Title
Evaluating Support for Global Address Space Languages on the Cray X1

Authors
Bell, C
Bonachea, D
Chen, WY
et al.

Publication Date
2004-06-26

DOI
10.1145/1006209.1006236

Peer reviewed

Evaluating Support for Global Address Space Languages on the Cray X1

Christian Bell
Computer Science Division
University of California at Berkeley

Wei-Yu Chen
Computer Science Division
University of California at Berkeley

Dan Bonachea
Computer Science Division
University of California at Berkeley

Katherine Yelick
Computer Science Division
University of California at Berkeley

Abstract

The Cray X1 was recently introduced as the first in a new line of parallel systems to combine high-bandwidth vector processing with an MPP system architecture. Alongside capabilities such as automatic fine-grained data parallelism through the use of vector instructions, the X1 offers hardware support for a transparent global-address space (GAS), which makes it an interesting target for GAS languages. In this paper, we describe our experience with developing a portable, open-source and high performance compiler for Unified Parallel C (UPC), a SPMD global-address space language extension of ISO C. As part of our implementation effort, we evaluate the X1's hardware support for GAS languages and provide empirical performance characterizations in the context of leveraging features such as vectorization and global pointers for the Berkeley UPC compiler. We discuss several difficulties encountered in the Cray C compiler which are likely to present challenges for many users, especially implementors of libraries and source-to-source translators. Finally, we analyze the performance of our compiler on some benchmark programs and show that, while there are some limitations of the current compilation approach, the Berkeley UPC compiler uses the X1 network more effectively than MPI or SHMEM, and generates serial code whose vectorizability is comparable to the original C code.

Categories and Subject Descriptors

C.4 [**Hardware**]: Performance of Systems – performance attributes

General Terms

Performance, Languages, Experimentation

ICS'04, June 26–July 1, 2004, Malo, France.

Keywords

UPC, X1, global address space

1. Introduction

Global Address Space (GAS) languages have recently emerged as a promising alternative to the traditional message passing model for parallel applications. Designed as parallel extensions for popular sequential programming languages, GAS languages such as UPC [15], Titanium [29, 16], and Co-Array Fortran [22] provide better programmability through the support of a user-level global address space, leading to more flexible remote accesses through language-level one-sided communication. GAS languages thus offer a more convenient and productive programming style than explicit message passing (e.g., MPI [21]), and good performance can still be achieved because programmers retain explicit control of data placement and load balancing. Another virtue of GAS languages is their versatility; while it has not yet reached the level of MPI's ubiquity, UPC implementations are now available on a significant number of platforms, ranging from multiprocessors to the many flavors of networks of workstations.

Meanwhile in the architectural world of supercomputing, parallel vector systems (led by the Earth Simulator [13] and the Cray X1 System [10]) are mounting a comeback to challenge the dominance that superscalar microprocessors have established in the last decade. By effectively exploiting fine-grained data parallelism through vector arithmetic instructions, these vector architectures offer the potential to narrow the growing gap between sustained and peak performance for scientific applications [23]. With its unique distinction of delivering powerful vector processing over non-uniform shared memory hardware, the Cray X1 in particular presents an interesting target platform for GAS languages. In addition to simplifying communication operations as direct reads and writes to remote memory locations, the system's raw performance is impressive both in terms of communication (peak memory bandwidth and low communication latency) as well as computation (powerful vector pipelines). Furthermore, its efficient hardware support for strided accesses and scatter/gather memory operations has the potential to substantially reduce overheads associated with fine-grained remote accesses. Such an array of features would appear to be quite suitable for languages such as UPC that adopt a global address space memory model.

This paper describes our experiences in implementing and tuning the portable Berkeley UPC compiler [2] for the Cray X1 system. Berkeley UPC is the first open source, portable, and high-performance GAS language implementation on the X1, and the lessons we learned from this language implementation study should be useful not only for

UPC but also the Global Address Space language community in general. Our experiences demonstrate the potential of the X1 architecture, but also expose areas where more effort is required before it can be viewed as an ideal architecture for UPC and GAS languages in general. While many of the language primitives can be implemented directly using the hardware global pointer support, the absence of a rich set of user-level communication primitives, in particular the X1's lack of per-operation completion guarantees, limits the opportunities for compiler optimizations and the system's extensiblity via third-party libraries. Similarly, the heavy reliance on vectorization to achieve reasonable performance also increases the difficulties of performance tuning for compiler implementors. In particular, each layer in portable compilers such as Berkeley UPC must be tuned to pay careful attention to vectorization constraints, which places a relatively heavy burden across the entire software stack. As the hardware and system software matures, we expect to see performance improvements as well as more flexible support for portable implementations of GAS languages.

The rest of the paper is organized as the follows. Sections 2 and 3 describe the Cray X1 system, UPC, and the Berkeley UPC compiler. Section 4 details our implementation of the communication operations, which satisfy the basic requirements for a functioning UPC compiler on the X1. Sections 5 and 6 detail our efforts at tuning the performance of the Berkeley UPC compiler for the X1's unique architecture: the former summarizes our optimizations for shared memory accesses, while the latter discusses our strategy for achieving good serial performance. Section 7 analyzes the impact of the X1's tightly-coupled design on our portable architecture and on the effectiveness of compiler communication optimizations. Section 8 evaluates our compiler's parallel performance, and finally sections 9 and 10 conclude the paper with an evaluation of the X1's architectural support for GAS languages.

2. The Cray X1

The X1 [10] is a supercomputer system developed by Cray which combines powerful vector processors with high memory and network interconnect bandwidth. In order to sustain high bandwidth vector processing, the X1 is based on previous MPP Cray designs that emphasized memory bandwidth, and features some more recent vector concepts such as multi-streaming and vector caching. The system uses a network interconnect reminiscent of the Cray T3E to connect Cray nodes in order to unite long, latency-tolerant vector computations with the scalability to be expected from MPPs.

Figure 1 illustrates the architecture of a single Cray X1 node, the basic building block of the system. Each node consists of four multi-streaming processors (MSPs) and a flat, shared 16GB physical memory. Each MSP in turn is composed of four single-streaming processors (SSPs), each with two vector pipelines and one scalar processor. The four SSPs also share a 2MB data "E-Cache", which helps supply enough memory bandwidth to saturate the vector units. As is the case with many vector platforms, applications whose critical paths do not vectorize tend to exhibit poor performance; in addition to operating at twice the clock speed, the ability of the vector units to overlap memory operations with computation makes the Cray X1's vector units significantly more powerful than the scalar pipeline. The X1 offers two configurations for program execution. Explicit parallelism is achieved in the SSP mode by treating each SSP as a separate processor, such that each node essentially behaves as a 16-way SMP. The alternative MSP mode maps each execution thread to an MSP, and utilizes compiler-directed *multi-streaming* transformations to accomplish automatic parallelization of computational loops across the constituent SSP hardware. The multi-streaming process divides either vectorized inner loops or unvectorized outer loops into four independent segments, and assigns them to different SSPs to be executed in parallel. An early performance evaluation of the Cray X1 [12] suggests

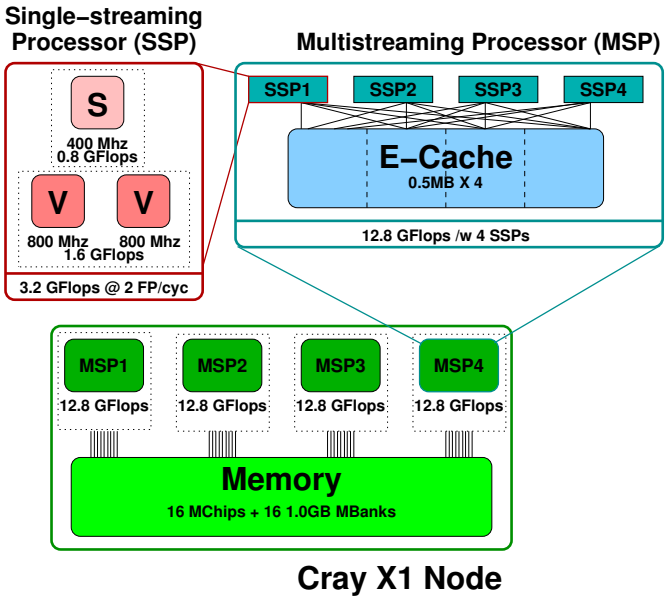

Figure 1. Cray X1 single node: Each MSP contains 4 SSPs each with 2 vector and 1 scalar unit

that many parallel applications can achieve significant performance on the machine, given sufficient porting and optimization efforts. The benchmark results reported in this paper are collected on a four node X1 system at Cray (a total of 48 SSPs*), running Unicos/MP version 2.4 and Cray C version 5.1.0.5.

3. Unified Parallel C

UPC (Unified Parallel C) is a parallel extension of the C programming language aimed at supporting high performance scientific applications. The language adopts the SPMD programming model, such that every thread runs the same program but keeps its own private local data. In addition to each thread's private address space, UPC provides a shared memory area to facilitate implicit communication amongst threads, and programmers can create shared objects through the use of the `shared` type qualifier or the dynamic shared memory allocation library functions. While a private object may only be accessed by its owner thread, all threads can read or write objects in the shared address space. Because the shared memory space is logically divided among all threads, from a thread's perspective the shared space can be further divided into a local shared memory and remote one. Data located in a thread's local portion of the shared space are said to have "affinity" with the thread, and compilers can utilize affinity information to exploit data locality in applications and help reduce communication overhead.

UPC gives users direct control over shared data placement through distributed arrays. When creating a shared array, programmers specify a block size in addition to the dimension and element type, and the system uses this value to distribute the array elements block by block in a round-robin fashion over all threads. For example, a declaration of `shared [2] int ar[10]` tells the compiler to allocate the first two elements of `ar` on thread 0, the next two on thread 1, and so on. If the block size is omitted the value defaults to one (cyclic layout), while a layout of `[]` or `[0]` indicates indefinite block size, i.e., that the entire array should be allocated on a single thread. A pointer-to-shared thus needs three logical fields to fully represent the address of a shared object: `address`, `thread_id`, and `phase`.

*one node is reserved for system tasks

The `thread_id` indicates the thread that the target has affinity to, the `address` field stores some representation of the object's "local" address on the thread, while the `phase` field gives the offset of the target within the current block. Other notable UPC features include a `upc_forall` parallel loop, block transfer library functions, synchronization constructs, and flexible language-level control of the memory consistency model; consult the UPC language specification for more details [15].

3.1 The Berkeley UPC Compiler

Prior to our work, the only UPC implementation for the X1 was the Cray C compiler's UPC extensions, which currently only provide support for a subset of the UPC language specification. Important missing features include block cyclic pointers, `upc_forall` loops, non-collective shared memory allocation, and restrictions on the block size of shared arrays. While some of the deferred features merely offer syntactic convenience, many provide essential functionality for UPC applications and have no easy workarounds without impacting program design. Their exclusion thus severely limits the usefulness of the Cray UPC compiler, which in our experiments fails to compile several of the NAS UPC benchmarks. Our goal is thus to implement an open-source and portable compiler that performs comparably to Cray UPC and is fully compliant with the latest UPC 1.1.1 specification.

Figure 2 shows the overall structure of the Berkeley UPC compiler [2], which is divided into three components: the UPC-to-C translator, the UPC runtime system, and the GASNet communication system [4]. During the first phase of compilation, the Berkeley UPC compiler preprocesses and translates UPC programs into ISO-compliant C code in a platform-independent manner, with many UPC-related parallel features converted into calls to the runtime library. The translated C code is then compiled using the target system's C compiler and linked to the runtime system, which performs initialization tasks such as thread generation and shared data allocation. The Berkeley UPC runtime delegates communication operations to the GASNet communication layer, which provides a uniform interface for low-level communication primitives on all networks.

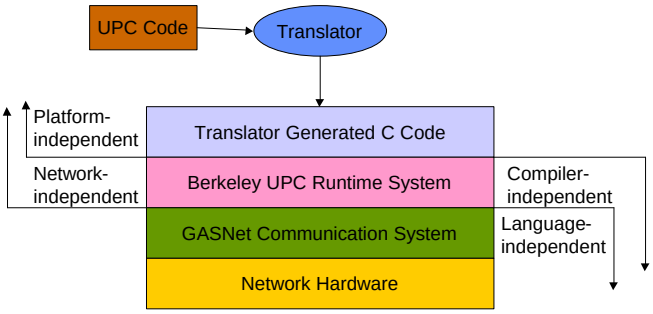

Figure 2. Architecture of the Berkeley UPC compiler

We believe this three-layer design has several advantages. First, because of the choice of C as our intermediate representation, our compiler can be made available on any UNIX platform that has an ISO-compliant C compiler; most other currently available UPC compilers generate assembly language and therefore only support systems with specific CPU architectures. Second, both the UPC runtime system and GASNet implement a well-defined public interface: the runtime offers a flexible pointer-to-shared abstraction with the option of running multiple threads per node and GASNet implements network-independent global-address space communication primitives. This two-tier approach can be tailored to move more or less functionality into the runtime or GASNet based on how close either layer can target native communication primitives. In a previous work [7], we have validated our design by showing that, in spite of the modularity used to support portability, the Berkeley UPC compiler performs well on today's high-performance clusters. However, our compilation strategy finds an interesting challenge in the X1, whose compiler and application software is very tightly integrated with the hardware. The next sections underline how the major components of our architecture were adapted in order to maintain our goals of both portability and high performance on the X1.

4. Porting the Berkeley UPC Compiler to the Cray X1

This section describes our initial efforts in porting the Berkeley UPC compiler to the X1. The modular design of our compiler greatly simplifies the porting process for supporting new architectures; generally no changes are required for the translator, whose code generation is entirely platform-independent, with the exception of a few general architectural parameters such as register size and the integral type width. The implementation of the communication operations is the system component which is generally most sensitive to platform characteristics, and therefore this functionality has been encapsulated entirely within the GASNet implementation for each platform. Consequently, despite the fact the X1's architecture differs substantially from other systems we have targeted, we were able to build a working implementation of the Berkeley UPC compiler on the system in about one week.

4.1 The GASNet Communication Layer

The main purpose of GASNet is to provide a portable, language-independent and high-performance communication interface. Designed primarily as a compilation target, GASNet incorporates a set of network communication primitives crafted to provide high levels of performance and expressiveness tailored for code generation, in contrast to end-user library interfaces such as MPI that prioritize other design goals such as interface generality/minimality, code readability and universal interoperability. As a result, GASNet delivers communication performance very close to the native hardware peak across many systems, while leveraging platform and network-specific features (such as RDMA support and/or block transfer engines).

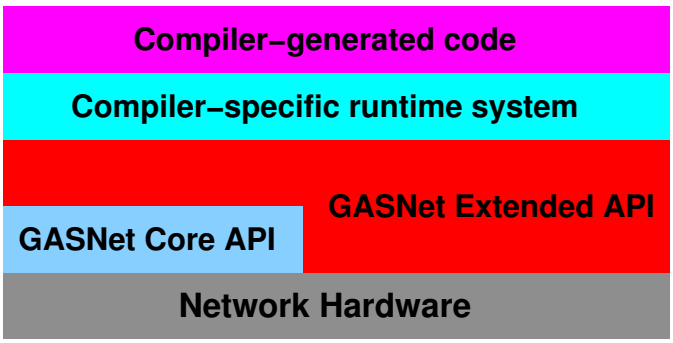

Figure 3. GASNet communication system: the narrow, AM-based Core API is sufficiently general to implement the entire system, but can also be bypassed to implement functionality from the Extended API directly on the underlying network to exploit available hardware support

Figure 3 illustrates the basic abstraction stack of the Berkeley UPC and Titanium compilers over GASNet. The existing GASNet infrastructure greatly simplifies the porting effort for new platforms. Building upon a provided "fill-in-the-blanks" template framework, implementors are encouraged to proceed in a two-stage porting process. A

complete working GASNet implementation can be obtained entirely in the first phase by implementing the intentionally narrow but general GASNet Core API, whose design is based heavily on Active Messages [19]. The wider and more expressive interface of GASNet, the Extended API, is already available as a reference implementation written solely in terms of the Core and can be used to provide full GASNet functionality over the ported Core. Second, primitives available in the reference implementation of the Extended API can be selectively replaced with more efficient network primitives offered by the underlying networking software or hardware. Based on prior experience in porting GASNet to five other networks, we have found this approach to be very effective in quickly obtaining a working conduit and gradually refining it with more efficient primitives.

4.2 Porting GASNet to the X1

As a portable communications interface with well-defined semantics, GASNet's ability to provide an optimal implementation for a particular platform depends mostly on what the target system exposes in terms of network features and how these features can be leveraged using existing target software interfaces. Since loosely coupled platforms are typically based on a some form of messaging, higher-level software layers often have much more control over initiation and completion of remote memory operations than on platforms where global memory reads and writes are transparent to the user. The Cray X1, with its transparent global memory support falls into the latter category, which constitutes a departure from previous messaging-based Cray hardware. In the previous family of Cray MPP designs, the Cray T3D and T3E systems provided programmers with user-level communication primitives. Communication on the T3D could be performed using three mechanisms: a prefetch queue for individual loads and stores, a *memory centrifuge* facility for global memory access, and a block transfer engine for large asynchronous transfers. The T3E provided these communication mechanisms as well as extended support for synchronization and collective operations, encompassed in a set of general-purpose user-level network registers (E-registers) [25]. These systems were successful in part because of their high performance and their support for programming models characterized by low cost asynchronous communication enabled by E-registers or lower-level programming interfaces such as *shmem* [26]. Although these systems predate the Berkeley UPC compiler, their user-level messaging interface would allow GASNet to exploit much of their functionality for efficient fine-grained control over communication. In contrast, the X1 adopts an approach akin to shared memory platforms whereby communication, whether scalar or vector, is enabled exclusively through assembly-generated load/store instructions. This hardware interface offers arguably better programmability to end users, but it unfortunately prevents code generators and communication software layers from maximizing hardware utiliization by explicitly controlling the specific parts of communication scheduling that involve initiation and synchronization. Many of the standard high-level communication optimizations for hiding network latencies through communication overlap are difficult to achieve without more explicit control over the communication hardware.

4.2.1 *shmem* as a GASNet target

As the first stage of the GASNet porting strategy, we have targeted the *shmem* communication interface as a general mechanism for implementing the GASNet Core. After sufficiently experimenting with the system, we were able to complete the Core and use GASNet's Extended reference implementation to obtain a complete GASNet X1 implementation in a matter of days. While tuning the X1 port of the GASNet Extended API, we again considered the *shmem* interface as a potential target, because Cray has been promoting the interface as being the best communication interface for low latency and high bandwidth communication [24]. While we have found *shmem* useful for quickly prototyping a functional GASNet Core, we encountered several deficiencies in *shmem* while considering it for implementing the Extended API primitives – specifically, *shmem* offers only blocking versions of the *get* operation, lacks some expressiveness in its synchronization mechanisms and presents an additional source of library call overhead for small messages. More importantly, when *shmem* library calls appear in any loop structure, the Cray C compiler turns off automatic vectorization optimizations. This weakens the utility of *shmem* as a programming interface for any interesting programming style other than bulk synchronous MPI-style, where communication is reduced to moving large amounts of data between processing elements and carrying out computation exclusively over local data. Obviously, we would like to leverage the expressive power of GAS languages and the support for fast remote accesses and global non-unit stride accesses on the X1, without requiring programmers to adopt a clumsy bulk synchronous programming style in order to achieve vectorization.

4.2.2 Hardware Global Pointers as a GASNet target

In considering another possible target for GASNet, we modified the Extended API to take advantage of the properties of the X1 global virtual addresses and memory centrifuge (illustrated in figure 6). When global memory is allocated on the X1 symmetric heap using a collective memory allocation call, the memory segment returned to each caller contains the caller's processing element (PE) number in the high-order bits of the pointer representation and is mapped at corresponding virtual memory locations in each PE's address space. Since GASNet allocates a segment on each node at initialization, the segment can be allocated from the symmetric heap and the resulting segment addresses can be globally published to GASNet clients and used without indirection or translation. Most importantly, by allowing GASNet put/get operations to be fully inlined, this newer version of the Extended API side-steps the limitation the vectorizer imposes on the presence of function or library calls. We believe that this refined approach is reasonable for our portable system, as it tailors the GASNet implementation to the platform without changing the interface and gains the most from our mismatch with the Cray compilation strategy. Although we were able to overcome some of the constraints imposed by the vectorizer, we could not sufficiently integrate GASNet and X1 communication to our satisfaction due to the lack of support for inline assembly in the Cray C compiler. GASNet's current major clients are source-to-source GAS language translators that do not participate in any platform-specific code generation, and are thus dependent on the amount of functionality that existing compilation and system software infrastructures are willing to expose. We expect other third-party software packages that integrate a fair amount of complexity in their runtime and communication components to suffer from this limitation as well.

The performance results of our tuning efforts are compared to other X1-specific communication libraries in figures 4 and 5 for individual one-sided put and get operations. Since puts are translated into store instructions, the put message gap corresponds to the amount of time during which the processor is tied up injecting each global store into the network. Furthermore, the get operations correspond to blocking library calls under *shmem*, but are compiled to simple assembly load instructions under GASNet, which gives the processor more opportunities to overlap outstanding loads in its load queue. Aside from 1-byte sized messages that necessitate read-modify-write operations, the GASNet performance significantly exceeds that of MPI and trims roughly two microseconds off *shmem* performance for latency-sensitive small message operations. The performance improvement over shmem is primarily due to the removal of address translation and library call overheads from the critical path. For larger messages (starting at 80 bytes), the `bcopy()` library call provided by Cray gives the best performance, and is used by both layers.

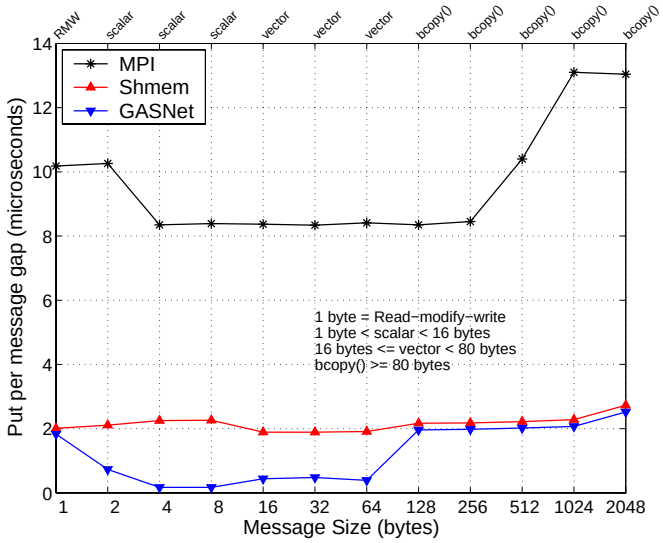

Figure 4. Small put performance

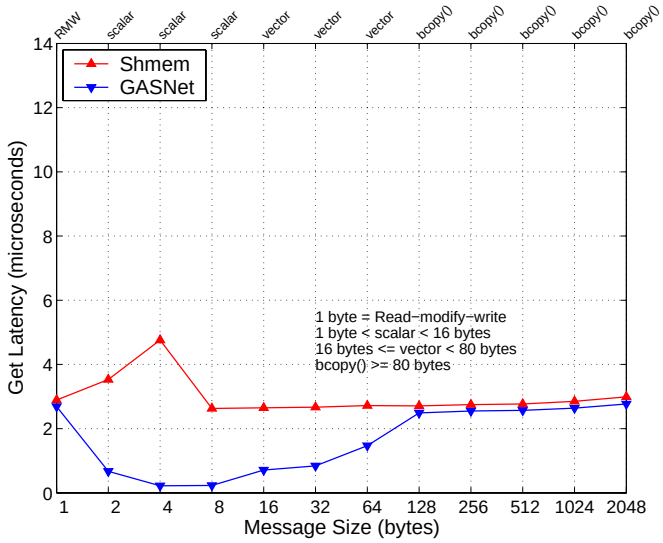

Figure 5. Small get performance

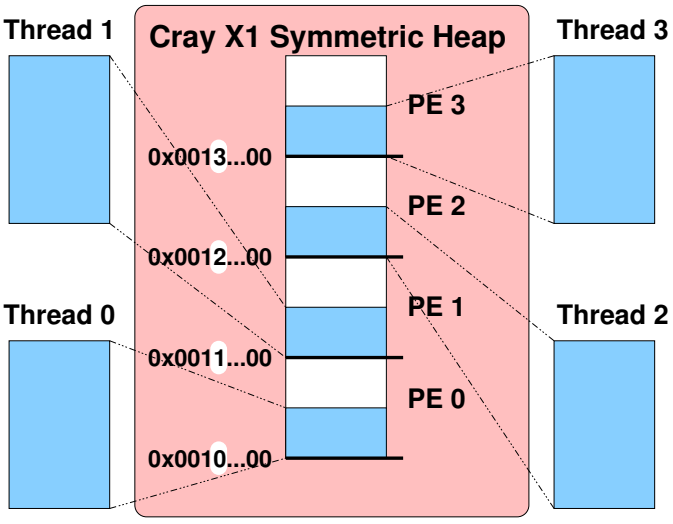

Figure 6. The Cray X1 *memory centrifuge*

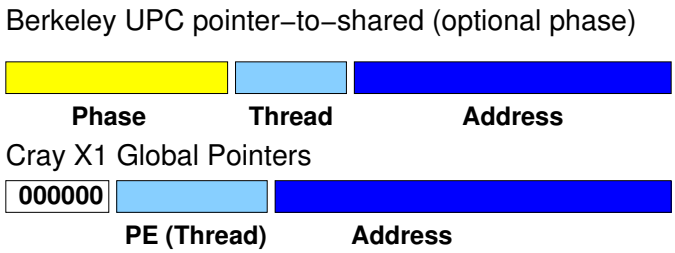

Figure 7. Cray global pointer and Berkeley UPC pointer-to-shared representations

5. Tuning the UPC Runtime System for the X1

Having described our implementation of an efficient communication layer for GAS languages on the X1, we now discuss our strategy for implementing UPC's shared memory accesses. Compared to regular C pointers, a generic UPC pointer-to-shared logically contains two additional `thread_id` and `phase` fields. Both fields are generally updated while manipulating a pointer-to-shared, making such operations inevitably slower than local pointer arithmetic. To overcome this overhead, the Berkeley UPC compiler implements an optimization for the important special case of "phaseless" pointers, namely those with a cyclic distribution where the block size is 1 element (and the phase field is always zero) or an indefinitely blocked distribution where the pointer always has affinity to a single thread (and the phase is defined to be zero). Cyclic and indefinite pointers are thus "phaseless", an important static property that allows our compiler to generate significantly more efficient pointer manipulation arithmetic for these types. Experimental results [7] show this approach to be effective in improving the performance of pointer-to-shared arithmetic, removing 50% of the overhead from cyclic pointer arithmetic and making indefinite pointers almost as fast as regular C pointers for pointer-integer addition.

5.1 Pointer-to-shared representation for the Cray X1

Given the success of our phaseless pointer optimization, we naturally want to exploit some of the properties common to our pointer representation and X1 global pointers. The first step in tuning UPC's shared memory accesses is to ensure that the pointer-to-shared representation deviates from the Cray X1's global pointers as little as possible. For multi-node applications, Cray's notion of processing elements exactly matches that of UPC threads, whereby each thread is given a distinct address space and the Cray symmetric heap can be used to provide per-thread UPC global shared and local heaps. As previously explained, the UPC thread id or virtual PE number can easily be extracted from each Cray global virtual address, which allows this representation of UPC phaseless pointers to exactly match the hardware's global pointers. This approach eliminates the overhead of a pointer translation step, and additionally allows the Cray C compiler to optimize our generated shared accesses to cyclically and indefinitely distributed data as if they were regular C pointer dereferences. Generic pointers-to-shared present more obstacles, as UPC semantics require that phase information can be extracted from arbitrary pointers-to-shared to permit easy indexing into the beginning of a block. The phase field is thus an intrinsic component of pointers to block-cyclically distributed shared data, and must be explicitly stored in the pointer construct. The representations for phased and phaseless pointers-to-shared as well as Cray global pointers are shown in figure 7.

5.2 Vectorizing Shared Memory Accesses

Once the appropriate representations for pointer-to-shared were cho-

sen, we carefully tuned the shared memory access primitives to exclude constructs that could interfere with vectorization. All function calls to the runtime and GASNet which occur in critical paths are either fully inlined or replaced with macros, and GASNet translates the common case of 4/8 byte transfers into simple pointer assignments and dereferences. Since the runtime supports running UPC threads over hierarchical node configurations, such as clusters of SMPs, it is also responsible for translating operations on a pointer-to-shared's thread affinity into local or remote accesses. However, the carefully-designed infrastructure of the runtime allows the execution-time cost for determining local or remote affinity to be eliminated entirely for platforms such as the X1 that feature global address space hardware.

An interesting issue for vectorizing shared memory accesses arises in implementing blocking put operations, whose semantics require that the value being stored be completely written to the destination address prior to returning. The Cray X1, however, does not provide hardware support that polls for the completion of remote writes, and the only alternative for mimicking this behavior is to issue a global memory barrier that enforces global ordering of all prior references before all subsequent references. Not only is the global memory barrier overkill when all that is needed is the guarantee of the completion of a single access, but the presence of such a barrier immediately inhibits all automatic vectorization of the enclosing code. Our solution is to take advantage of UPC's relaxed consistency model to eliminate the memory barrier altogether for relaxed writes. UPC supports both a strict and a relaxed memory model, and relaxed shared memory accesses can be freely reordered as long as local data dependencies are still preserved [28]. Since the Cray X1 maintains the program order for two scalar references to overlapping locations, correct local data dependencies will be maintained, and there is therefore no need for explicit instructions to enforce the completion of a put operation. This allows the Cray C compiler to freely vectorize scalar memory references and schedule synchronizations as necessary. While strict accesses require stronger ordering guarantees and thus do not benefit from this optimization, they occur with significantly lower frequencies in real applications and therefore are much less performance-critical.

5.3 Scalar Performance Microbenchmarks

In order to examine the execution overheads of the system, we measured the scalar overhead of various UPC shared memory operations for both the Berkeley UPC and Cray UPC compilers. The numbers reported here represent an upper bound on communication overhead for applications whose fine-grained remote accesses could not be vectorized. Figure 8 presents the execution time of the pointer-to-shared manipulation functions, while Figure 9 presents the respective memory access time.

As the results show, the Berkeley UPC compiler offers competitive performance on pointer-to-shared arithmetic; block cyclic (generic) pointers, in particular, demonstrate overhead comparable to that of cyclic pointers, indicating there is little performance incentive for Cray UPC to omit support for block cyclic pointers. The execution time of UPC shared remote accesses is very close to GASNet's get/put latencies, signifying the overhead incurred by the runtime layer is very low. A substantial difference in performance is also observed between blocking and non-blocking remote puts, which can be attributed to the cost of the global memory barrier that is included with each blocking put operation, but can be amortized over many non-blocking put operations.

6. Sequential Performance

The popular GAS languages are designed as parallel extensions of sequential programming languages, and UPC is no exception; a thread's local computation in its private address space is generally written in language very similar to ordinary C code, and therefore uniprocessor execution time is an important criteria in evaluating a UPC compiler's performance [14]. From previous work [7] we have discovered that despite a source-to-source translation from UPC to C, our compiler still delivers good serial performance on conventional superscalar architectures. Cray X1's dramatically different architectural approach, however, challenges this observation by making vectorization the dominant factor for achieving high performance. Although our translator preserves the semantics of the sequential portions of the program, the output will not be syntactically identical to the program source, due to optimizations performed by the translator and the lack of a one-to-one mapping between its intermediate representation and C. Furthermore, the Cray C compiler's vectorizer is highly sensitive to small changes in inner loop expressions; our experiments have identified several constructs that tend to inhibit a loop's vectorization, such as function calls, type casts, and access to global variables in the presence of pointer arithmetic. One important topic in optimizing UPC application performance for the Cray X1 thus involves investigating if our code generation process can be extended to minimize interferences with the C compiler's ability to automatically vectorize application code.

6.1 Implementation Approach

Our goal in this section is to evaluate the serial performance of the Berkeley UPC compiler, concentrating on its ability to maintain the vectorizability of the sequential portion of the program. With full optimizations enabled, the Cray C compiler [9] performs automatic vectorization on expressions inside a loop that it detects to be free of cycles of dependences, after applying vectorization-enabling transformations such as inlining, loop splitting, and loop interchange. The compiler also vectorizes certain special recurrences such as reduction and scatter/gather. Cray C provides two program-level techniques to assist the compiler's alias and dependence analysis: `restrict` pointers and the pragmas that declare a loop to be free of vector dependences or recurrences between array accesses. As such, our strategy is to keep the translated output as syntactically similar as possible to the original source. The level of the intermediate representation is kept sufficiently high such that C loops are preserved in their original form. Similarly, array expressions are recognized and handled specially by the translator, both to allow for more aggressive transformations by its optimizer and to provide the C compiler with more precise information. Multidimensional array accesses are preserved in their original form instead of being linearized into one dimensional arrays. Because the Berkeley UPC compiler complies with the ISO C99 standard [5], it already supports `restrict`-qualified pointers, and additionally UPC source-level vectorization pragmas are accepted by the translator and appear unchanged in the same relative location in the generated C output. We are also currently implementing optimizations in the translator that will leverage the semantic information available at UPC source level to identify some vectorizable loops and automatically generate the appropriate pragmas in the output.

6.2 Livermore Kernels

We chose the C version of the Livermore Kernels [20] to evaluate the serial performance of our compiler. The Livermore Loops consist of 24 sequential computation loops extracted from common scientific applications, and should closely reflect the sequential computational performance offered by our compiler. In particular, the X1's reliance on the vector unit to achieve both fast computation and high memory bandwidth means that application performance will often hinge on whether the main computation loops can be efficiently vectorized. In this test, we do not supply any vectorization pragmas and do not perform any manual transformations, as our goal is to test if the translation process interferes with Cray C's automatic vectorization. Table 1

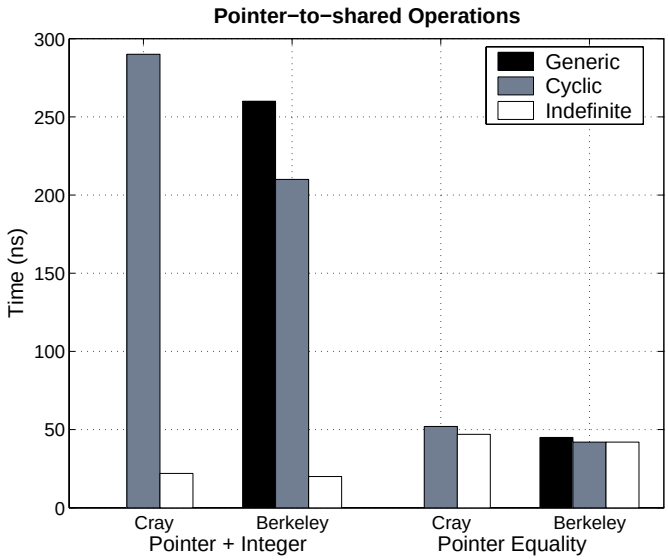

Figure 8. Performance of pointer-to-shared arithmetic – results for generic pointers are missing for Cray UPC, since it does not support block cyclic pointers-to-shared

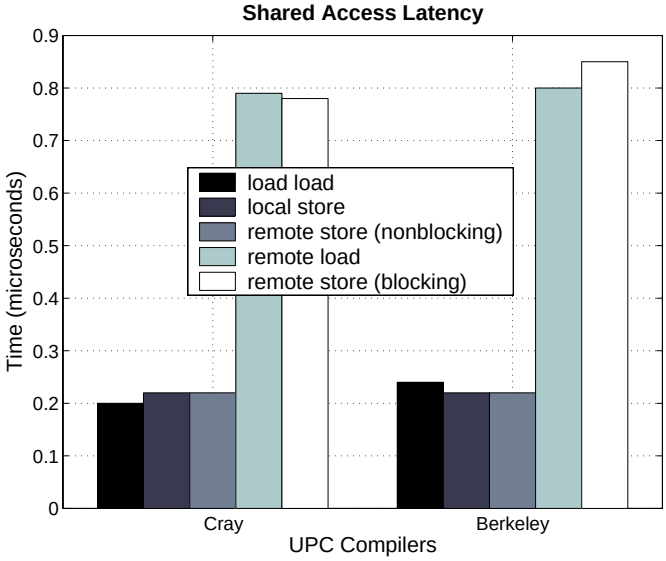

Figure 9. Execution time of UPC shared memory access

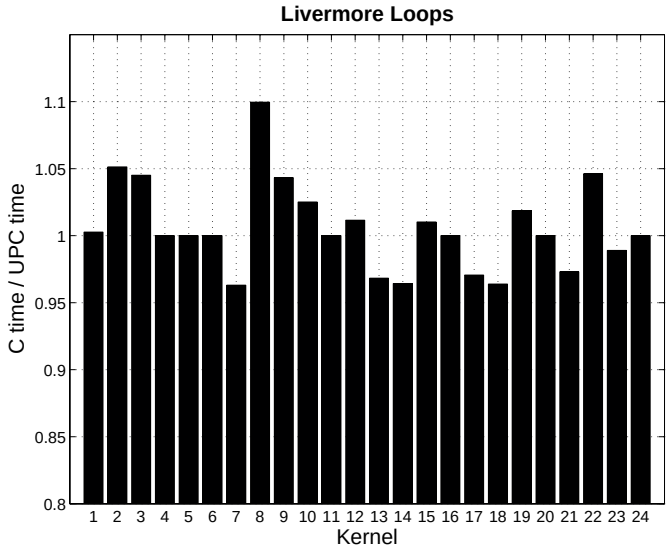

Figure 10. Performance of the individual Livermore kernels

	Geo. Mean	Avg. Rate	Har. Mean	Max	Min
C	160	756	58.7	6561	9.0
UPC	161.9	762	59.6	6652	9.0

Table 1. Aggregate performance of the Livermore Loops (in MFLOPS)

presents the aggregate performance for both the original C source and the translated output with the `-O3` flag, while Figure 10 displays the normalized performance of the individual kernels.

As Table 1 shows, Berkeley UPC's translated output performs almost identically to the original C source code. Performance results from the individual benchmarks confirm this observation; the ratio of UPC running time versus C running time is within 5% for nearly all of the kernels (and the remaining differences can be attributed to measurement noise). One notable exception occurs in kernel 8, where Berkeley UPC's output surprisingly outperforms the C code by about 10%. Examination of the translated output suggests that its performance benefits from the Berkeley UPC translator recognizing several three dimensional array accesses in the loop as common subexpressions and replacing them with stack temporaries. The introduction of the stack variables does not affect vectorization, and saves three address calculations per iteration. Because the translated output exhibits similar performance to the C code for most of the kernels, we expect the Berkeley UPC compiler to offer competitive serial performance on a vector platform like the X1. Finally, we note that Cray C has failed to vectorize a substantial number of the benchmarks, even though many of them do not contain any vector dependences. This suggests that automatic vectorization alone is not sufficient for good absolute computational performance, due to the inherent limitations of static analysis – in general, vectorization directives, code reorganizations, and algorithm changes may all be required for achieving good performance when porting applications to the X1. Furthermore, some algorithms are inherently not amenable to vectorization, and applications whose performance hinges on such algorithms are unlikely to ever perform well on the machine.

7. Potential for UPC Parallel Compiler Optimizations

In an earlier paper [7], we identified several compiler optimizations that prove valuable for implementing GAS languages such as UPC in a distributed memory environment: communication and computation overlap, prefetching of remote data, message aggregation, and privatization of local shared data. The performance characteristics of the Cray X1 that we have observed thus far, however, raise questions about the appropriateness of these optimizations for this machine, whose tightly-coupled architecture delivers impressive peak performance but also limits the opportunities for GAS language implementations to exploit alternative techniques in reducing communication overhead. In this section, we evaluate the effectiveness of two important optimization techniques on the Cray X1.

7.1 Message Coalescing and Aggregation

The widely used LogGP network performance model [1] speaks volumes about the effectiveness of message coalescing and aggregation; by combining small puts and gets into large messages, not only does one save on the per-message startup overheads, but one can also exploit the higher bandwidth offered by modern high-performance networks for large messages. The most common realization of this optimization, called *message vectorization*, significantly improves the performance of a fine-grained loop by fetching all the remote values it needs in a single bulk transfer instead of issuing fine-grained read operations in every iteration. Other similar techniques include copying an entire object when accessing its fields, and packing together messages bound for the same destination node.

Our benchmarking of the Cray X1's memory and communication performance, however, raises doubt about the relevance of converting fine-grained accesses into coarse-grained bulk transfers on this platform. If the latency and bandwidth of a remote memory access are comparable to those of a local access, it may not make sense to bulk fetch remote data into local buffers, since one still has to pay for the overhead of moving data from the main memory into cache. Furthermore, hardware support for vectorized loads can alleviate much of the communication overhead for small messages. On the other hand, if a remote shared object is to be referenced multiple times, it might be beneficial to copy the object locally (as permitted by UPC's relaxed consistency model) so that its value resides in cacheable local memory, because X1 nodes do not cache remote memory locations. Essentially, we seek to evaluate the impact of a shared memory programming paradigm for UPC application performance on the Cray X1; if the X1's transparent global loads and stores can efficiently support fine-grained accesses to remote data, programmers can enjoy both the simplicity offered by a shared memory programming style and application performance comparable to programming with coarse-grained bulk-synchronous style communication.

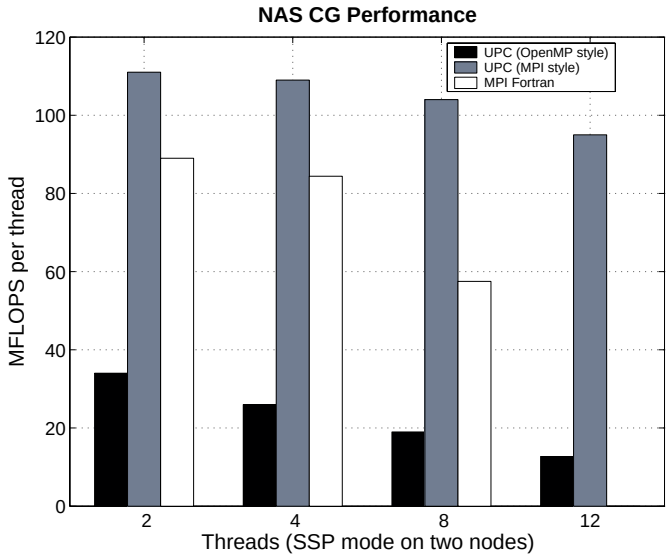

Figure 11. NAS CG (class B): fine-grained vs. coarse-grained

To answer these questions, we compared the performance of two versions of the NAS conjugate gradient (CG) benchmark from [3]. The first is derived from an OpenMP-style shared memory implementation, with the exception that the column vector is replicated to avoid repeated random indexing into it. The second version is written in the bulk-synchronous style of one-sided coarse-grained communication, through the use of `upc_memget` library calls. The sparse matrix-vector multiplication in both versions was tuned to ensure that the inner loops were vectorized. Both are compiled in SSP mode and executed such that the UPC threads are evenly distributed among the two nodes. Performance results from the MPI Fortran version of the benchmark were also included for comparison[†]. As Figure 11 shows, performance of the shared memory style version lags behind that of code with coarse-grained parallelism. Much of the performance advantage offered by the coarse-grained version can be attributed to a tighter inner loop for the matrix-vector product, as the boundary information for each thread can be precomputed due to explicit partitioning of the sparse matrix. In summary, although the Cray X1's tightly-coupled shared memory interface lowers the communication overhead, a coarse-grained communication pattern is likely to still outperform a fine-grained access pattern, even for applications with irregular and dynamic parallelism. This also suggests that UPC's hybrid programming model can be well-suited for the Cray X1; fine-grained accesses through pointers-to-shared can deliver acceptable performance if they can be vectorized, while performance critical sections of the code can be further optimized into a bulk synchronous programming style.

7.2 Communication/Computation Overlap

Compiler-controlled overlapping of communication and computation is a crucial optimization for parallel programs on conventional distributed-memory systems, as it can effectively hide communication latencies by keeping the processor busy with independent local computation while waiting for remote data to arrive. This capability is especially relevant for UPC programs; unlike other parallel programming paradigms such as MPI or Split-C [11], UPC currently offers no non-blocking communication operations at the language level and instead expects compilers to perform all such optimizations automatically. The straightforward approach to applying this transformation is to convert one-sided blocking get/put operations into an initiation call and a corresponding synchronization call, then perform code motion to separate the two as far as possible while inserting independent computation or communication code in between. Several studies [30, 17, 6] have proposed global communication scheduling techniques that attempt to find an optimal arrangement for all non-blocking memory accesses. Other variants of this optimization such as message strip mining [27] and software prefetching [18] are also useful in reducing an application's stall times due to communication latencies.

The Cray X1's choice to hide the messaging layer and instead rely on vectorization for communication performance makes it a less-than-ideal target for GAS language implementations that wish to explicitly overlap communication and computation. As Section 4.2 mentions, the Cray X1 offers only a load/store based interface for remote accesses. While limited overlapping between scalar loads may be achieved with code scheduling to exploit instruction-level parallelism, such assembly-level optimizations are generally not applicable for implementations relying on source-to-source transformation, at least not in a portable manner. The X1 ISA also provides limited software prefetching support with a scalar data prefetch instruction, but the lack of inline assembly support in the C compiler prevents portable implementations from accessing the feature. Stores are inherently non-blocking, and as mentioned in Section 5, we pipeline outstanding relaxed scalar puts and eliminate the individual synchronization calls that block for their completion, taking advantage of hardware memory ordering guarantees on scalar conflicting accesses. However, this code generation strategy hinges on the automatic vectorizor's ability to vectorize these fine-grained writes in the inner loops in order to achieve efficient communication.

This heavy reliance on vectorization to effectively utilize the high

[†]The MPI Fortran code only works for threads in powers of 2.

memory bandwidth is a major reason the Cray X1 can achieve a high percentage of the peak hardware performance, but imposes unfortunate limitations for portable parallel language compilers or libraries seeking to exercise detailed control over communication optimizations. Compiler and library developers have no direct control of an application's parallel performance, other than to apply transformations that result in the most vectorization; our experiences with UPC benchmarks suggest that vectorization directives and code modifications are generally necessary for good performance. While the amount of reengineering required for vectorization will likely decrease as Cray's compiler matures, a fundamental problem is that C makes a poor compilation target for vectorization, due to the lack of language-level vector operations and the conservatism introduced into data dependence analysis by pointer aliasing. Portable implementations for Fortran-based GAS languages such as Co-Array Fortran [8] will likely fare better on the X1, due to the relative ease of vectorizing Fortran 90 code. However, programs with distributed pointer-based data structures are unlikely to benefit from vectorization at all, whereas compiler-controlled data prefetching transformations using split-phase operations could be an effective approach.

8. Parallel Performance

The NAS Multigrid (MG) benchmark was used to evaluate our compiler's parallel performance, as the program contains a good balance of computation and communication. Running in both SSP and MSP mode, we compared two configurations: UPC compiled with Berkeley UPC versus Fortran MPI with Cray Fortran. The Cray C compiler fails to automatically vectorize the computation loops in the UPC code, and we had to explicitly insert pragmas to enable vectorization and multistreaming. As Figure 12 shows, both UPC and MPI Fortran perform well in the absolute sense, with performance in the giga-flops range. This result is expected, as both the UPC and MPI version use coarse-grained communication, and their computation code is very similar. A more interesting comparison is between the relative performance of the MSP and SSP configurations; in MSP mode the Cray compilers determine (with help from programmer annotations) how to distribute loop iterations among the four SSPs, while the SSP mode introduces more parallelism at the program level by mapping application threads to each individual SSP. The measured performance of one MSP is approximately three times of that of a single SSP while it uses four times the amount of hardware, which would seem to suggest that the SSP mode makes more efficient use of the available hardware. Performance data from executing more than four SSPs on the same node, however, contradicts this hypothesis. Regardless of the programming model used, a significant performance degradation was observed when scaling from 4 to 8 threads under SSP mode. Our investigation reveals the cause to be increased cache miss traffic in the two-way set-associative E-cache shared by the four SSPs in an MSP. The X1 currently does not grant users control of SSP placement across the MSPs under SSP mode, and the scheduler attempts to allocate application threads to all four SSPs in the same MSP. Four independent threads therefore share a two-way set-associative cache [24], and due to the SPMD model all have the same memory layout; as the four processors execute in parallel, the private objects owned by different threads map to the same cache entry due to identical offsets in the low bits of the virtual address, resulting in a significant, pathological increase in cache misses from conflict interferences for memory intensive benchmarks. The MSP mode, on the other hand, is not susceptible to this phenomenon, as there is only one process image (and hence one copy of the private objects) per MSP/E-cache.

In our performance study we next used the NAS integer sort (IS) kernel, a benchmark written in a bulk synchronous style with high communication bandwidth requirements. A UPC version of the benchmark compiled with Berkeley UPC was compared against another version written with MPI in C. Both versions were compiled with full optimizations enabled, and we do not distinguish between SSP and MSP mode, as the benchmark contains no loops that can profit from multi-streaming. As Figure 13 shows, Berkeley UPC achieves similar performance to MPI, with both scaling well, even in the presence of inter-node communication. In terms of absolute performance, however, both versions are quite inefficient, achieving only 1% of peak performance on the X1. This is due to the fact that loops in the benchmark have true recurrences and thus do not benefit from vectorization, but instead must be executed on the slower scalar processor. This supports our argument in Section 7.2 that questions the elimination of split-phase remote gets from the X1; whereas vectorization has failed to optimize the IS benchmark, split-phase operators could still be used to convert the remote bulk transfers into non-blocking operations and overlap the communication time with independent computation.

9. Analysis

Superficially, the X1 is an ideal machine for GAS languages, because the global memory operations are directly supported in hardware. However, we found several features of the X1 present challenges for our compilation approach, and we believe this experience may be useful for the designers of future GAS language compilers and system architects.

- Heavy reliance on vectorization: The performance gap between scalar and vector code is dramatic, due to the 2x factor in clock rate and 2x factor in available functional unit parallelism. Both a faster scalar processor and a more powerful vectorizing compiler would help address this issue. The use of vectorization to mask communication means that, even if the scalar processor were faster, vectorization would still be critical for communication overlap.

  Applications whose main computation loops contain true recurrences (e.g., NAS IS) run very inefficiently because they do not benefit from the computational power and memory bandwidth offered by the vector pipelines, which are essentially wasted. The C compiler's ability to automatically identify candidates for vectorization could also be further improved, as demonstrated by the Livermore loop results and the fact that we had to manually insert multiple pragma directives to achieve acceptable performance on our parallel benchmarks.

- Limited forms of communication: Cray's decision to avoid caching remote data matches UPC's affinity model quite well, and allows for a simpler hardware design and faster remote access times – however this mandates more careful attention to the data access locality pattern. The hardware provides fast communication between memory and registers, but no direct support for memory to memory operations, which are important for non-blocking bulk-synchronous communication. While the register level operations are powerful in tightly integrated communication and computation code, they consume critical register resources and provide only limited forms of synchronization. For remote loads, the synchronization is automatic when a successive operation accesses the register, but for remote stores, explicit synchronization is often needed. The X1 architecture provides only a single heavyweight synchronization mechanism (*gsync*) to wait for the completion of all outstanding remote writes, and lacks fine-grained forms of synchronization which could admit more general forms of communication pipelining. The addition of memory-to-memory operations would provide additional flexibility in code generation, especially if they were combined with flexible synchronization primitives to synchronize on sets of outstanding operations. Although memory-to-cache operations (such as prefetches) have some advantages in

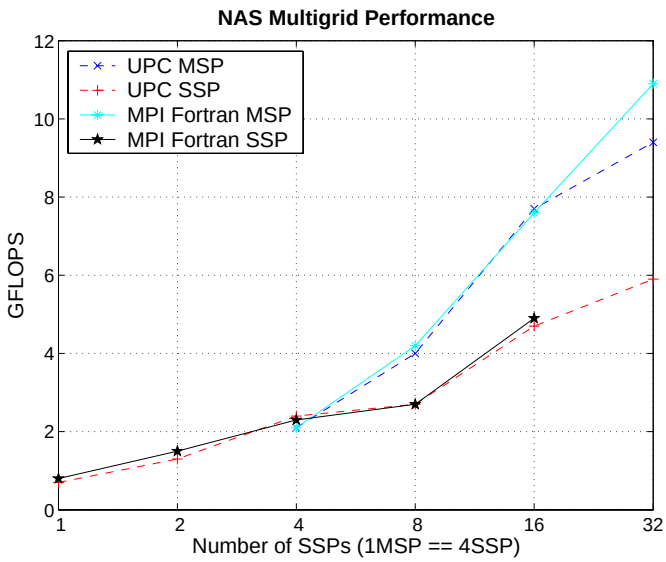

Figure 12. NAS MG (class B)

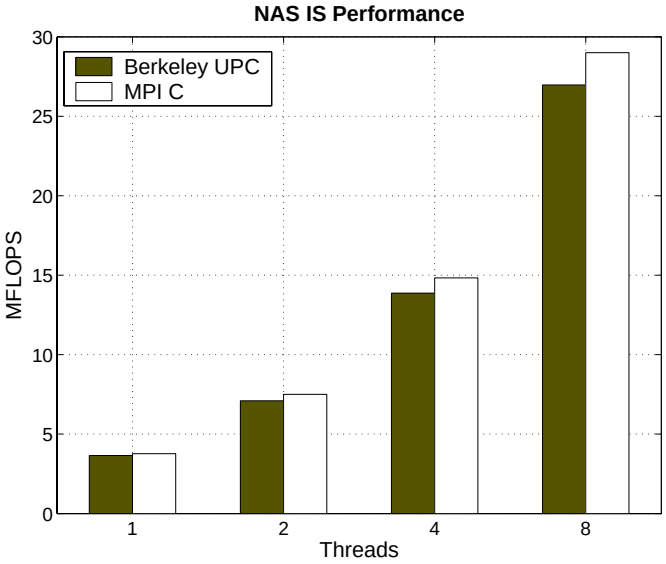

Figure 13. NAS IS (class B)

enabling prefetching optimizations, we believe that the added complexity currently required to exploit this functionality from the C source level is probably not justified.

- Cache structure: The mapping algorithm in the shared E-cache within an MSP makes it difficult to obtain high performance for SPMD programs in SSP mode, due to the high likelihood of cache conflicts between symmetric data objects associated with each SSP thread. We believe a software workaround may be possible by staggering allocations in memory, at the cost of some added complexity in pointer arithmetic. A slightly smarter cache hashing function or a 4-way set associative E-cache would have been a better design match to the 4 SSPs sharing the cache.

10. Conclusions

We have described our implementation of the Berkeley UPC compiler on the Cray X1 architecture. We showed that the Berkeley compiler performs comparably to the Cray UPC compiler, even though the Berkeley compiler supports the entire UPC language specification, while the Cray compiler omits support for some UPC language features. One of the key features currently missing from Cray UPC is support for arbitrary blocked cyclic data layouts. We have shown that static typing information can be used to specialize the generation of pointer arithmetic for the important special case of phaseless pointers, ensuring that programmers only pay for the generality of blocked-cyclic pointers-to-shared when they are actually used in the application. As a concession to portability and compiler development time, the Berkeley compiler generates C, rather than native assembly, and relies on the vendor compiler to perform most serial optimizations. Surprisingly, the generated C code from our compiler often vectorizes as well as the input code, which validates our approach.

Our benchmarks demonstrate that the X1's architectural global address space support is used most effectively by a global address space programming model that integrates communication and computation. The one-sided put/get model in *shmem* is significantly faster than the two-sided MPI interface, however the use of direct loads and stores that we leverage in our GASNet implementation is faster still. We built an implementation of our GASNet layer starting with the active message-based Core API, whose generality is exploited to provide a high-performance implementation of the more challenging features of UPC, such as the ability to non-collectively allocate remote memory. The user does not pay for this generality when it is not needed, because the remote memory access operations in the extended GASNet interface are implemented as macros that translate to direct loads and stores.

A remaining open issue is the vectorization of code that mixes communication and computation. The reliance on vectorization in the X1, not only for computational performance but also to overlap remote access times, means that vectorization of communication code is critical. For programs with fine-grained irregular accesses, the vector instruction set supports indexed loads and stores (scatter/gather), yet the Cray compiler will not vectorize loops that contain function calls (most notably including calls to *shmem*), nor does it support inline assembly instructions. This limits our ability to generate the type of mixed communication and computation code that would most effectively use the hardware. We believe this is an issue for application-level library efforts, not just our own source-to-source compilation strategy, because scientific libraries are often written with separate modules and runtime phases for communication and computation. A strategy for annotating or rewriting the separate communication and computation code to enable vectorization may perform reasonably well, but will miss the opportunity to take advantage of Cray's tight integration of the network and the processor, in which the basic communication mechanism is a transfer between local registers and remote memory. In one of these phased, bulk-synchronous programs, data will flow from remote memory to the local vector registers and then to local memory during communication, and from local memory back to registers when that data is needed during computation. Cray compiler support for inline assembly and interprocedural analyses to support the automatic vectorizer, or at least special recognition of the *shmem* calls, would all help to address this issue.

On balance, our layered approach to compiler design has proven quite effective across a wide range of architectures. GASNet has a carefully designed API which is used in generating code for Titanium as well as UPC, and we are currently extending it to include strided and scatter/gather accesses that are important for enabling various parallel compiler optimizations and supporting Co-Array Fortran. A key design point has been the use of macros and inline expansion for simple GASNet functions, such that despite API abstraction layering, accesses to remote values translate directly to loads and stores on machines that support direct remote access. Our UPC runtime layer now contains three different UPC pointer-to-shared representations, including one designed specifically to match the memory layout on the X1.

We believe that both the pointer and GASNet work will be useful on other architectures with similar memory layout and access characteristics (specifically including the SGI Altix), and that our analysis of the architectural support provided by the X1 will be useful in the design of future architectures intended to support GAS languages.

11. REFERENCES

[1] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: Incorporating long messages into the LogP model for parallel computation. *Journal of Parallel and Distributed Computing*, 44(1):71–79, 1997.

[2] The Berkeley UPC Compiler, 2002. http://upc.lbl.gov.

[3] K. Berlin, J. Huan, M. Jacob, et al. Evaluating the impact of programming language features on the performance of parallel applications on cluster architectures. In *16th International Workshop on Languages and Compilers for Parallel Processing (LCPC)*, October 2003.

[4] D. Bonachea. GASNet specification. Technical Report CSD-02-1207, University of California, Berkeley, October 2002.

[5] Programming Languages – C, 1999. The ISO C Standard, ISO/IEC 9899:1999.

[6] S. Chakrabarti, M. Gupta, and J. Choi. Global communication analysis and optimization. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 68–78, 1996.

[7] W. Chen, D. Bonachea, J. Duell, P. Husband, C. Iancu, and K. Yelick. A performance analysis of the Berkeley UPC Compiler. In *Proceedings of the 17th International Conference on Supercomputing (ICS)*, June 2003.

[8] C. Coarfa, Y. Dotsenko, J. Eckhardt, and J. Mellor-Crummey. Co-array Fortran performance and potential: An NPB experimental study. In *16th International Workshop on Languages and Compilers for Parallel Processing (LCPC)*, October 2003.

[9] Cray C/C++ reference manual. http://www.cray.com/craydoc/manuals/004-2179-003/html-004-2179-003/.

[10] Cray X1 system overview. http://www.cray.com/craydoc/20/manuals/S-2346-23/html-S-2346-23/S-2346-23-toc.html.

[11] D. Culler, A. Dusseau, S. Goldstein, A. Krishnamurthy, S. Lumetta, T. Eicken, and K. Yelick. Parallel programming in Split-C. In *Supercomputing (SC1993)*, 1993.

[12] T. Dunigan, M. Fahey, J. White, and P. Worley. Early evaluation of the Cray X1. In *Supercomputing 2003 (SC2003)*, November 2003.

[13] The Earth Simulator Center. http://www.es.jamstec.go.jp/.

[14] T. El-Ghazawi and F. Cantonnet. UPC performance and potential: A NPB experimental study. In *Supercomputing2002 (SC2002)*, November 2002.

[15] T. El-Ghazawi, W. Carlson, and J. Draper. *UPC specification*, 2003. http://upc.gwu.edu/documentation.html.

[16] P. Hilfinger et al. Titanium language reference manual. Technical Report CSD-01-1163, University of California, Berkeley, November 2001.

[17] A. Krishnamurthy and K. Yelick. Analyses and optimizations for shared address space programs. *Jorunal of Parallel and Distributed Computing*, 1996.

[18] C. Luk and T. Mowry. Compiler-based prefetching for recursive data structures. In *Architectural Support for Programming Languages and Operating Systems*, pages 222–233, 1996.

[19] S. Lumetta and D. Culler. Managing concurrent access for shared memory active messages. In *Proceedings of the International Parallel Processing Symposium*, pages 272–279, 1998.

[20] F. McMahon. The Livermore Fortran Kernels: A computer test of the numerical performance range. Technical report, Lawrence Livermore National Laboratory, December 1986.

[21] The Message Passing Interface (MPI) standard. http://www.mpi-forum.org/.

[22] R. Numwich and J. Reid. Co-Array Fortran for parallel programming. Technical Report RAL-TR-1998-060, Rutherford Appleton Laboratory, 1998.

[23] L. Oliker et al. Evaluation of cache-based superscalar and cacheless vector architectures for scientific computations. In *Supercomputing 2003 (SC2003)*, November 2003.

[24] Optimizing applications on the Cray X1 system. http://www.cray.com/craydoc/20/manuals/S-2315-51/html-S-2315-51/S-2315-51-toc.html.

[25] S. L. Scott. Synchronization and communication in the T3E multiprocessor. In *Architectural Support for Programming Languages and Operating Systems*, pages 26–36, 1996.

[26] Man page collections: Shared memory access (SHMEM). http://www.cray.com/craydoc/20/manuals/S-2383-22/S-2383-22-manual.pdf.

[27] A. Wakatani. Effectiveness of Message Strip-Mining for Regular and Irregular Communication. In *PDCS*, Oct 94.

[28] K. Yelick, D. Bonachea, and C. Wallace. A proposal for a UPC memory consistency model. Technical Report LBNL-54983, Lawrence Berkeley National Laboratory, May 2004.

[29] K. Yelick et al. Titanium: a high performance java dialect. In *proceedings of ACM 1998 Workshop on Java for High-Performance Network Computing*, February 1998.

[30] Y. Zhu and L. Hendren. Communication optimizations for parallel C programs. *Jorunal of Parallel and Distributed Computing*, 58(2):301–312, 1999.