Enabling PGAS Productivity with Hardware Support for Shared Address Mapping: A UPC Case Study

OLIVIER SERRES, George Washington University, Washington, DC

ABDULLAH KAYI, Parallel Computing Lab, Intel Corporation, Hillsboro, OR

AHMAD ANBAR and TAREK EL-GHAZAWI, George Washington University, Washington, DC

Due to its rich memory model, the partitioned global address space (PGAS) parallel programming model strikes a balance between locality-awareness and the ease of use of the global address space model. Although locality-awareness can lead to high performance, supporting the PGAS memory model is associated with penalties that can hinder PGAS's potential for scalability and speed of execution. This is because mapping the PGAS memory model to the underlying system requires a mapping process that is done in software, thereby introducing substantial overhead for shared accesses even when they are local. Compiler optimizations have not been sufficient to offset this overhead. On the other hand, manual code optimizations can help, but this eliminates the productivity edge of PGAS. This article proposes a processor microarchitecture extension that can perform such address mapping in hardware with nearly no performance overhead. These extensions are then availed to compilers through extensions to the processor instructions. Thus, the need for manual optimizations is eliminated and the productivity of PGAS languages is unleashed. Using Unified Parallel C (UPC), a PGAS language, we present a case study of a prototype compiler and architecture support. Two different implementations of the system were realized. The first uses a full-system simulator, gem5, which evaluates the overall performance gain of the new hardware support. The second uses an FPGA Leon3 soft-core processor to verify implementation feasibility and to parameterize the cost of the new hardware. The new instructions show promising results on all tested codes, including the NAS Parallel Benchmark kernels in UPC. Performance improvements of up to $5.5\times$ for unmodified codes, sometimes surpassing hand-optimized performance, were demonstrated. We also show that our four-core FPGA prototype requires less than 2.4% of the overall chip's area.

Categories and Subject Descriptors: B.1.4 [Hardware]: Emerging Technologies, Analysis and Design of Emerging Devices and Systems, Emerging Languages and Compilers

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: PGAS, hardware support, productivity, UPC, address translation

ACM Reference Format:

Olivier Serres, Abdullah Kayi, Ahmad Anbar, and Tarek El-Ghazawi. 2015. Enabling PGAS productivity with hardware support for shared address mapping: A UPC case study. *ACM Trans. Archit. Code Optim.* 12, 4, Article 52 (December 2015), 26 pages.

DOI: <http://dx.doi.org/10.1145/2842686>

This work was partially supported by the National Science Foundation under grant CCF-1547980.

Authors' addresses: O. Serres, A. Anbar, and T. El-Ghazawi, ECE Department, School of Engineering and Applied Science, George Washington University, 800 22nd Street NW, Washington, DC 20052; emails: {serres, anbar, tarek}@gwi.edu; A. Kayi, Parallel Computing Lab, Intel Corporation, Hillsboro, OR; email: abdullah.kayi@intel.com.

Extension of a Conference Paper: "Enabling PGAS Productivity with Hardware Support for Shared Address Mapping: A UPC Case Study," O. Serres, A. Kayi, A. Anbar, and T. El-Ghazawi, in Proceedings of the 2014 IEEE International Conference on High Performance Computing and Communications.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2015 ACM 1544-3566/2015/12-ART52 \$15.00

DOI: <http://dx.doi.org/10.1145/2842686>

1. INTRODUCTION

The two widely accepted parallel programming models—shared memory and message passing—each have advantages and disadvantages in terms of programmability and performance. The shared memory model gives the programmer an easy-to-program shared view of the memory space and supports one-sided communication. However, the shared memory model has no data locality awareness; that can lead to a severe degradation in performance. This is because the memory access latency across the memory space of a distributed system is not even, and therefore a programmer may create accesses to remote data without an explicit awareness. In contrast, the message passing model allows the programmers to fully exploit data locality awareness to achieve high performance at the cost of ease of programming. As this model allows only explicit two-sided communication, it is the programmer's burden to explicitly express the data movements, both at the sender and receiver tasks. The partitioned global address space (PGAS) programming model [El-Ghazawi et al. 2005] strikes an optimal performance and productivity balance between these two models. PGAS presents a partitioned memory view, allowing the programmer to exploit data locality while maintaining a shared view of the memory with one-sided communication capability. However, this partitioned global view of the memory entails a more complex addressing mode to map the programmer's view of the memory to the actual physical layout of the memory. This combination of data locality awareness and programmability necessitates (1) a way to represent shared addresses in the global view and (2) a translation between the global representation and the virtual address representation. For example, in the Unified Parallel C (UPC) language, three different fields are necessary to represent a shared address, creating significant overhead within the runtime environment to translate this representation to a regular memory address. This overhead can create performance degradation that force programmers to manually optimize their codes with the use of complex pointers and MPI-like messaging. These manual optimizations significantly reduce the overall productivity advantage of PGAS.

To solve this issue, we propose a hardware support mechanism to handle complex PGAS address mapping tasks with hardware assistance via a new set of instructions, eliminating the need to manually tune codes to increase PGAS performance. A PGAS compiler can easily make use of these new instructions to efficiently manipulate shared addresses and efficiently convert those to the internal system representation for fast access. The proposed hardware support aims to close the performance gap between shared pointer addressing and private pointer addressing with modest hardware changes. To verify the performance improvements and feasibility of our proposed hardware mechanism, we conducted an extensive full-system analysis using the gem5 full-system simulator to obtain performance results with up to 64 cores, as well as a complete prototyping in an FPGA environment with a Leon3 soft-core processor, allowing us to evaluate the feasibility and the required chip area needed.

The rest of this article is organized as follows. Section 2 provides an introduction to PGAS languages. Section 3 reviews the related work. Section 4 presents the UPC language and the implementation details for its partitioned shared memory model. Section 5 discusses the proposed PGAS hardware support. Section 6 describes our prototype implementations. Section 7 presents and discusses the experimental results. Section 8 covers our conclusions and future work.

2. INTRODUCTION TO PGAS LANGUAGES

Domain scientists and programmers have the most knowledge about their respective computational problems. The motivation behind the development of the PGAS languages, such as UPC [El-Ghazawi et al. 2005], Chapel [Chamberlain et al. 2007], and

Fig. 1. Vector summation in UPC—memory layout.

```

1  shared [4] double data[16];
2  shared [1] double partialSum[4]; // Local shared partial sums
3  partialSum[MYTHREAD] = 0; // Initialize partial sums
4
5  double* privateSum = (double *) &partialSum[MYTHREAD]; // sharer pointer privatization
6
7  upc_forall (int i=0; i<16; i++; &data[i])
8      *privateSum += data[i]; // Compute local sums
9
10 upc_barrier;
11 if (MYTHREAD == 0){ // Reduce at thread 0
12     double totalSum = 0.;
13     for (int i=0; i<4; i++)
14         totalSum += partialSum[i];
15 }

```

Listing 1. Vector Summation in UPC

X10 [Charles et al. 2005], is to provide a programmer with (1) precise control over the affinity of shared data elements to specific threads of the application and (2) semantic constructs for workload distribution that are resolved based on the data affinity. Programmers thus have a straightforward way to design a data layout in the shared memory space that enforces how an application’s data maps to specific threads. While imperative for attaining good performance, these correlated data locality and workload distribution capabilities significantly improve overall programmer productivity.

A simple UPC vector summation example is shown in Listing 1 to demonstrate these features. UPC is a parallel extension of ISO C 99 programming language implementing the PGAS model. It follows a single program multiple data (SPMD) execution model in which a specified number of identically coded thread instances are executed in parallel. Although the code base for each thread is identical, each thread may execute against a different portion of the data space.

The code sample in 1 can be simplified by using UPC collective operations, but in this form it highlights the basics of the UPC language syntax. Lines 1 and 2 show an example of the declaration of a shared memory variable in UPC. The `shared` keyword is used to mark shared arrays, and the memory layout of the array is determined by the *blocking factor*, a value appearing in brackets in front of the array name as shown in Figure 1. In this case, the runtime system will distribute 4 sequential elements of the array to each of the four UPC threads for a total of 16 elements. The `upc_forall` construct is used for parallel workload distribution. It resembles a regular C `for` statement with an additional *affinity* parameter that controls the way the runtime system will distribute a particular iteration of the `upc_forall` loop to a specific thread or set of threads. The key concept is that specific iterations of the `upc_forall` loop will execute against the portion(s) of the dataset(s) with which they have affinity. Note that there is a dependency between how the shared memory variables are declared and how the affinity of the `upc_forall` is specified to ensure optimal performance.

The `upc_barrier` keyword is used to enforce interthread synchronization by forcing all threads to block until all threads reach the barrier statement. Lines 11 through 15 are only performed by thread 0, to combine partial results obtained from all threads.

Privatization is a common code optimization, which seeks to obtain a private pointer of a shared variable, as shown in line 5. Using the `privateSum` pointer, instead of `partialSum` shared pointer, will help avoid a more performance expensive shared data access. In line 8, each thread will do a private access to `privateSum` pointer and a shared access to data shared array. The privatization optimization can also be applied to data array; however, for the sake of simplicity, it was not included in this example.

PGAS languages tend to incur a significant performance overhead when accessing shared data. From a PGAS compiler/runtime perspective, there are two different address spaces: the private address space and the shared address space. The private address space corresponds to the native view of memory provided by the system. The distributed shared address space view is not usually provided by the operating system or hardware, and a PGAS compiler/runtime system must emulate the shared view of the memory distributed across the several SMP nodes of a compute cluster. When accessing PGAS private data, a compiler can generate a normal C language pointer, representing a virtual address against which to issue platform standard load and store instructions. Programs can manipulate the pointer with simple additions and subtractions. In contrast, a PGAS compiler must manipulate a more complex (and thus more performance costly) shared pointer to access even local data. As shared data might be stored locally or on a remote node relative to the locality of the calling thread, the PGAS-generated code also needs to check if the shared data access requires remote communications. Once this is determined, the compiler computes the virtual address required to perform the access. In a previous study, it was shown that a simple local access (including the shared pointer structure manipulation) can generate up to 20 load and 6 store instructions [Cantonnet et al. 2003]. In the absence of any hardware support, the compiler either needs to optimize all of the shared memory accesses, which is nontrivial in view of the potential data layout permutations, or the code developer needs to manually optimize the code (e.g., by casting local pointers to private pointers when it safe to do so), which dramatically reduces the productivity and maintainability of PGAS language based codes. This is discussed in more depth in the related work and in Section 5.1.

Although we used UPC as a case study, other PGAS languages behave in similar fashion as it relates to shared memory addressing. For example, Chapel allows the distribution of data structures across multiple *locales* [Dun and Taura 2012]. The Chapel compiler must then maintain the mappings that allow a programmer to address distributed arrays or other data structures. In X10, a user can distribute data across multiple *places* and use Global references to access the data [Grove et al. 2014]. In addition, *unsafe native* methods can be used by the programmer to manually optimize the local data accesses, similar to using a private pointer in UPC. The Chapel locales and the X10 places are locality domains that would also benefit from hardware support for shared memory accesses.

The hardware support that we propose would allow the programmer to simply use global references in all cases, and the compiler would then invoke the new instructions rather than attempt any further memory access optimizations. Furthermore, this eliminates the need for any additional manual optimizations from the programmer.

3. RELATED WORK

Previous studies have analyzed both the productivity advantage and the performance of PGAS languages [De Wael et al. 2015] under different levels of manual code optimizations and on a variety of systems. Many of those studies have shown that the shared pointer arithmetic and the address translation overhead represent the principal sources of performance degradation in UPC codes.

In Cantonnet et al. [2004], a clear advantage in terms of productivity is demonstrated when using UPC compared to MPI. Their experiments showed that UPC has a consistent improvement over MPI in terms of number of lines of code, number of characters, and conceptual effort to write a program of equivalent complexity. In Ebcioglu et al. [2006], a 4.5-day study is performed on 27 subjects to compare the productivity of parallel programming languages. The time to reach the correct output when using several parallel languages, including C+MPI and UPC, is compared. The study showed that the use of PGAS languages can significantly improve programmer productivity.

Many efforts were also performed in evaluating the potential of achieving performance using UPC. In Yelick et al. [2007], it is demonstrated that hand-tuned UPC code can achieve comparable performance to, and sometimes even better than, code in MPI. In Zhang and Seidel [2005], the performance of different UPC compilers on three different machines (x86 cluster, AlphaServer SC and Cray® T3E) is evaluated, and a significant overhead is noted for local shared accesses.

In El-Ghazawi et al. [2006], the overhead of the PGAS shared memory model is clearly demonstrated. They proposed a framework to assess the compilers and the runtime systems capabilities to optimize the overhead. Different compiler optimizations were researched, including optimization techniques such as lookup tables [Cantonnet et al. 2005; Serres et al. 2011a]; the reduced overhead is still significant, and the methods can consume a significant amount of memory. Alternative representations for shared pointers have also been implemented. For example, *phaseless* pointers are used for shared addresses with a block size of 1 or infinity [Chen et al. 2003]. Although this approach is only applicable to a minority of cases, it still incurs a significant performance overhead. More advanced compiler optimizations use linear memory descriptors [Alvanos et al. 2014] to reduce the amount of shared address translations. Although this provides good results for applications exhibiting regular memory access patterns, it still has a performance overhead due to the inspector/executor loop and scheduling, which in some cases can reduce the performance by an order of magnitude compared to hand-optimized code. In Dalton et al. [2014], *symmetric heap mirroring* is presented to reduce the performance cost of threads accessing elements in a shared memory space. Although reduced, the authors note that a substantive overhead still remains.

Hardware support for shared memory accesses has been implemented on multiple systems. An example of this is the T3D supercomputer that uses a “Support Circuitry” chip located between the processor and the local memory [Arpaci et al. 1995]. This chip, on top of providing functionality like message passing and synchronization, allows the processor to access any memory location across the machine.

In Fröning and Litz [2010], a network engine especially designed for PGAS languages is proposed. The engine allows network communication between nodes by mapping other nodes’ memory space across the network and providing a relaxed memory consistency model best suited for PGAS. The results were only presented in terms of read/write throughput and transaction rates, as no PGAS applications or benchmarks were actually tested. This approach is complementary to our work, as it focuses on improving the network interface performance for PGAS languages, and our work focuses on improving the shared memory addressing performance. An integrated implementation that leveraged improvements from both approaches, effectively creating a noncache coherent distributed memory system optimized for PGAS, would provide an additional improvement opportunity, and this is noted as future work in Section 8.

Another relevant work in the context of improving the network interface performance for PGAS languages is the Cray T3E [Scott 1996; Mueller 2000; Carlson et al. 1999]. The improvement focused on providing E-registers and a “centrifuge” hardware that performs an array mapping using four registers (index, mask, base address, stride and addend). This approach provides a good support for the data layout of PGAS languages

Fig. 2. UPC memory model.

Fig. 3. UPC memory layout for arrayA. Some shared pointers are also represented.

at the level of the network interface; however, it has multiple drawbacks. First, the system uses a large number of registers that are memory mapped and hence relatively slow to access. Second, the register hardware is located outside of the processor chip. As such, these registers are close to the network interface and do not improve the performance of very frequent local accesses.

4. PGAS MEMORY MODEL IMPLEMENTATION IN UPC

As noted in Section 2, UPC realizes the PGAS memory model by providing a shared memory view across the system that can be accessed by any thread; each thread has an affinity to the part of the shared memory residing locally. In addition, each thread has access to a local private space that is accessible only by the thread itself. This thread-private space has low overheads and allows for the best performance in terms of memory access time. Figure 2 provides an overview of the UPC memory model. The language also provides all of the facilities needed for parallel programming, such as process synchronization (locks, barriers), collective operations (scatter, gather, broadcast), and memory consistency constructs (fences, strict/relaxed reference type qualifiers). Accesses to either the shared or the private memory space are syntactically identical and done through simple variable accesses or assignments.

As shown in Figure 1, the distribution of shared data across different threads is controlled by a *block size* specified by the user for a given array: elements are distributed in group of block size elements in a round-robin fashion. Thus, the blocking factor gives the programmer a mechanism to control the data distribution in the shared space. Figure 3 presents how the elements of the following array are distributed across four UPC threads. Each thread has its own contiguous address space starting at a specific *base address*.

```
shared [4] int arrayA[32];
```

```

Require: blocksize, elemsize, increment, numthreads, shptr
Ensure: nshptr
    phinc = shptr.phase + increment
    thinc = phinc / blocksize
    nshptr.phase = phinc % blocksize
    blockinc = (shptr.thread + thinc) / numthreads;
    nshptr.thread = (shptr.thread + thinc) % numthreads;
    eaddrinc = (nshptr.phase - shptr.phase) + blockinc * blocksize;
    nshptr.va = shptr.va + eaddrinc * elemsize;

```

Fig. 4. Shared pointer incrementation algorithm.

To address such arrays, a UPC *shared pointer* is used. UPC shared pointers are similar to C pointers but are able to traverse shared arrays with the same C syntax, even if the elements of the shared array are spread across a distributed PGAS address space. Shared pointers effectively provide a mapping from the logical array construct to the actual physical location of the data across the whole shared address space of the system. Most UPC shared pointer implementations use a 64-bit address and consist of three fields: *thread*—thread affinity of the pointed data, *virtual address*—address of the current element in the local space, and *phase*—position inside the current block. This implementation allows the programmer to traverse a shared array in a logical, intuitive way.

Even if a UPC compiler uses a proprietary internal representation, the UPC specification [UPC Consortium 2005] provides the following function to indirectly access them: `upc_threadof`, `upc_phaseof`, `upc_addrfieldof`, and `upc_resetphase`.

Figure 3 presents a few examples of shared pointers (`ptrA`, `ptrB`, and `ptrC`) alongside their specific internal representations.

5. PROPOSED PGAS HARDWARE SUPPORT

Up to this point, we have described how PGAS (in general) and UPC (in particular) implement shared memory constructs and some examples of situations in which these implementations are suboptimal in terms of performance. In this section, we now present an overview of our proposed processor hardware extension to support compiler-driven, programmer-independent optimal shared memory accesses.

5.1. PGAS Memory Model Overheads

PGAS shared pointer manipulations are currently performed in software. To increment a shared pointer, the three fields of the pointer must be updated to point to a different element in the shared array, and extra information like the array block size, the element size (e.g., 4 bytes for `int`), and the number of running threads are needed. As an example, the algorithm presented in Figure 4 increments `shptr` to the new pointer `nshptr`. Pointer-wise, this is a particularly complex operation involving addition, subtraction, multiplication, and division. It requires temporary registers that can increase *register spill*, which has performance implications by itself, but this also creates extra pressure on the cache because of the possible eviction of data.

Shared pointer manipulations may be optimized by compilers using various methods. These methods include using a simpler representation for specific class of shared pointers (e.g., not storing the phase field for pointer with a block size of 1), removing the shared pointer construct itself when accessing the local space, or using transformations based on linear memory access descriptors (LMADs). However, these optimizations are not always feasible or fully effective due to code complexity, dependencies on intermediate results, or separately compiled code segments.

Fig. 5. Memory access cost for a 64-bit write on a dual-socket quad-core AMD cluster.

When an element is accessed in a shared array, the relative locality (remote vs. local) of the element is determined, and the shared pointer is then converted to a virtual address and then to a physical address so that the processor can perform the access. The virtual address is obtained by adding the virtual field of the shared pointer to the base address of the current thread. Even though this is not a compute-intensive operation, the performance overhead accumulates because it is done for each shared access and greatly increases the overall access time.

5.2. Overheads Quantification on Real Hardware

To quantify the overhead of the UPC memory model, a set of microbenchmarks were created to evaluate how the different potential latency dimensions (Network, Address Translation, Address Incrementation, Memory Access) accumulate to affect the performance of PGAS shared address manipulations. To get a precise timing, the benchmarks traverse a vector of 4 million elements and average access times are recorded. The tests were performed on a cluster of dual-processor (quad-core) AMD[®] Opteron[®] 2354 @2.2GHz interconnected with a 40Gbit/s QDR Infiniband[®] network. We used Berkeley UPC Compiler version 2.18.0 with the Infiniband (ibv) conduit.

In Figure 5, we can see that the overheads (Address Translation and Address Incrementation) are substantial, especially for the most frequent local accesses.

5.3. PGAS Memory Model Hardware Support

We have summarized the specific performance degradation aspects of shared memory access in the PGAS context. To systematically address this deficiency, our proposed hardware extension adds new instruction support to traverse PGAS shared pointers. From our verification tests, the new hardware extension almost eliminates the performance gap between the shared memory space addressing and the private space addressing in many common cases. Two principal types of operations need to be optimized to get an efficient addressing of the shared space: (1) array traversal, where pointers are incremented, and (2) translating the shared addresses to the final physical address for efficient reading and writing. Other operations that can benefit from hardware support include testing the locality of a shared pointer (i.e., checking if a shared pointer points to local data or not), thereby reducing the latency before invoking a communication subroutine if the data is remote.

The algorithm used to increment a shared pointer (see Figure 4) can be efficiently pipelined and implemented in hardware. This is especially true when `numthreads`, `blocksize`, and `elemsize` are powers of 2. This assumption is used in our implementations, allowing the replacement of (more performance costly) divisions and multiplications with simple bitwise operations (e.g., shifting and masking). This functionality is supplemental to the compiler's existing capability to fall back on the software implementation for the cases not supported by the hardware extension.

When using a shared pointer to access data, the physical location should first be computed. This is done by adding the base address of the thread accessing the pointer to the virtual address contained in the pointer. The virtual address will then be transformed to the final physical address using the conventional translation lookaside buffer (TLB) and the memory management unit (MMU). Access via the virtual address maintains the system integrity by providing memory isolation between processes.

At least two different implementations are possible for the address translation: (1) the base address can be computed directly from the thread number (similarly to Fröning and Litz [2010] and Dalton et al. [2014]) because the thread address spaces start at regular intervals, or (2) a lookup table can be used to retrieve the base address. The first method is more restrictive in terms of the possible addresses used but is more scalable, as it does not require storing a complete base addresses table. We used the second approach in our FPGA prototype implementation because it is a simpler implementation. For example, for `ptrC` in Figure 3, the system virtual address of the element would be computed by retrieving the base address of Thread 1 and adding the virtual address from the shared pointer as follows: $0\text{xff}0\text{b}00000000 + 0\text{x}3\text{f}00 = 0\text{xff}0\text{b}00003\text{f}00$.

5.4. Instruction Set Extension

To efficiently support the PGAS shared memory address space (not counting the variants for the access size or mode), compilers can leverage the hardware extension by calling the following new instructions to manipulate shared pointers and to load/store using a shared pointer as if it were an actual address: (1) *load/store from/to a shared address* and (2) *increment shared address*. The 64-bit shared addresses can be stored in processor registers, and the other information (i.e., block size/element size) can be directly encoded in the instructions. The increment value can be an immediate value or register direct, the latter providing easy array traversal. In our implementations, we also used a special register to store the number of UPC threads for the currently running program. The new instructions are preferable to software-based mechanisms because they allow for pipelining with other instructions. They also provide very low and predictable latency, resolving the local shared access issue described in the related work [Scott 1996; Mueller 2000; Carlson et al. 1999].

6. EXPERIMENTAL SETUP

We validated our hardware extension with two different experimental systems: (1) a full-system simulation allowing the evaluation of the performance characteristics of the hardware support and (2) a prototype FPGA hardware implementation allowing us to study the details of a physical hardware system and evaluate how much additional chip area is needed for the hardware extension.

6.1. Full-System Simulation with Gem5

The `gem5` [Binkert et al. 2011] simulator was used to model the extended instruction set. `Gem5` offers several advantages: (1) support for a wide variety of hardware architectures; (2) a large number of cache configuration/CPU model combinations, which

Table I. Instructions Added to the Alpha ISA

Shared Address Loads	
pgas_ld{bu, wu, l, q}	Load Unsigned {Byte, Word, Long, Quad} (8, 16, 32, 64 bits)
pgas_ld{s, t}	Load {S_float, T_float} (32 bits float, 64 bits double)
Shared Address Stores	
pgas_st{b, w, l, q}	Store Unsigned {Byte, Word, Long, Quad} (8, 16, 32, 64 bits)
pgas_st{s, t}	Store {S_float, T_float} (32 bits float, 64 bits double)
Shared Address Incrementations	
pgas_inc_imm	Address increment, immediate
pgas_inc_reg	Address increment, register
Initialization	
set_threads	Initialize the “threads” register
set_base_address	Set the base address lookup table

Fig. 6. New Alpha instructions format.

gave us a way to evaluate the trade-offs between simulation speed and accuracy; and (3) a high level of simulation accuracy [Butko et al. 2012].

We configured the simulator with the (custom) BigTsunami variant of the Alpha architecture because it supports full-system simulation up to 64 cores with GNU/Linux. Gem5 simulates 64-bit Alpha 21264 processors with the BWX, CIX, FIX, and MVI extensions. This architecture provides 32 integer registers (R0-R31) and 32 floating-point registers (F0-F31). Each 2GHz CPU core is configured with the *Classic* gem5 memory model, 32kb L1 code/data cache, and a 4MB shared L2 cache. Linux kernel version 2.6.27.62 (patched for BigTsunami) was used for the simulation, and the programs were compiled with the Berkeley UPC version 2.14.2 compiler and GCC version 4.3.2.

The Alpha instruction set is extended with the instructions shown in Table I. The instruction format is presented in Figure 6. The 64-bit integer registers (R0-R31) are used to store the shared addresses.

For loads/stores, RA and RB represents the source/destination registers. Opcode is an unused opcode. Func defines which type of load/store is going to be executed. Short disp is an offset added to the resulting virtual address after the shared pointer has been translated to the system’s virtual address. Short disp is particularly useful to access different members of a shared data structure.

For shared address *incrementation* instructions, RA and RC represents the source and destination registers. RB is used in the register form of the instruction to specify the increment register. Any value can be used with an increment register. Esize, Bsize, and Incram are 5-bit encoded immediate values for the element size, block size, and increment; they can represent any 32-bit value for which only one bit is set (1, 2, 4...).

The new address increment instructions use the same timing and functional unit as the Alpha integer instructions. The shared load/store instructions use the same timing as the usual load/store instructions. We verify this claim with our FPGA prototyping implementation as discussed in Section 6.2.

Table II. Leon3 Configuration

	Configuration
Cores	4x SPARC cores (SMP) @ 75MHz
Features	2-cycle multiplier, branch prediction, cache coherent, FPU not implemented
L1 I	I: 2 sets, 8 kB/set, 32 bytes/line, LRU; D: 4 sets, 4 kB/set, 16 bytes/line, LRU
Memory	Xilinx MIG-3.7 DDR3-800 connected with AMBA AHB with fast snooping
OS	GNU/Linux, Linux version 2.6.36

The usability of the new hardware instructions depends on the following criteria, which are true for most use cases. First, the data element size and block sizes must be a power of 2. Second, if an address offset is used, it must be a constant value. For each shared pointer manipulation or access, we check these criteria, and where possible, we issue the hardware instruction with GCC `asm()` statements. If the shared pointer does meet these criteria, we direct the compiler to choose the normal software incrementation routines like the one shown in Figure 4. A few simple compiler optimizations are also implemented for cases that on first inspection do not meet the criteria by using instruction combinations. For example, to increment by 3, the compiler generates an incrementation by 1, immediately followed by an increment by 2. The compiler will also generate extra initialization code, which is executed before the user main to initialize the base address of each thread and the special threads register.

We recognize that to deliver the productivity improvement of UPC, the new instructions need to be usable by a compiler without any programmer intervention. Our prototype compiler is based on the Berkeley UPC source-to-source compiler. Beyond the inclusion of the new instructions, we disabled the *phaseless* pointer optimization because it does not provide any benefit when the new hardware extension is present and also has the drawback of an incompatible shared pointer format. The source-to-source compiler takes as input a UPC source code and generates equivalent C code that is then compiled with the unmodified system GCC C compiler. We also modified the associated assembler to recognize the new instructions and their respective opcodes.

6.2. FPGA Prototyping with Leon3

We evaluated the execution timing and the chip area as usage feasibility of our hardware extension by using a Virtex-6 (XC6VLX240T) FPGA mounted on an ML605 evaluation board. The logic synthesis, place, and route for the design was performed using Xilinx ISE Release 13.4. Table II provides the complete FPGA prototype configuration.

A softcore Leon3 processor was extended with our proposed PGAS hardware support for shared addressing by using the reserved SPARC V8 coprocessor instructions. Extending the Leon3 softcore processor via the coprocessor interface is described in Serres et al. [2011b]. The Leon3 softcore processor implements the 32-bit SPARC V8 architecture with a seven-stage pipeline. It supports a cache-coherent symmetric multiprocessor (SMP) system running a full GNU/Linux operating system. The Leon softcore processor family has been used in prior studies [Guironnet De Massas and Amblard 2006; Danek et al. 2010] to explore a variety of hardware extension scenarios.

Figure 7 presents the Leon3 pipeline extended with hardware support for shared pointers. The new instructions (Table III) are fully integrated with the main processor pipeline. Since the default register file is 32 bits wide, we implemented a new 64-bit register file (identical to the default floating-point register file) for storing shared 64-bit address pointers. The new register file can read two 64-bit values and can write one 64-bit value per clock cycle. This new register file is not needed for any 64-bit architecture, such as our gem5 Alpha simulator described earlier.

Fig. 7. The seven-stage Leon3 pipeline extended with PGAS support.

Table III. PGAS Hardware Support SPARC V8 ISA Extension

Coprorocessor Load/Store	
LDC	Load to Coproc. reg. (32 bits)
STC	Store from Coproc. reg. (32 bits)
Shared Address Load/Store	
LDCM	Load Long (32 bits)
STCM	Store Long (32 bits)
Branch	
CB123	Branch on locality
Shared Address Incrementation	
SH_ADD_INC	Immediate
SH_ADD_INC_REG	Register

The address incrementation instruction is fully pipelined over two stages, allowing performance of one address incrementation per clock cycle. It also generates a coprocessor condition code based on the locality of the incremented address. Four condition codes are possible: 0—local (the referenced data is owned by the current thread), 1—located on the same memory controller, 2—accessible by the load/store from shared instructions, and 3—located on another node. The coprocessor branch (CB) instruction allows conditional branching based on any combination of the condition codes. Loads/stores from shared addresses (LDCM, STCM) are performed as fast as the normal load/store instructions.

7. RESULTS

In this section, the results from the gem5-based full-system simulation and the prototype FPGA hardware implementation are presented. We used three different

Fig. 8. Gem5 atomic model: NAS Parallel Benchmark—CG class W.

benchmark variants. The *No Manual* variant represents a baseline performance case because the kernels contain no manual optimizations and are compiled with an unmodified Berkeley UPC compiler with all compiler optimizations enabled. The *Privatization* variant is similar to the *No Manual* case but uses manually optimized kernels in which the shared pointers have been replaced by normal C pointers. Finally, the *No Manual, HW* variant compiles the non-hand-optimized kernels with our prototype compiler to demonstrate the performance advantage of the PGAS hardware support.

7.1. Full-System Simulation Results

Gem5 provides a number of CPU architecture models, each offering different trade-off combinations of simulation speed and accuracy. Specifically, we used three different CPU models: *atomic*—a single instruction per clock model; *timing*—which includes the simulation of the cache hierarchy and is a reasonable approximation of the performance of an in-order processor like the Leon3, Xeon Phi, or Tile64; and *detailed*—which accurately simulates a complex out-of-order processor and its associated cache hierarchy.

To evaluate the performance improvement of the PGAS hardware support, five kernels from the NAS Parallel Benchmarks [Bailey et al. 1995; El-Ghazawi and Cantonnet 2002; UPC NPB 2014] and two synthetic benchmarks were used:

EP: Embarrassingly parallel—generates pairs of Gaussian random variates

CG: Conjugate gradient—approximately computes the small eigenvalues of a symmetric positive matrix; exhibits long-range communication

FT: Fast Fourier transform (FFT)—solves a partial differential equation with a discrete 3D FFT

IS: Integer sort—bucket sort of small integers

MG: Multigrid—3D Poisson equation solving using a V-cycle multigrid method

MM: Matrix multiplication 512×512 elements

SOBEL: Sobel edge detection benchmark on a 4096×4096 image

We used two versions of the benchmarks: a nonoptimized version and a manually privatized version, which is the most common hand-tuning operation done by PGAS programmers. The privatized version was manually optimized to replace UPC shared pointers with private pointers [El-Ghazawi and Chauvin 2001]. Given the length of time needed for multicore simulations, only the relatively small W (workstation) class of the NAS Parallel Benchmarks was used.

7.1.1. Atomic Model. Figures 8 through 13 present the results when using the atomic model. The atomic model of gem5 is relatively fast, and we were able to run the

Fig. 9. Gem5 atomic model: NAS Parallel Benchmark—FT class W.

benchmarks with up to 64 cores (the limit of the BigTsunami architecture). The execution time of the atomic model is directly proportional to the number of dynamic instructions.

The figures for the EP kernel are not included because the results show that the hardware support does not provide any performance improvement. This makes intuitive sense, as EP does not use shared pointers in its main loops due to its embarrassingly parallel nature.

For the CG kernel, not all shared address incrementation instances were compiled using the new hardware instructions. There were a total of 309 shared pointer incrementation instances in the code. Twenty of them used a non-power of 2 element size (the arrays `w` and `w_tmp` have an element size of 56016). These instances had to be implemented with the normal (default) software incrementation algorithms. The code could have been changed to use an element size of 65536 (64K) for extra performance but was left unmodified to preserve the integrity of the comparison. All other shared pointer manipulations, including 236 loads/stores, were implemented using the hardware instructions. For the CG kernel (Figure 8) the *No Manual, HW* variant ran 2.6x faster than the *No Manual* variant. The *No Manual, HW* variant is 17% faster than the *Privatization* (manually optimized) variant.

All shared pointer manipulations in the FT kernel were compiled with hardware instructions (79 address incrementations and 47 load/store instructions). The FT kernel could only use up to 16 threads because of the geometric limitations of the data distribution of the W class. The FT kernel performance (see Figure 9) was improved 2.3x without manual optimizations. Similar to CG, the *No Manual, HW* variant is 17% faster than the *Privatization* variant.

For IS (see Figure 10), the *No Manual, HW* variant performed 3x better than the *No Manual* variant, but the *No Manual, HW* variant is still 13% slower than the *Privatization* variant. This might be due to missed optimization opportunities during the C compilation. The `asm` statements representing the PGAS store instructions are (1) marked as *volatile* and (2) flagged to indicate that they might have written a value anywhere in the memory space. These qualifiers prevent the GCC compiler from moving the store instructions and forces it to reload the data stored in registers.

The MG kernel *No Manual* variant performance is improved by $5.5\times$ (see Figure 11), but it runs 10% slower than the *Privatization* variant.

For the matrix multiplication kernel (see Figure 12), two different levels of manual optimizations were performed: *privatization 1* uses private pointers to access one of

Fig. 10. Gem5 atomic model: NAS Parallel Benchmark—IS class W.

Fig. 11. Gem5 atomic model: NAS Parallel Benchmark—MG class W.

Fig. 12. Gem5 atomic model: matrix multiplication 512×512 .

Fig. 13. Gem5 atomic model: Sobel benchmark 4096 × 4096.

Fig. 14. Gem5 timing model: NAS Parallel Benchmark—CG class W.

the matrices, and *privatization 2* uses a nonstandard UPC extension to access all matrices with private pointers, even when the matrix elements are not local to the calling thread. All shared operations benefited from the hardware support. The *No Manual, HW* variant showed improvements up to 7× over the *No Manual* variant. The performance of the *No Manual, HW* variant approaches the fully optimized (*privatization 2*) level, which provides an additional performance of 8% to 14%.

For the Sobel benchmark (see Figure 13), the *Privatization* variant is not available, so only the 3× to 4.5× speedup obtained by the *No Manual, HW* variant is presented.

7.1.2. Timing Model. Figures 14 through 19 present the results obtained using the *gem5* timing model. The timing model adds caches and memory timing to the simulation. The performance improvement of the hardware support is slightly reduced in proportion to the time spent accessing the memory. The single shared L2 cache also becomes a bottleneck for some benchmarks at 16 cores, as shown on the scalability graphs 14 through 19(b). Nevertheless, the performance improvement of the hardware support is still substantial, showing up to a 4.7× improvement with the MG kernel. In this case, the *No Manual, HW* variant is on par with the *Privatization* variant, which in the worst case is only 13% slower.

7.1.3. Detailed Model. Figures 20 through 25 present the results obtained using the detailed model. The detailed model supports a limited number of cores due to the

Fig. 15. Gem5 timing model: NAS Parallel Benchmark—FT class W.

Fig. 16. Gem5 timing model: NAS Parallel Benchmark—IS class W.

Fig. 17. Gem5 timing model: NAS Parallel Benchmark—MG class W.

Fig. 18. Gem5 timing model: matrix multiplication 512×512 . Fig. 19. Gem5 timing model: Sobel benchmark 4096×4096 .

Fig. 20. Gem5 detailed model: NAS Parallel Benchmark—CG class W.

Fig. 21. Gem5 detailed model: NAS Parallel Benchmark—FT class W.

Fig. 22. Gem5 detailed model: NAS Parallel Benchmark—IS class W.

Fig. 23. Gem5 detailed model: NAS Parallel Benchmark—MG class W.

Fig. 24. Gem5 detailed model: matrix multiplication 512×512 . Fig. 25. Gem5 detailed model: Sobel benchmark 4096×4096 .

relatively long simulation time (more than 35,000 CPU/hours were used to produce the results in this article). Even though the performance gain of the hardware support is more limited, it is still substantial, ranging from $1.2\times$ to $3.5\times$. The gain of the hardware support is reduced because the cores (with out-of-order capability) can improve performance through reordering the instructions and thus obviating part of the shared address manipulation activity.

Overall, our proposed hardware support for PGAS address mapping provides results that are comparable to, and in some cases better than, the manually optimized codes. Moreover, our approach does not require any modification or manual tuning of the UPC code to take advantage of the improved performance, thus enabling both the performance and productivity of PGAS languages.

7.2. PGAS Instruction Latency Sensitivity Study

In the previous section, we presented a detailed analysis of the PGAS hardware support, a fully pipelined hardware implementation that does not add additional instruction latency. However, there is a design trade-off between the instruction latency and the area of the chip used for the implementation. For example, a chip designer may prefer to reduce the area allocated for the new instructions by increasing the instruction latency (i.e., increasing the clock cycles/instructions to reduce the number of transistors used). To explore this trade-off, we analyzed the sensitivity of the hardware support's

Table IV. Effects of Load/Store Stall Cycles on Performance—CG

Th (#)	Atomic			Timing			Detailed		
	0/0	+1/0	+1/+1	0/0	+1/0	+1/+1	0/0	+1/0	+1/+1
1	1.20x	1.20x	1.20x	1.15x	1.15x	1.15x	1.09x	1.09x	1.08x
2	1.30x	1.30x	1.30x	1.22x	1.22x	1.22x	1.11x	1.10x	1.10x
4	1.41x	1.40x	1.40x	1.26x	1.26x	1.25x	1.04x	1.03x	1.03x
8	1.64x	1.62x	1.61x	1.34x	1.34x	1.34x	1.07x	1.07x	1.07x
16	1.89x	1.86x	1.85x	1.21x	1.20x	1.20x			
32	2.13x	2.08x	2.07x	1.11x	1.11x	1.11x			
64	2.62x	2.54x	2.53x	1.13x	1.13x	1.13x			

Note: 0/0 is the baseline, +1/0 adds one stall cycle for loads, and +1/+1 adds one stall cycle for both loads and stores.

Table V. Effects of Load/Store Stall Cycles on Performance—FT

Th (#)	Atomic			Timing			Detailed		
	0/0	+1/0	+1/+1	0/0	+1/0	+1/+1	0/0	+1/0	+1/+1
1	1.58x	1.55x	1.52x	1.44x	1.44x	1.43x	1.27x	1.24x	1.20x
2	2.27x	2.21x	2.15x	1.79x	1.78x	1.76x	1.45x	1.41x	1.37x
4	2.36x	2.30x	2.24x	1.75x	1.74x	1.73x	1.33x	1.30x	1.28x
8	2.34x	2.28x	2.22x	1.58x	1.57x	1.56x			
16	2.34x	2.28x	2.22x	1.21x	1.20x	1.20x			

Table VI. Effects of Load/Store Stall Cycles on Performance—IS

Th (#)	Atomic			Timing			Detailed		
	0/0	+1/0	+1/+1	0/0	+1/0	+1/+1	0/0	+1/0	+1/+1
1	1.84x	1.77x	1.66x	1.75x	1.73x	1.69x	1.23x	1.21x	1.18x
2	3.07x	2.95x	2.77x	2.44x	2.41x	2.36x	1.30x	1.28x	1.25x
4	3.44x	3.30x	3.09x	2.50x	2.47x	2.43x	1.34x	1.31x	1.28x
8	3.88x	3.70x	3.47x	2.40x	2.37x	2.34x			
16	3.74x	3.57x	3.38x	1.94x	1.93x	1.91x			
32	3.64x	3.50x	3.33x	1.37x	1.37x	1.36x			
64	3.12x	3.04x	2.94x	1.07x	1.07x	1.07x			

performance to increased instruction latency in the following cases: 0 extra cycles, 1 extra cycle for loads, and 1 extra cycle for both loads and stores. We model the extra cycles as a complete stall of the core, which is the worst-case performance degradation. The stall would most likely only impact the PGAS functional unit.

Tables IV through IX compare the speedup results of the *No Manual*, *HW* variant over the *No Manual* variant. The impact of the extra hardware cycles is quite small for CG, FT, IS, and MG. Matrix multiplication and Sobel showed a reduction in performance improvement for matrix multiplication from $3.45\times$ to $2.37\times$ and for Sobel a reduction in performance improvement from $2.74\times$ to $2.04\times$ using the gem5 detailed model with eight cores. The performance benefit of the proposed hardware support remains significant across all of the cases we studied.

7.3. FPGA Prototyping Results

Two microbenchmarks (vector addition and matrix multiplication) were implemented to verify the functionality and the performance of the hardware design implemented on the FPGA. They were compiled using Berkeley UPC 2.12.1 and GCC 4.4.2 for SPARC with all optimizations enabled (-O, -opt for BUPC, -O3 for GCC). Adding support for the extra shared address register file in GCC was prohibitively complex, so we did not develop a full prototype compiler as we did in the case of our gem5 simulation. Thus,

Table VII. Effects of Load and Store Stall Cycles on Performance—MG

Th (#)	Atomic			Timing			Detailed		
	0/0	+1/0	+1/+1	0/0	+1/0	+1/+1	0/0	+1/0	+1/+1
1	4.25x	3.56x	3.50x	3.65x	3.45x	3.43x	2.93x	2.21x	2.16x
2	6.39x	5.37x	5.29x	4.76x	4.51x	4.48x	3.03x	2.42x	2.37x
4	6.71x	5.65x	5.56x	4.63x	4.41x	4.39x	2.34x	1.99x	1.96x
8	6.58x	5.56x	5.48x	4.25x	4.06x	4.04x	1.61x	1.47x	1.45x
16	6.54x	5.54x	5.45x	3.11x	3.02x	3.01x			
32	6.31x	5.38x	5.30x	1.83x	1.80x	1.80x			
64	5.73x	4.97x	4.90x	1.29x	1.28x	1.28x			

Table VIII. Effects of Load and Store Stall Cycles on Performance—MM

Th (#)	Atomic			Timing			Detailed		
	0/0	+1/0	+1/+1	0/0	+1/0	+1/+1	0/0	+1/0	+1/+1
1	5.99x	5.00x	4.99x	5.14x	4.79x	4.79x	3.41x	2.34x	2.34x
2	7.10x	5.92x	5.92x	5.88x	5.48x	5.48x	3.45x	2.37x	2.36x
4	6.98x	5.82x	5.82x	5.88x	5.48x	5.48x	3.63x	2.49x	2.48x
8	7.05x	5.88x	5.88x	5.88x	5.48x	5.48x	3.45x	2.37x	2.37x
16	7.12x	5.93x	5.93x	5.83x	5.43x	5.43x			
32	6.95x	5.82x	5.82x	5.75x	5.37x	5.37x			
64	7.00x	5.86x	5.86x	5.61x	5.25x	5.25x			

Table IX. Effects of Load and Store Stall Cycles on Performance—Sobel

Th (#)	Atomic			Timing			Detailed		
	0/0	+1/0	+1/+1	0/0	+1/0	+1/+1	0/0	+1/0	+1/+1
1	2.70x	2.42x	2.38x	2.67x	2.55x	2.54x	2.01x	1.55x	1.50x
2	3.05x	2.73x	2.69x	2.97x	2.84x	2.82x	2.20x	1.69x	1.63x
4	3.06x	2.74x	2.69x	2.98x	2.85x	2.83x	2.20x	1.69x	1.63x
8	3.06x	2.74x	2.69x	2.96x	2.83x	2.81x	2.74x	2.11x	2.04x
16	4.61x	4.15x	4.08x	2.96x	2.84x	2.82x			
32	4.53x	4.08x	4.01x	2.95x	2.83x	2.81x			
64	3.00x	2.70x	2.66x	2.95x	2.82x	2.81x			

we manually inserted the PGAS hardware instructions to the assembly code of the compiled benchmarks.

In the vector addition benchmark, two integer vectors are added together. Results are shown in Figure 26. Both the *Privatization* variant and the *No Manual, HW* variant ran $3.5\times$ faster than the *No Manual* variant. The performance improvement diminishes as number of threads increases because this memory-intensive benchmark is able to quickly saturate the bandwidth-limited shared AMBA bus.

The matrix multiplication benchmark performs a 64×64 integer matrix multiplication. We used the same two manual optimizations (*Privatization 1*, *Privatization 2*) for the FPGA version of the benchmark that we used in the gem5 version. Figure 26 shows that the *No Manual, HW* variant matches the performance of the *Privatization 2* and was $8.4\times$ faster than the *No Manual* variant. The results are consistent with those obtained using the full-system simulator, an average $6\times$ improvement using the timing model because the Leon3 is an in-order processor. For both benchmarks, the *No Manual, HW* variant matches or surpasses the performance of *Privatization 2*.

The FPGA prototyping exercise gave us an opportunity to evaluate the chip area needed for the PGAS hardware support. Table X presents the FPGA resources used for a four-core Leon3 SMP system with and without the hardware support. Results are presented both in terms of the performance increase relative to the base Leon3

Fig. 26. Leon 3 with PGAS HW support.

Table X. Area Cost Evaluation for PGAS Hardware Support

Configuration	Slice Resources		BRAM		DSP48Es
	Registers	LUTs	18kb	36kb	
Leon3, 4 cores	46, 718	59, 235	106	34	16
Leon3, 4 cores + PGAS hardware support	49, 325	62, 572	126	34	24
Virtex 6—XC6VLX240T	301, 440	150, 720	832	416	768
Increase	2, 607	3, 337	20	0	8
Area increase, percentage of base	+5.6%	+5.6%	+18.9%		+50.0%
Area percentage of Virtex 6	+0.9%	+2.2%	+2.4%		+1.0%

implementation and in terms of the relative percentage of the FPGA chip used for the implementation. The area evaluation is very conservative, as we are comparing our implementation against a very simple processor core that does not include floating-point support. The BRAM and DSP48E increase may seem substantial as a percentage of the Leon3 but represents a tiny increase relative to the overall size of the Virtex 6 FPGA. Moreover, most of the additional BRAM and DSP48E instances are used to create the extra 64-bit register file for the 32-bit Leon3 (component not needed on an actual 64-bit architecture). The proposed hardware support mechanism for four cores utilizes less than 2.4% of the overall FPGA chip.

8. CONCLUSIONS

This work focuses on what we believe is presently the biggest impediment to the widespread adoption of PGAS languages: the manipulation of shared addresses, which creates a major performance penalty especially for local accesses. Although the PGAS shared memory programming model offers significant productivity improvements for the parallel programmer, the model's performance can degrade due to the overhead associated with accessing and traversing the shared memory space. Automatic compiler optimizations may mitigate this overhead, but they are often insufficient to provide competitive performance, especially for complex codes. Manual code optimizations are another mitigation technique that can achieve reasonable performance improvements, but this approach is labor intensive and diminishes programmer productivity.

We propose a novel processor hardware extension that can provide a transparent mechanism to reduce or eliminate the PGAS shared memory access overhead without creating an additional burden on the programmer. Using a prototype compiler, we demonstrated that the proposed instruction set extension is easily exploitable by compilers. It is important to note that the performance improvement of using the hardware instructions can surpass that of manually optimized code for two reasons. First, programmers have a tendency to focus manual optimization efforts on the shared pointers accessed from inner loops. Second, it is not always possible to manually optimize all shared pointers in the code (e.g., due to complex or random access patterns).

Experimental validation was conducted with the gem5 full-system simulator using seven major kernels: the five kernels of the well-accepted UPC NAS Parallel Benchmark suite, matrix multiplication, and a 2D Sobel edge detection benchmark. The results were consistently comparable to manually optimized code without the programmer overhead of hand tuning. Unmodified code compiled with our prototype compiler (using the proposed hardware support) achieved up to a $5.5\times$ performance gain over the same code compiled with full compiler optimizations running without our hardware support. In addition, implementation of the new hardware functionality requires a minimal increase in the overall chip area usage.

For our future work, we plan to explore hardware support for complex data structures such as the multidimension blocking in Chapel and X10. We are also evaluating hardware solutions that could provide further improvements to remote data accesses across a full cluster system of interconnected nodes. This would require extending the PGAS hardware instruction support to the network interface in conjunction with developing on-chip shared address management capabilities optimized for the PGAS programming model. A hierarchical approach to this would limit the cost of additional hardware and improve network performance by relying on shared addresses to quickly locate and communicate with other nodes. This will also create new opportunities not currently available to compilers for dynamic data aggregation and prefetching between remote threads.

ACKNOWLEDGMENTS

The authors would like to express their special thanks to David K. Newsom for his invaluable insights and editorial expertise.

REFERENCES

- Michail Alvanos, José Nelson Amaral, Ettore Tiotto, Montse Farreras, and Xavier Martorell. 2014. Reducing compiler-inserted instrumentation in Unified-Parallel-C code generation. In *Proceedings of the 2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'14)*. IEEE, Los Alamitos, CA, 270–277.
- Remzi H. Arpaci, David E. Culler, Arvind Krishnamurthy, Steve G. Steinberg, and Katherine Yelick. 1995. Empirical evaluation of the CRAY-T3D: A compiler perspective. In *ACM SIGARCH Computer Architecture News* 23, 2, 320–331.
- David H. Bailey, Tim Harris, William Saphir, Rob van der Wijngaart, Alex Woo, and Maurice Yarrow. 1995. *NAS Parallel Benchmarks 2.0*. Technical Report NAS-95-020. NASA Ames Research Center, Moffett Field, CA.
- Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The gem5 simulator. *SIGARCH Computer Architecture News* 39, 2, 1–7. DOI: <http://dx.doi.org/10.1145/2024716.2024718>
- Anastasiia Butko, Rafael Garibotti, Luciano Ost, and Gilles Sassatelli. 2012. Accuracy evaluation of gem5 simulator system. In *Proceedings of the 7th International Workshop on Reconfigurable Communication-Centric Systems-on-Chip (ReCoSoC'12)*. IEEE, Los Alamitos, CA, 1–7.

- François Cantonnet, Tarek El-Ghazawi, Pascal Lorenz, and Jaafar Gaber. 2005. Fast address translation techniques for distributed shared memory compilers. In *Proceedings of the 19th International Parallel and Distributed Processing Symposium*. 52b.
- François Cantonnet, Yiyi Yao, Smita Annareddy, Ahmed Mohamed, and Tarek El-Ghazawi. 2003. Performance monitoring and evaluation of a UPC implementation on a NUMA architecture. In *Proceedings of the International Conference on Parallel and Distributed Parallel Systems (IPDPS'03)*. 274.2.
- François Cantonnet, Yiyi Yao, Mohamed Zahran, and Tarek El-Ghazawi. 2004. Productivity analysis of the UPC language. In *Proceeding of the 18th International Parallel and Distributed Processing Symposium*. 254–260. DOI: <http://dx.doi.org/10.1109/IPDPS.2004.1303318>
- William W. Carlson, Jesse M. Draper, David E. Culler, Kathy Yelick, Eugene Brooks, and Karen Warren. 1999. *Introduction to UPC and Language Specification*. Technical Report. Center for Computing Sciences, Institute for Defense Analyses, Alexandria, VA.
- Bradford L. Chamberlain, David Callahan, and Hans P. Zima. 2007. Parallel programmability and the Chapel language. *International Journal of High Performance Computing Applications* 21, 3, 291–312.
- Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph Von Praun, and Vivek Sarkar. 2005. X10: An object-oriented approach to non-uniform cluster computing. *ACM SIGPLAN Notices* 40, 10, 519–538.
- Wei-Yu Chen, Dan Bonachea, Jason Duell, Parry Husbands, Costin Iancu, and Katherine Yelick. 2003. A performance analysis of the Berkeley UPC compiler. In *Proceedings of the 17th Annual International Conference on Supercomputing*, Vol. 4. ACM, New York, NY, 63–73. DOI: <http://dx.doi.org/10.1145/782814.782825>
- Barnaby Dalton, Gabriel Tanase, Michail Alvanos, George Almasi, and Ettore Tiotto. 2014. Memory management techniques for exploiting RDMA in PGAS languages. In *Proceedings of the Workshop on Languages and Compilers and Parallel Computing (LCPC'14)*.
- Martin Danek, Leos Kafka, Lukas Kohout, and Jaroslav Sykora. 2010. Instruction set extensions for multi-threading in LEON3. In *Proceedings of the 2010 IEEE 13th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS'10)*. IEEE, Los Alamitos, CA, 237–242.
- Mattias De Wael, Stefan Marr, Bruno De Fraine, Tom Van Cutsem, and Wolfgang De Meuter. 2015. Partitioned global address space languages. *ACM Computing Surveys* 47, 4, Article No. 62.
- Nan Dun and Kenjiro Taura. 2012. An empirical performance study of Chapel programming language. In *Proceedings of the 26th IEEE International Parallel and Distributed Processing Symposium Workshops and PhD Forum (IPDPSW'12)*. IEEE, Los Alamitos, CA, 497–506.
- Kemal Ebcioglu, Vivik Sarkar, Tarek El-Ghazawi, and John Urbanic. 2006. An experiment in measuring the productivity of three parallel programming languages. In *Proceedings of the Workshop on Productivity and Performance in High-End Computing (P-PHEC'06)*. IEEE, Los Alamitos, CA.
- Tarek El-Ghazawi and François Cantonnet. 2002. UPC performance and potential: A NPB experimental study. In *Proceedings of the ACM/IEEE Conference on Supercomputing*. IEEE, Los Alamitos, CA, 1–26.
- Tarek El-Ghazawi, François Cantonnet, Yiyi Yao, Smita Annareddy, and Ahmed S. Mohamed. 2006. Benchmarking parallel compilers: A UPC case study. *Future Generation Computer Systems* 22, 7, 764–775. DOI: <http://dx.doi.org/10.1016/j.future.2006.02.002>
- Tarek El-Ghazawi, William Carlson, Thomas Sterling, and Katherine Yelick. 2005. *UPC: Distributed Shared Memory Programming*. Wiley.
- Tarek El-Ghazawi and Sébastien Chauvin. 2001. UPC benchmarking issues. In *Proceedings of the International Conference on Parallel Processing (ICPP'01)*. IEEE, Los Alamitos, CA, 365–372.
- Holger Fröning and Heiner Litz. 2010. Efficient hardware support for the partitioned global address space. In *Proceedings of the IEEE International Symposium on Parallel Distributed Processing, Workshops, and PhD Forum (IPDPSW'10)*. 1–6. DOI: <http://dx.doi.org/10.1109/IPDPSW.2010.5470851>
- David Grove, Josh Milthorpe, and Olivier Tardieu. 2014. Supporting array programming in X10. In *Proceedings of the ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming (ARRAY'14)*. ACM, New York, NY, Article No. 38. DOI: <http://dx.doi.org/10.1145/2627373.2627380>
- Pierre Guironnet De Massas and Paul Amblard. 2006. Experiments around SPARC Leon-2 for MPEG encoding. In *Proceedings of the International Conference on Mixed Design of Integrated Circuits and System (MIXDES'06)*. 285–289. DOI: <http://dx.doi.org/10.1109/MIXDES.2006.1706585>
- Matthias M. Mueller. 2000. Efficient address translation. *Interner Bericht. Universität Karlsruhe, Fakultät für Informatik* 2000, 12.
- Steven L. Scott. 1996. Synchronization and communication in the T3E multiprocessor. *ACM SIGPLAN Notices/SIGOPS Operating Systems Review* 30, 5, 26–36.

- Olivier Serres, Ahmad Anbar, Saumil G. Merchant, Abdullah Kayi, and Tarek El-Ghazawi. 2011a. Address translation optimization for Unified Parallel C multi-dimensional arrays. In *Proceedings of the 16th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'11)*. IEEE, Los Alamitos, CA.
- Olivier Serres, Vikram K. Narayana, and Tarek El-Ghazawi. 2011b. An architecture for reconfigurable multi-core explorations. In *Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig'11)*.
- UPC Consortium. 2005. UPC Language Specifications V1.2. (May 2005).
- UPC NPB. 2014. UPC NAS Parallel Benchmarks threads.seas.gwu.edu/sites/npb-upc. Retrieved November 11, 2015, from http://www.gwu.edu/~upc/docs/upc_specs_1.2.pdf.
- Katherine Yelick, Dan Bonachea, Wei-Yu Chen, Phillip Colella, Kaushik Datta, Jason Duell, Susan L. Graham, Paul Hargrove, Paul Hilfinger, Parry Husbands, Costin Iancu, Amir Kamil, Rajesh Nishtala, Jimmy Su, Michael Welcome, and Tong Wen. 2007. Productivity and performance using partitioned global address space languages. In *Proceedings of the International Workshop on Parallel Symbolic Computation (PASCO'07)*. ACM, New York, NY, 24–32.
- Zhang Zhang and Steven R. Seidel. 2005. Benchmark measurements of current UPC platforms. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (PMEOPDS'05)*.

Received May 2015; revised September 2015; accepted October 2015