Reducing Compiler-Inserted Instrumentation in Unified-Parallel-C Code Generation

Michail Alvanosl†, José Nelson Amaral‡, Ettore Tiotto‡, Montse Farreras§, Xavier Martorell§

†Department of Computer Science, University of Alberta, Edmonton, Canada. Email: amaral@cs.ualberta.ca
‡Static Compilation Technology, IBM Toronto Laboratory, Markham, Canada. Email: etiotto@ca.ibm.com
§Department of Computer Architecture, Universitat Politècnica de Catalunya, Email: {mfarrera,xavim}@ac.upc.edu

ABSTRACT

Programs written in Partitioned Global Address Space (PGAS) languages can access any location of the entire address space via standard read/write operations. However, the compiler have to create the communication mechanisms and the runtime system to use synchronization primitives to ensure the correct execution of the programs. However, PGAS programs may have fine-grained shared accesses that lead to performance degradation. One solution is to use the inspector-executor technique to determine which accesses are indeed remote and which accesses may be coalesced in larger remote access operations. A straightforward implementation of the inspector-executor in a PGAS system may result in excessive instrumentation that hinders performance. This paper introduces a shared-data localization transformation based on linear memory descriptors (LMADs) that reduces the amount of instrumentation introduced by the compiler into programs written in the UPC language and describes a prototype implementation of the proposed transformation. A performance evaluation, using up to 2048 cores of a POWER 775 supercomputer, allows for a prediction that applications with regular accesses can achieve up to 180% of the performance of hand-optimized versions while applications with irregular accesses yield performance gain from 1.12X up to 6.3X speedup.

I. INTRODUCTION

New parallel languages and programming models provide simpler means to develop applications that can run on parallel systems without sacrificing performance. Partitioned Global Address Space (PGAS) languages [1], [2], [3], [4], [5], [6] extend existing languages with constructs to express parallelism and data distribution. These languages provide a shared-memory-like programming model, where the address space is partitioned, and the programmer has control over the data layout. Unified Parallel C (UPC) [1], an extension of the C programming language, follows the PGAS programming model.

PGAS languages offer the advantage of fast development of parallel applications, in comparison with the Message Passing Interface (MPI) [7], through the use of a shared memory abstraction. These programs may contain fine-grained shared accesses that lead to performance degradation. Often, after initial coding, the programmer tunes the source code to produce a more scalable version. However, the reality is that, at the end of this tuning for performance, the PGAS code resembles its MPI equivalent. Therefore, the performance tuning often nullifies the ease-of-coding and ease-of-maintenance advantages of PGAS languages.

A solution to improve the performance of fine-grained communication is to apply compiler and runtime optimizations to reduce the cost of inter-node communication. Researchers proposed different methods to optimize the fine-grained communication in PGAS languages, such as inspector-executor transformation [8], [9], [10], static coalescing [11], [12], [13], limited privatization [14], [15], and software caching [16], [17]. However, a big hurdle in the code generation of UPC language is that the compiler ends up inserting runtime calls to transform UPC “shared” accesses into requests for data (or actions) to other address partitions. Thus, an important question to answer is: how can we achieve performance comparable to C or MPI using the UPC programming model? Despite the great work done both with High Performance Fortran and UPC, today no compiler delivers acceptable performance in the case of fine-grained communication. Table I presents the execution time between serial C and UPC versions running with one UPC thread without the inspector-executor technique. The compiler inserts a runtime call on each shared access when no optimization is applicable.

The focus of this paper is to explore ways for removing the instrumentation code created by the inspector-executor transformation. The inspector-executor transformation is a powerful optimization that can improve the performance of fine-grained communication in PGAS languages in orders of magnitude. This is the first paper that focus on solving the problem of excessive instrumentation in PGAS languages. The main contributions of this paper are:

- A shared-data localization transformation based on Constant-Stride Linear Memory Descriptors (CSLMADs). This transformation improves, the performance of programs containing fine-grained

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Dataset</th>
<th>Serial C</th>
<th>UPC Single-Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sobel [18]</td>
<td>64K x 64K Pic</td>
<td>92.1</td>
<td>110.1</td>
</tr>
<tr>
<td>Fish [19]</td>
<td>16K Objects</td>
<td>155.7</td>
<td>727.7</td>
</tr>
<tr>
<td>Guppie [20]</td>
<td>2^{28} Elements</td>
<td>70.5</td>
<td>349.6</td>
</tr>
</tbody>
</table>

TABLE I

EXECUTION TIME OF DIFFERENT BENCHMARK VERSIONS IN SECONDS.
accesses by orders of magnitude.

- A through performance study of this combined approach using the IBM Power 775 architecture [21]. This analysis indicates that this approach is an important step toward delivering the promised combination of productivity and performance through the use of the UPC language.

II. UNIFIED PARALLEL C

The Unified Parallel C (UPC) language follows the PGAS programming model. It is an extension of the C programming language designed for high-performance computing on large-scale parallel machines. UPC uses a Single-Program-Multiple-Data (SPMD) model of computation in which the amount of parallelism is fixed at program startup time. The UPC language can be mapped to either distributed-memory machines, shared-memory machines or hybrid machines that are clusters of shared-memory machines.

Listing 1, presents the computation kernel of the fish gravitational benchmark. The benchmark emulates fish movements based on gravity. The benchmark is an N-Body gravity simulation that solves ordinary differential equations in parallel [19]. Arrays fish and accel are declared as shared (lines 7-8). Shared arrays and shared objects are accessible from all UPC threads. The layout qualifier [NFISH/THREADS] specifies that the shared object is distributed in blocked form to different UPC threads. The construct upc_forall (line 13) distributes loop iterations among the UPC threads. The fourth expression in the upc_forall construct is the affinity expression. The affinity expression specifies that the owner thread of the specified element executes the ith loop iteration.

```c
typedef struct fish { double x; double vx; double y; double vy; } fish_t;

typedef struct f_acc { double ax; double ay; } fish_accel_t;

shared [NFISH/THREADS] fish_t fish[NFISH];
shared [NFISH/THREADS] fish_accel_t accel[NFISH];

for (each time step) {
  /* Phase 1: Force calculation */
  upc_forall (i=0; i<NFISH; ++i; &fish[i]) {
    tmpx = tmpy = 0;
    for (j=0; j<NFISH; j++) {
      dx = fish[j].x - fish[i].x;
      dy = fish[j].y - fish[i].y;
      a = calculate_force(dx, dy);
      tmpx += a * dx / r;
      tmpy += a * dy / r;
    }
    acc[i].ax = tmpx; acc[i].ay = tmpy;
  }
  upc_barrier();
  /* Phase 2-3: max_norm calc & Fish movement */
}
```

Listing 1. UPC version of gravitational Fish.

The UPC compiler translates the shared accesses to runtime calls. Runtime calls are responsible for fetching, or modifying, the requested data. Each runtime call may imply communication of one element of the array, leading to fine-grained communication that results in poor performance.

Furthermore, the compiler privatizes the accesses fish[i].x and fish[i].y in lines 15 and 16. However, the accesses fish[j].x and fish[j].y on lines 15 and 16 are not privatized because the program accesses the full shared array. In this case, the prefetching optimization of the compiler [22] transforms the loop into an inspector-executor form and aggregates, at runtime, the shared accesses. Listing 2 presents a simplified version of prefetching using the inspector-executor loop transformation. There are three entry points: the __sched_add_access, the __sched_dereference, and the __schedule calls. Before accessing shared pointers, the compiler also creates calls to shared pointer arithmetic (__ptr_arithmetic). The shared pointer is a fat pointer that contains information about the offset, the thread, and the allocated size [23]. When the number of the UPC threads is an integer power of two and it is known at compile time, then the pointer arithmetic call is replaced with shifts and masks. Two problems arise from codes with fine-grained accesses to shared data: (i) inefficient communication because of the exchange of many short messages, and (ii) high overhead because of the large number of runtime calls created.

```c
1 ... upc_forall (i=0; i<NFISH; ++i; &fish[i]) {
2  tmpx = tmpy = 0;
3  for (j=0; j<NFISH; j++) {
4    ptr = __ptr_arithmetic(&fish[j].x);
5    __sched_add_access(ptr, ...);
6    tmpx += a * dx / r;
7    __sched_dereference(ptr, ...);
8    ptr = __ptr_arithmetic(&fish[j].y);
9    __sched_add_access(ptr, ...);
10   }
11   __schedule(); /* Schedule shared accesses */
12 for (j=0; j<NFISH; j++) {
13  ptr1 = __ptr_arithmetic(&fish[j].x);
14  tmpl = __sched_dereference(ptr1, ...);
15  ptr2 = __ptr_arithmetic(&fish[j].y);
16  tmp2 = __sched_dereference(ptr2, ...);
17  ...
18 }
19 }
20 ...
```

Listing 2. Simplified example of inspector-executor.

III. INSPECTOR-EXECUTOR IMPROVEMENTS

The experimental prototype for the code transformations described in this paper is built on top of the XLUPC compiler framework [24].

The biggest drawback of the inspector-executor code transformation is the overhead of function calls that the compiler introduces in order to inspect which data transfers are amenable for coalescing. Therefore, an important goal is to decrease the overhead of inspector-executor transformations by reducing the number of function calls executed at run time. Figure 1 presents the algorithm used to optimize loops with fine grain accesses using the inspector-executor transformation. After categorizing the access pattern into regular or irregular, the compiler analyses the stride to select an appropriate code transformation. A loop with regular accesses [25] is transformed
using the CSLMAD framework (C.1) and all the runtime calls are removed from the inspector loops (C.1.1). Alternatively, for arrays that are not allocated in blocking fashion, two versions of the loop are created: one with run-time calls and the other without (C.1.2). In the case of irregular accesses, the compiler creates a temporary array to collect the elements (C.2). Finally, if the programmer uses aggregated data types (such as structs in C), the compiler tries to apply static codeless (C.3.1) or applies the original form of inspector-executor transformation (C.3.2).

A. Constant-Stride Linear Memory Descriptors

An array access analysis based on the Constant-Stride Linear Memory Access Descriptors (CSLMADs) identifies the type of access in a loop [26]. If the accesses on a shared array are regular, then the calls in the inspector loops are replaced with a single call.

CSLMADs are a restricted form of Linear Memory Descriptors [25] used to describe array accesses. CSLMADs lead to much simpler code transformations than the more general Linear Memory Access Descriptors description. The main restriction that differentiates a CSLMAD from an LMAD is that a CSLMAD cannot represent overlapping indexing expressions. CSLMADs still capture a surprisingly large set of index expressions that appear in numerical applications — it also captures all the expressions that appear in the benchmarks used in this paper.

Each array access in CSLMAD form can be expressed as:

\[ f(x) = b + a \times x \]

The constant \( a \) is the stride of the CSLMAD and the integer constant \( b \) is the base of the CSLMAD (offset). Using the loop-range information the compiler transforms the descriptors to the following format:

\[ (a, local\_offset, low\_bound + b, upper\_bound + b) \]

B. Runtime improvements

The runtime uses a compact form to keep track of shared accesses if the shared array is allocated in blocked form. The runtime stores the shared accesses in the form of: \((\text{stride, local\_offset, lower\_bound, upper\_bound})\). Hence, an additional benefit of using the compact representation form in the runtime is that the accesses do not require additional analysis. Also, the runtime tries to merge different CSLMADs when the descriptors have the same shared base array and stride. Thus, there is no duplication in the data transfers. Moreover,
the runtime reuses the internal data structures for subsequent iterations by setting the new range of iterations to inspect.

C. CSLMADs in dynamic environments

Another challenge the compiler must address is the usage of shared pointers when the number of threads are not available at compile time. In this case, the compiler produces two variants of the executor loop (C.1.2). The first assumes that the loop has blocked allocation and the second that the loop has blocking factor other than ideal. The compiler adds a branch to verify that all the arrays accessed are in blocked fashion. Thus, when any of the arrays has a blocking factor that is not ideal, the program executes the loop with the calls.

D. Usage of vectors

The compiler analyzes shared accesses occurring in irregular fashion to check if they access more complex structures, such as shared arrays of aggregated data types. For irregular accesses on shared arrays of native types, a temporary array (vector) stored in the stack is used to collect the shared indexes. The code generation inserts a call to inspect the elements at the end of each inspector loop. Internally the runtime processes the elements one by one. Thus, the performance gain is limited compared with the previous approach. The main benefit of this solution is a reduction in the number of calls in the inspector loop.

E. Combining dynamic with static Coalescing

Finally, the array-access analysis checks if the loop contains shared accesses to fields of aggregated data types that have constant stride. The algorithm uses a previously proposed combination of dynamic and static coalescing methods [22].

```
shared int A[128];
for (i=0;i<PF;i++)
    ptr = __ptr_arithmetic(&A[i]);
if ( ptr.thread != MYTHREAD )
    __sched_add_access( ptr, ...);
__schedule();
// Similar approach in the executor loop:
for (i=0;i<PF;i++)
    if ( ptr.thread != MYTHREAD )
        local_ptr = __sched_dereference(ptr,...);
    else {
        // Simple pointer additions
        local_ptr = CALC_LOCAL(ptr);
        ...
        = local_ptr;
```

Listing 3. Example of code modifications for the inline checks.

F. Inline checks

A number of UPC applications contain shared references that target the local address partition but cannot be proven to be local at compile time. To solve this problem, this paper proposes a new idea of inlining. The inline check optimization inserts a branch before two entry points of the runtime: the __sched_add_access and the __sched_dereference calls. These branches check if the data accessed are remote or local. When the shared accesses are local the runtime avoids collecting the accesses. Instead the executor loops use thin pointers to read the local data. The compiler applies this transformation, in addition to the transformations described earlier, when the benchmark exhibits irregular access pattern. Listing 3 presents an example of the code transformation.

IV. Experimental Methodology

This evaluation uses an IBM® Power® 775 supercomputer [21] with 64 nodes with 32 Power7 [29] cores on each node, running at 3.856 GHz, totaling 2048 cores. The machines are grouped in drawers consisting of eight nodes. Four drawers are connected to create a SuperNode (SN). The nodes are equipped with the POWER7 Hub chip interconnect [30] for communication.

All runs use one process per UPC thread and schedule one UPC thread per POWER7 core. There are 32 UPC threads on each node and each UPC thread is bound to its own core. The results presented in this evaluation are the average of the execution time of five runs. The maximum execution time variation is less than 3%. All benchmarks are compiled using the `-qarch=pwr7 -qtune=pwr7 -O3 -qprefetch` compiler flags to enable POWER7 specific code transformations. The evaluation tries to keep the computation constant per UPC threads (weak scaling).

The code transformations described target applications that contain fine-grained accesses of field members on shared structures. Table II presents the list of the benchmarks used in this evaluation and their communication pattern — they all use blocked data allocation. For this evaluation, five different binaries were generated for each program:

- **Baseline**: compiled with a dynamic number of threads and with the code transformations described in this paper disabled. The baseline is the best available compiler and contains a number of optimizations: static coalescing [12], privatization [14], and remote updates [31].
- **Prefetch**: compiled with the inspector-executor code-transformation that prefetches and coalesces shared references at runtime [10].
- **Prefetch Optimized**: combines the inspector-executor transformation with the improvements presented in Section III and with dynamic number of threads.
- **Hand-optimized**: uses coarse-grained communication, manual pointer privatization, and collective communication whenever possible. This version also uses dynamic number of threads.
- **MPI**: contains coarse-grained communication and uses collective communication whenever possible. This version uses blocking communication and it does not use the one-side communication model introduced in MPI 2.0.

V. Experimental results

This experimental evaluation assesses the effectiveness of the transformations by presenting the following: (1) the perfor-
Table II: Benchmarks and Communication Type.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
<th>Transformations (Section III)</th>
<th>Communication Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stream-like</td>
<td>Microbenchmark: read data from the next thread.</td>
<td>Prefetch: C.3.1</td>
<td>Stream-like from neighbour thread</td>
</tr>
<tr>
<td>Random-access</td>
<td>Microbenchmark: read data randomly.</td>
<td>Prefetch: C.3.1</td>
<td>Random access</td>
</tr>
<tr>
<td>Sobel [18]</td>
<td>Computes an approximation of the gradient of the image intensity function, using a nine-point stencil.</td>
<td>Prefetch: C.1.1/C.1.2</td>
<td>All-to-all/Reduction</td>
</tr>
<tr>
<td>WaTor [27]</td>
<td>Simulates the evolution over time of predators and preys in an ocean.</td>
<td>Prefetch: C.1.1/C.2/C.2 and Inline</td>
<td>All-to-all / Irregular</td>
</tr>
<tr>
<td>Mcop [28]</td>
<td>Matrix chain multiplication problem: finds the most efficient way to multiply these matrices together.</td>
<td>Prefetch: C.2 and Inline</td>
<td>Random updates</td>
</tr>
<tr>
<td>Guppie [20]</td>
<td>Random read/modify/write accesses to a large distributed array.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. 3. Performance in GB/s for the microbenchmark reading four fields from the same data structure for different versions.

Table III: Benchmarks compared with the serial C non-instrumented version and UPC version measured in seconds.

<table>
<thead>
<tr>
<th>Bench</th>
<th>Dataset</th>
<th>Seq. C</th>
<th>1 UPC Thread</th>
<th>32 UPC Threads</th>
<th>256 UPC Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sobel</td>
<td>64K x 64K Pic</td>
<td>92.1</td>
<td>110.1</td>
<td>3.6</td>
<td>0.7</td>
</tr>
<tr>
<td>Fish</td>
<td>16K Objects</td>
<td>155.7</td>
<td>727.7</td>
<td>24.8</td>
<td>7.2</td>
</tr>
<tr>
<td>WaTor</td>
<td>4K x 4K Grid</td>
<td>9.8</td>
<td>791.3</td>
<td>98.2</td>
<td>28.1</td>
</tr>
<tr>
<td>Guppie</td>
<td>2^28 Elements</td>
<td>70.5</td>
<td>349.6</td>
<td>104.3</td>
<td>22.1</td>
</tr>
<tr>
<td>Mcop</td>
<td>1024 Arrays</td>
<td>16.6</td>
<td>29.3</td>
<td>140.8</td>
<td>22.0</td>
</tr>
</tbody>
</table>

This higher bandwidth results from the random traffic pattern in combination with the high-radix interconnect when using direct routes. Previous research on the PERCS interconnect architecture confirms these experimental results [32], [22].

B. UPC Single-Threaded Slowdown

This section, prior to the scalability measurements, studies the performance of UPC language compared with the serial version. The single-thread overhead, shown in Table III compares the execution time of the UPC version of the program running on a single thread with the execution time of a sequential C version of the code. The most important cause for the increase in the single-thread overhead is the use of the pointers to reference data in distributed arrays. The runs with 32 and 256 UPC threads are performed with the inspector-executor transformation and the other code transformations presented in this paper. Some benchmarks, such as Fish and Guppie, run much slower than the C version even with a large number of threads because the compile time data-access analysis is unable to detect and simplify accesses that are local.

The large slowdown for the dynamic single-threaded UPC version of WaTor can be explained by its large number of shared accesses for which the compiler generates calls to the runtime system. On the other hand, the smaller single-thread slowdown for Guppie can be explained by its irregular accesses that make its serial C version slower because of poor cache utilization. Sobel has the best potential compared with the other benchmarks for two reasons. First, it has good shared data locality because it fetches data only from the neighboring threads. Second, the optimizations removes the calls from the inspector and executor loops. The low performance in the single-thread version occurs because the program executes the unoptimized version of the loop to avoid the overhead of the shared-access analysis. This slowdown underscores the key role of the code transformations that remove unneeded runtime...
calls automatically. A point to take is that PGAS programmers and compilers should not focus only on reducing the cost of communication, but also in reducing the runtime calls.

C. Application Performance

This section explores the performance of the code transformations when applied to benchmarks. As described in Table II, the access-analysis leads only to the removal of the runtime calls in Sobel and Fish benchmarks because those are the only benchmarks that contain regular accesses. The analysis allows the partial removal of the calls in MCop, WaTor and Guppie have complex access patterns and the compiler uses strict-field coalescing and the vector collection of elements in the inspector loop. The inline code transformation is also applied to the MCop, WaTor, and Guppie benchmarks.

Sobel achieves a performance gain between 1.5X and 2X using the inspector-executor (prefetch) code transformation as shown in Figure 4(a) The prefetch optimized technique achieves from 9.2X up to 12.3X speedup over the baseline because it allows for the complete removal of library calls. The hand optimized UPC version is faster than the MPI version because it uses one-side communication. However, the performance of the hand optimized and the MPI versions are converging with more than 256 UPC threads. One interesting observation is that the prefetched optimized version is faster than the UPC hand-optimized because of double buffering. The current version of the UPC language does not support asynchronous memget/memput calls. Thus, the exploitation of the overlapping communication and computation is the main advantage of the compiler transformation.

The Fish benchmark exhibits high performance gains because the baseline is inefficient, as shown in Figure 4(b). The compiler uses the CSLMADs representation to remove the runtime calls from the inspector and executor loops. The benchmark achieves from 40% up to 80% of the performance of the hand optimized version of the benchmark. The compiler successfully transforms one out of the two loops that contain fine-grained communication. The second loop implements a data reduction and becomes the bottleneck after the compiler applies the loop transformations.

The performance gain of the WaTor benchmark is lower than the Sobel and Fish: the prefetch optimized version is 1.12X to 1.72X faster than the baseline (Figure 5(a)). The compiler transforms a loop structure that has constant number of iterations (25): the stencil computation. The compiler improves the performance of the remaining fine-grained shared accesses using the remote update to eliminate runtime calls [31]. The MPI version is faster but requires additional code before and after the calculation of force to move objects.

The Guppie benchmark uses random remote updates across a large shared array and calculates the performance in MegaUpdates/s. Due to irregular accesses, the prefetch optimized version of the benchmark achieves between 1.6X and 2.53X speedup over the baseline (Figure 5(b)). The compiler removes the calls from the inspector loops, thus decreasing the instrumentation overhead of collecting shared accesses. It is known that manual code modifications to this benchmark allow the application of the remote-update code transformations [31]. The benchmark uses a temporary buffer to fetch the data, modify, and write them back. The typical size of this buffer is 512 elements. In the UPC hand optimized version the number of elements is set to one. Thus, the compiler collapses the loops to apply the remote-update optimization and to exploit the hardware acceleration.

The MPI version of the Guppie benchmark generates the data on all processors and distributes the global table uniformly to achieve load balancing. The benchmark sends the addresses to the appropriate processors and the local process performs the updates. The MPI version is faster than the UPC versions for small number of threads. On the other hand, the manual UPC optimized version is 46X times faster than the MPI version running with 2048 Threads. The automatic compiler-optimized version achieves from 22% up to 48% the speed of the MPI version.

The prefetch code transformation gives a speedup from 1.6X up to 2.6X compared with the baseline version in the MCop benchmark, as shown in Figure 5(c). Applying the code transformations and manually unrolling the loops by four gives a speedup from 4.9X up to 6.3X. Despite the removal of most calls in the loops, there are still irregular references. The manually optimized version still contains irregular remote shared references. Prefetching these references improves the performance of the application. The hand optimized combined with the prefetching is two orders of magnitude faster than the MPI version.

![Fig. 4. Performance numbers for Sobel (a), and Fish (b) for different versions.](image)
D. Overhead Analysis

Two representative benchmarks are selected for the overhead evaluation: Sobel, which contains regular access and Guppie, which contains random accesses. Figure 6 presents a breakdown of the normalized execution time before and after the code transformations, using Linux Perf Tool. Using the inspector-executor approach (Prefetch) in Sobel benchmark, the time devoted to the computation decreases significantly. The Sobel benchmark spends more than 55% of the time in the shared-pointer arithmetic in the Prefetch version because of the additional calls in the inspector loops. The shared-pointer arithmetic translates the shared offset to the relative offset inside the thread. Removing the calls from the inspector and executor loops decreases the overhead to less than 8% of the application time.

On the other hand, the impact of the code transformations in Guppie is less than in Sobel. The optimized inspector-executor transformation in Guppie removes the calls from the inspector loops, but retains the calls in the executor loops. The improved inspector-executor transformation (Prefetch Optimized) reduces the communication overhead down to 57%. However, the overhead is transferred to the shared-reference analysis because of the irregular communication pattern. Therefore, the improved transformation successfully eliminates the apparent overhead in the application’s code, but the runtime still processes the elements one by one. The inline code transformation has minor impact on the achieved performance and it is only visible for lower number of threads, when a certain portion of shared accesses are local.

E. Summary and Discussion

The code transformations presented can generate code for applications with regular accesses that achieve between 60% and 180% of the performance of the hand-optimized UPC version. The evaluation results support the argument that code transformations should focus on removing run-time calls completely in addition to the traditional compile and runtime optimizations, such as privatization, coalescing, and shared-data caching.

VI. RELATED WORK

Shared Object Coalescing: Coalescing accesses that target a single shared object on the same UPC thread, or same remote node, is a well-known code transformation that aims to reduce instrumentation code. When using static analysis for data coalescing in Unified Parallel C [11], [12] and High Performance Fortran [33], the compiler identifies, through data and control flow analysis, shared accesses to specific threads and creates a single run-time call to access multiple data items from the same thread. However, existing solutions do not completely remove the calls.

Shared-Pointer Privatization: The compiler uses information provided by the affinity expression of an upc_forall loop to privatize shared accesses to the local partition of the memory [14], [15]. For instance, an affinity expression that is pointer-to-shared usually indicates that the references are to local memory. Thus, the compiler can transform a fat shared pointer into a thin private pointer and completely remove the run-time call. Unfortunately, this approach only works for upc_forall construct and requires that the physical data placement be known at compile time.

Array Accesses Analysis: Linear Memory Access Descriptors (LMADs) [25] are a well-known representation that describes linear accesses to an array. LMADs are used for array-access analysis, coalescing of accesses, and for privatizing array accesses on various platforms. For instance, Xhu et al. use the LMAD representation to translate programs manually for distributed-shared-memory (DSM) systems [34]. Xinzhao Li and Garg use a subset of LMAD called Restricted Constant Strided Linear Memory Access Descriptor (RCSLMAD) to identify memory locations of accessed array elements in Graphic Processor Units (GPUs).
VII. CONCLUSION

Eliminating runtime calls inserted by the compiler to translate shared-memory accesses into inter-node data transfers is essential to deliver reasonable performance in PGAS programming models executing in distributed-shared-memory machines. This paper uses a combination of new code transformations and adaptations of known techniques to the PGAS paradigm to eliminate such calls. The experimental evaluation indicates that when such calls are completely removed, the performance of automatically generated code is similar to the performance of coarse-grained versions of the benchmarks. On the other hand, there is room to improve the performance of applications that contain irregular accesses. The transformation helps to deliver the promised programming productivity of PGAS languages by allowing the programmer to write simpler fine-grained code.

ACKNOWLEDGMENTS

The researchers at Universitat Politècnica de Catalunya and Barcelona Supercomputing Center are supported by the IBM Centers for Advanced Studies Fellowship (CAS2012-069), the Spanish Ministry of Science and Innovation (TIN2007-60625, TIN2012-34557, and CSD2007-00505), and the European Commission in the context of the HiPEAC3 Network of Excellence. IBM researchers are supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002. The researchers at University of Alberta are supported by the NSERC Collaborative Research and Development program of Canada.

REFERENCES