 5. Relation of link fail rate and memory leaks

Table 3. Performance comparison for Rician Denoising: image size 2560*1280, tile size 128*64. Shortcircuit reductions DISABLED (top) and ENABLED (bottom)

Shortcircuit operators	CnC policy	Time (s)	Memory at end (MB)	(Items in each collection)				
				Image	Gradient	Image× Gradient	Factor	Convergence Status
Enabled	Baseline	6.5	2888	10800	10400	10400	10400	10400
	Get-Counts	**2.6**	740	800	0	0	0	9897
	Folding (Ordered)	**2.6**	800	1200	800	800	800	800
Disabled	Baseline	8.1	2888	108006	10400	10400	10400	10400
	Get-Counts	3.7	720	800	0	0	0	0
	Folding	**3.5**	830	1200	800	800	800	800

Rician Denoising Rician (Poisson) denoising is an image processing application. Its global convergence check is a reduction on the convergence status of all tiles and it is sped up using a short-circuit evaluation: if a single tile changes significantly we do not need to wait for the convergence condition of all the other tiles to be evaluated, we immediately know we will need an additional iteration and can start spawning the corresponding steps.

The results (Table 3) show the performance of get-counts and folding: folding offers the best performance. Furthermore, get-counts leads to leaks of items from the ConvergenceStatus item collection in which the operands of the short-circuit operators are stored (the cause of the leaks is the unknown number of gets): 95% of items stored in that item collection are leaked, totaling 20MB. However, without short-circuit operators, get-counts collects more items because at the end of the program, all the items stored in the item collections that store intermediate results (Gradient, ImageTimesGradient, etc) can be collected. This does not affect the actual high water-mark of the program which is the same in both folding and get-counts executions.

Cholesky Factorization Cholesky factorization is a numerical application whose input is a symmetrical positive-definite matrix and output a lower-triangular matrix. One possible CnC implementation was previously described and benchmarked in [4] and the results were encouraging.

Table 4 shows that the proposed update-in-place optimization, if applied on either get-counts or folding, can lead to a large performance increase. Using getForUpdate leads to a performance improvement between 10% and 20% for both collecting policies. Baseline CnC cannot safely apply this optimization without additional programmer input to ensure that whenever GetForUpdate is called, the item accessed is indeed dead. To work around this, we manually added this call only when such accesses are safe.

Table 4. Performance comparison for Cholesky factorization (125*125 tiles)

Input Size	CnC Policy	Without update-in-place			With update-in-place		
		Time (s)	Item collection memory (MB at end)	(items at end)	Time (s)	Item collection memory (MB at end)	(items at end)
2000	Baseline	**0.9**	142.2	952	0.8	33.7	952
	Get-Counts	**0.9**	33.7	272	0.8	33.7	272
	Folding	**0.9**	33.7	272	**0.7**	33.7	272
4000	Baseline	7.3	1008.5	6512	5.6	133.2	6512
	Get-Counts	**6.2**	133.2	1056	5.6	133.2	1056
	Folding	**6.2**	133.2	1056	**5.0**	133.2	1056
6000	Baseline	26.6	2680.2	20776	19.5	298.4	20776
	Get-Counts	22.3	298.4	2352	19.2	298.4	2352
	Folding	**21.6**	298.4	2352	**19.1**	298.4	2352

Memory High-watermark comparison Table 5 shows the maximum number of live items during the execution of the benchmarks. This metric shows, in the schedules and with the parallelism actually used during execution, what is the maximum number of items that were live - the memory "high-water mark" of the program. To obtain these values we used atomic counters that tracked the number of stored items. The results show that maximum live items number

Table 5. Maximum number of items live during execution

Benchmark		**Baseline**	**Get-Counts**	**Folding**
Nbody		1,000,005	19	20
Routing		102000	1100	2000
RicianDenoising	Image	10800	800	800
	Gradient	10400	27	800
	Image × Gradient	10400	26	800
	ConvergenceStatus	10400	9897	800
Cholesky (6000)		20776	2352	2352

is lower than the bound identified by folding. However, in the future, as the number of processors grows, more tasks will run concurrently and the number of live items will increase.

5 Conclusions and Future Work

In this paper, we introduced a new memory management approach based on user-specified *folding functions* that map logical dynamic-single-assignment (DSA) tags into dynamic-multiple-assignment (DMA) tags, while preserving semantic guarantees of data race freedom and determinism. Our approach is applicable to parallel programming models in which shared data accesses are coordinated by put/get operations on tagged DSA data structures. These models include dataflow programs with I-structures, functional subsets of parallel programs based on tuple spaces (notably, Linda), and programs written in the Intel Concurrent Collections (CnC) coordination language. Our conclusion, based on experimental evaluation of five CnC programs, is that folding can offer significant memory efficiency improvements, and that folding can handle cases that get-counts (an alternative approach to user-specified memory management) cannot. An interesting direction for future work is automatic generation of folding functions. In many of the benchmarks that we studied, it is possible to use static analysis of get and put function parameters to identify candidates for folding.

Acknowledgments. We are grateful to the Intel Concurrent Collection team, in particular Frank Schlimbach, James Brodman and Ryan Newton (now at Indiana University), for proposing the Get-Counts idea and for having stimulating discussions. We thank Shams Imam for his debugging help and thorough feedback and the reviewers for their helpful comments.

References

1. Arvind, Nikhil, R.S., Pingali, K.K.: I-structures: data structures for parallel computing. ACM Trans. Program. Lang. Syst. 11 (October 1989)
2. Boissinot, B., Darte, A., Rastello, F., de Dinechin, B.D., Guillon, C.: Revisiting out-of-ssa translation for correctness, code quality and efficiency. In: CGO 2009, Washington, DC, USA, pp. 114–125 (2009)
3. Budimlic, Z., Burke, M., Cavè, V., Knobe, K., Lowney, G., Newton, R., Palsberg, J., Peixotto, D., Sarkar, V., Schlimbach, F., Tasirlar, S.: Concurrent collections. Scientific Programming (2010)
4. Chandramowlishwaran, A., Knobe, K., Vuduc, R.: Performance evaluation of concurrent collections on high-performance multicore systems. In: IPDPS (2010)
5. Feautrier, P.: Array expansion. In: Proceedings of the 2nd International Conference on Supercomputing, ICS, pp. 429–441. ACM, New York (1988)
6. Feautrier, P.: Dataflow analysis of array and scalar references. International Journal of Parallel Programming 20(1), 23–51 (1991)
7. Gelernter, D.: Generative communication in linda. ACM Trans. Program. Lang. Syst. 7, 80–112 (1985)

8. Georges, A., Buytaert, D., Eeckhout, L.: Statistically rigorous java performance evaluation. In: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications (2007)
9. Moy, J.: OSPF Version 2. RFC 2178, Obsoleted by RFC 2328 (July 1997)
10. Oran, D.: Osi is-is intra-domain routing protocol. RFC 1142 (February 1990)
11. Torczon, L., Cooper, K.: Engineering A Compiler, 2nd edn. Morgan Kaufmann Publishers Inc., San Francisco (2011)