GASP: A Performance Tool Interface for Global Address Space Languages

Adam Leko[1], Dan Bonachea[2], Hung-Hsun Su[1], Bryan Golden[1], Hans Sherburne[1], Alan D. George[1]

[1]Electrical Engineering Dept., University of Florida
[2]Computer Science Dept., University of California at Berkeley

Version 1.3-pre

1 Introduction

1.1 Scope

Due to the wide range of compilers and the lack of a standardized performance tool interface, writers of performance tools face many challenges when incorporating support for global address space languages such as Unified Parallel C (UPC), Titanium, and Co-Array Fortran (CAF). This document presents a Global Address Space Performance tool interface (GASP) that is flexible enough to be adapted into current global address space compiler and runtime infrastructures with little effort, while allowing performance analysis tools to gather much information about the performance of global address space programs.

1.2 Organization

Section 2 gives a high-level overview of the GASP interface. As GASP can be used to support many languages, the interface has been broken down into language-independent and language-specific sections. Section 3 presents the language-independent portions of the GASP interface, and Sections 4 through 8 detail the language-specific parts of the interface. Finally, the appendices present the motivations behind the creation of GASP and give a review of the versions of this document.

1.3 Definitions

- **Users** – individuals using a parallel language such as UPC
- **Developers** – individuals who write parallel software infrastructure such as UPC, CAF, or Titanium compilers
- **Tools** – performance analysis tools such as Vampir, TAU, or KOJAK
- **Tool developers** – individuals who develop performance analysis tools
- **Tool code** – code or library implementing the tool developer's portion of the GASP interface
- **Thread** – a thread of control in a GAS program, maps directly to UPC's concept of threads or CAF's concept of images

2 GASP overview

The GASP interface controls the interaction between a user's code, a performance tool, and GAS language compiler and/or runtime system. This interaction is event-based and comes in the form of callbacks to the `gasp_event_notify` function at runtime. The callbacks may come from instrumentation code placed directly in an executable, from an instrumented runtime library, or any other method; the interface only requires that `gasp_event_notify` is called at appropriate times in the manner described in the rest of this document.

The GASP interface allows tool developers to support GAS languages on all platforms and languages supporting the interface. The interface is used in the following 3 steps:

1. Users compile their GAS code using compiler wrapper scripts provided by tool developers. Users may specify which analysis they wish the tool to perform on their code through either command-line arguments, environment variables or through other tool-specific methods.
2. The compiler wrapper scripts pass appropriate flags to the compiler indicating which callbacks the tool wishes to receive. During the linking phase, the scripts link in appropriate code from the performance tool that handles the callbacks at runtime. This tool-provided code shall be written in C.
3. When a user runs their program, the tool-provided code receives callbacks at runtime and may perform some action such as storing all events in a trace file or performing basic statistical profiling.

The specifics of each step will be discussed in Section 3. The language-specific interface parts of the GASP interface will be discussed in Sections 4 to 8.

This section presents the language-dependent part of the GASP interface. A GAS implementation may exclude any system-level event defined for each language if an application cannot be instrumented for that event.

Any action resulting from a violation of this specification shall result in undefined behavior. Tool and language implementors are strongly encouraged not to deviate from these specifications.

3 Language-independent interface

3.1 Instrumentation control

Instrumentation control is accomplished through either compilation arguments or compiler pragmas. Developers may use alternative names for the command-line arguments if the names specified below do not fit the conventions already used by the compiler.

3.1.1 User-visible instrumentation control

If a user wishes to instrument their code for use with a tool using the GASP interface, they shall pass either the `--profile` or `--profile-local` command-line arguments to the compiler wrapper scripts.

The `--profile` argument specifies that the user's code shall be instrumented for all events supported by the GAS language implementation, except for events resulting from access to objects or variables contained in the portion of the global address space local to each thread.

For languages that do not have any concept of local or remote memory accesses, this argument shall have the same semantics as the `--profile-local` argument, which specifies that the user's code shall be instrumented for all events support by the GAS language implementation.

3.1.2 Tool-visible instrumentation control

Compilers supporting the GASP interface shall provide three command-line arguments which may be used by the tool-provided compiler wrapper scripts.

The first two arguments `--profile` and `--profile-local` have the same semantics as the user-visible instrumentation flags specified in Section 3.1.1.

The third argument `--profile-only` takes a single argument `filename` which is a file containing a list of symbolic event names (as defined in Sections 4 to 8) separated by newlines. The file's contents indicate the events for which the performance tool wishes to receive callbacks. Events in this file may be ignored by the compiler if the events are not supported by the GAS language implementation.

3.2 Callback structure

During runtime, an instrumented executable shall call the `gasp_init` C function at the beginning of program execution after the language runtime has finished initialization but before executing the entry point in a user's code (e.g., `main` in UPC). The `gasp_init` function shall have the following signature:

```
typedef enum {
  GASP_LANG_UPC,
  GASP_LANG_TITANIUM,
  GASP_LANG_CAF,
  GASP_LANG_MPI,
  GASP_LANG_SHMEM
} gasp_lang_t;

struct _gasp_context_S;
typedef struct _gasp_context_S *gasp_context_t;

gasp_context_t gasp_init(gasp_lang_t srclanguage,
                         int *argc, char ***argv);
```

The `gasp_init` function and an implementation of the `_gasp_context_S` struct shall be provided by tool developers. A single running instance of an executable may call `gasp_init` one or more times if the executable contains code written in multiple languages (such as a hybrid UPC and CAF program).

The `gasp_init` function returns a pointer to a tool-implemented struct that shall be passed in for all subsequent callbacks to the tool developer's code. This pointer shall only be used with events for the language indicated by the `srclanguage` argument.

Tool code may modify the contents of the `argc` and `argv` pointers to support the processing of command-line arguments.

After the `gasp_init` function has been called by each thread of execution, the tool code shall receive all other callbacks through the two functions whose signatures are shown below:

```
typedef enum {
  GASP_START,
  GASP_END,
  GASP_ATOMIC,
} gasp_evttype_t;

void gasp_event_notify(gasp_context_t context, unsigned int evttag,
                       gasp_evttype_t evttype, const char *filename,
                       int linenum, int colnum, ...);
```

```
void gasp_event_notifyVA(gasp_context_t context, unsigned int evttag,
                         gasp_evttype_t evttype, const char *filename,
                         int linenum, int colnum, va_list varargs);
```

Both functions may be used interchangeably; the VA variant is provided as a convenience to developers.

The gasp_event_notify shall be written in C, but may make upcalls to code written in the language specified by the srclanguage argument passed to the gasp_init function on the thread that received the callback. If upcalls are used, the gasp_event_notify function shall also be re-entrant. Additionally, code that is used in upcalls shall be compiled using the same environmental specifications as the code in a user's application (e.g., gasp_event_notify shall only perform upcalls to UPC code compiled under a static threads environment when used with a UPC program compiled under the static threads environment).

For the first argument to gasp_event_notify, tool code shall receive the same pointer to a gasp_context_t that was returned from the gasp_init function. Tool developers may use this struct to store thread-local information for each thread. The gasp_event_notify function shall be thread-safe for languages that make use of pthreads or other thread libraries.

The evttag argument shall specify the event identifier as specified in Sections 4 to 8. The evttype argument shall be of type gasp_evttype_t and shall indicate whether the event evttag is a begin event, end event, or atomic event.

The filename, linenum, and colnum arguments shall indicate the source code line and column number that spawned the event evttag. GAS language implementations that do not retain column information during compilation may pass 0 in place of the colnum parameter.

The contents of the varargs argument shall be specific to each event identifier and type and will be discussed in Sections 4 to 8.

3.3 Measurement control

Tool developers shall provide an implementation for the following functions:

```
int gasp_control(gasp_context_t context, int on);

int gasp_control_query(gasp_context_t context);
```

All functions listed above take the context argument in the same manner as the gasp_event_notify function.

When tool code receives this function call with the value 0 for the on parameter, the tool shall not measure any performance data until the tool code receives another gasp_control call with a nonzero value for the on parameter.

The gasp_control_query shall return a zero value if measurement is currently disabled, or a nonzero value if measurement is enabled.

3.4 User events

Tool developers shall provide an implementation for the following function:

```
unsigned int gasp_create_event(gasp_context_t context,
                               const char *name, const char *desc);
```

The gasp_create_event shall return a tool-generated event identifier. This identifier shall be in the range from GASP_USEREVT_START to GASP_USEREVT_END, inclusive. The GASP_USEREVT_* macros shall be provided in the gasp.h header file described in Section 3.5.

Compilers shall translate the corresponding language-specific *_create_event functions listed in Sections 4 to 8 into corresponding gasp_create_event calls. The semantics of the gasp_create_event shall be the same as the corresponding *_create_event functions listed in Sections 4 to 8.

3.5 Header files

Developers shall distribute a `gasp.h` C header file with their GAS language implementations that contains the following definitions:

- Function prototypes for the `gasp_init`, `gasp_event_notify`, `gasp_control`, `gasp_control_query`, and `gasp_create_event` functions and associated typedefs, enums, and structs.
- A `GASP_VERSION` macro that shall be defined to the GASP version supported by this GAS implementation.
- Definitions for the `GASP_USEREVT_START` and `GASP_USEREVT_END` macros.
- Macro definitions that map the symbolic event names listed in Section 4 to 8 to 32-bit unsigned integers [1].

The `gasp.h` file shall be installed in a directory that is included in the GAS compiler's default search path.

4 UPC interface

4.1 Instrumentation control

Users may insert `#pragma pupc on` or `#pragma pupc off` in their code to instruct the compiler to avoid instrumenting lexically-scoped regions of a user's UPC code. These pragmas may be ignored by the compiler if the compiler cannot control instrumentation for arbitrary regions of code.

4.2 Measurement control

At runtime, users may call the following functions below to control the measurement of performance data:

```
int pupc_control(int on);
int pupc_control_query();
```

The `pupc_control` and `pupc_control_query` functions shall behave in the same manner as the `gasp_control` and `gasp_control_query` functions defined in Section 3.3.

4.3 User events

```
unsigned int pupc_create_event(const char *name, const char *desc);
void pupc_event_start(unsigned int evttag, ...);
void pupc_event_end(unsigned int evttag, ...);
void pupc_event_atomic(unsigned int evttag, ...);
```

The `pupc_create_event` function shall be translated at compile time into a corresponding `gasp_create_event` call defined in Section 3.4. The `name` argument shall be used to associate a user-specified name with the event, and the `desc` argument may contain either `NULL` or a `printf`-style format string.

The `pupc_event_start`, `pupc_event_end`, and `pupc_event_atomic` functions may be called by a user's UPC program during runtime. The `evttag` argument shall be any value returned by the `pupc_create_event` function. Users may pass in any list of values for the `...` arguments as long as the types passed in match the `printf`-style format string used in the corresponding `pupc_create_event`. A performance tool may use these values to display performance information alongside application-specific data captured during runtime to a user. A compiler

[1] This might create problems from cross-language apps such as UPC + CAF if developers reuse the same event IDs

shall translate the `pupc_event_start`, `pupc_event_end`, and `pupc_event_atomic` function calls into corresponding `gasp_event_notify` function calls during compilation.

When a compiler does not receive any `--profile` or `--profile-local` arguments, the `pupc_event_*` function calls shall be excluded from the executable.

4.4 System events

4.4.1 Exit events

Table 1 shows system events related to the end of a program's execution.

Symbolic name	**Event type**	`vararg` **arguments**
`GASP_COLLECTIVE_EXIT`	Atomic	**`int`** `status`
`GASP_NONCOLLECTIVE_EXIT`	Atomic	**`int`** `status`

Table 1: Exit events

The `GASP_COLLECTIVE_EXIT` event shall occur at the end of a program's execution on each thread when a collective exit occurs. The `GASP_NONCOLLECTIVE_EXIT` event shall occur at the end of a program's execution on a single thread when a non-collective exit occurs.

4.4.2 Synchronization events

Table 2 shows events related to synchronization constructs.

Symbolic name	**Event type**	`vararg` **arguments**
`GASP_UPC_NOTIFY`	Start	**`int`** `named`, **`int`** `expr`
`GASP_UPC_NOTIFY`	End	**`int`** `named`, **`int`** `expr`
`GASP_UPC_WAIT`	Start	**`int`** `named`, **`int`** `expr`
`GASP_UPC_WAIT`	End	**`int`** `named`, **`int`** `expr`
`GASP_UPC_BARRIER`	Start	**`int`** `named`, **`int`** `expr`
`GASP_UPC_BARRIER`	End	**`int`** `named`, **`int`** `expr`
`GASP_UPC_FENCE`	Start	(none)
`GASP_UPC_FENCE`	End	(none)

Table 2: Synchronization events

These events shall occur before and after execution of the notify, wait, barrier, and fence synchronization statements. The `named` argument to the notify, wait, and barrier start events shall be nonzero if the user has provided an integer expression for the corresponding notify, wait, and barrier statements. In this case, the `expr` variable shall be set to the result of evaluating that integer expression. If the user has not provided an integer expression for the corresponding notify, wait, or barrier statements, the `named` argument shall be zero and the value of `expr` shall be undefined.

4.4.3 Work-sharing events

Table 3 shows events related to work-sharing constructs.

Symbolic name	**Event type**	`vararg` **arguments**
`GASP_UPC_FORALL`	Start	(none)
`GASP_UPC_FORALL`	End	(none)

Table 3: Work-sharing events

These events shall occur on each thread before and after `upc_forall` constructs are executed.

4.4.4 Library-related events

Table 4 shows events related to library functions.

Symbolic name	**Event type**	`vararg` **arguments**
`GASP_MALLOC`	Start	`size_t nbytes`
`GASP_MALLOC`	End	`void* newptr`
`GASP_REALLOC`	Start	`void *ptr, size_t size`
`GASP_REALLOC`	End	`void* newptr`
`GASP_FREE`	Start	`void *ptr`
`GASP_FREE`	End	(none)
`GASP_UPC_GLOBAL_ALLOC`	Start	`size_t nblocks, size_t nbytes`
`GASP_UPC_GLOBAL_ALLOC`	End	`void** newshrd_ptr`
`GASP_UPC_ALL_ALLOC`	Start	`size_t nblocks, size_t nbytes`
`GASP_UPC_ALL_ALLOC`	End	`void** newshrd_ptr`
`GASP_UPC_ALLOC`	Start	`size_t nblocks, size_t nbytes`
`GASP_UPC_ALLOC`	End	`void** newshrd_ptr`
`GASP_UPC_FREE`	Start	`void** shrd_ptr`
`GASP_UPC_FREE`	End	(none)
`GASP_UPC_GLOBAL_LOCK_ALLOC`	Start	(none)
`GASP_UPC_GLOBAL_LOCK_ALLOC`	End	`upc_lock_t* lck`
`GASP_UPC_ALL_LOCK_ALLOC`	Start	(none)
`GASP_UPC_ALL_LOCK_ALLOC`	End	`upc_lock_t* lck`
`GASP_UPC_LOCK_FREE`	Start	`upc_lock_t* lck`
`GASP_UPC_LOCK_FREE`	End	(none)
`GASP_UPC_LOCK`	Start	`upc_lock_t* lck`
`GASP_UPC_LOCK`	End	(none)
`GASP_UPC_LOCK_ATTEMPT`	Start	`upc_lock_t* lck`
`GASP_UPC_LOCK_ATTEMPT`	End	`int result`
`GASP_UPC_UNLOCK`	Start	`upc_lock_t* lck`
`GASP_UPC_UNLOCK`	End	(none)
`GASP_UPC_MEMCPY`	Start	`void** restrict dst,` `const void** restrict src,` `size_t n`
`GASP_UPC_MEMCPY`	End	`void** restrict dst,` `const void** restrict src,` `size_t n`
`GASP_UPC_MEMGET`	Start	`void* restrict dst,` `const void** restrict src,` `size_t n`

Continued on next page

Symbolic name	**Event type**	`vararg` **arguments**
`GASP_UPC_MEMGET`	End	**`void`**`* restrict dst,` **`const void`**`** restrict src,` `size_t n`
`GASP_UPC_MEMPUT`	Start	**`void`**`** restrict dst,` **`const void`**`** restrict src,` `size_t n`
`GASP_UPC_MEMPUT`	End	**`void`**`** restrict dst,` **`const void`**`** restrict src,` `size_t n`
`GASP_UPC_MEMSET`	Start	**`void`**`** dst,` **`int`**` c,` `size_t n`
`GASP_UPC_MEMSET`	End	**`void`**`** dst,` **`int`**` c,` `size_t n`

Table 4: Library-related events

The `GASP_MALLOC`, `GASP_REALLOC`, and `GASP_FREE` stem directly from the standard C definitions of `malloc`, `realloc`, and `free`. The rest of the events stem directly from the UPC library functions defined in the UPC specification. The `vararg` arguments for each event callback mirror those defined in the UPC language specification.

The `gasp_event_notify` function shall receive all UPC `shared` pointers through an extra level of indirection (i.e., the `void**` pointers above). Tool code wishing to dereference these pointers shall cast them to appropriate UPC `shared` pointers in upcalled code.

4.4.5 Direct shared variable access events

Table 5 shows events related to direct shared variable accesses.

Symbolic name	**Event type**	`vararg` **arguments**
`GASP_UPC_GET`	Start	**`int`**` is_relaxed,` **`void`**`* dst,` **`const void`**`** src,` `size_t n`
`GASP_UPC_GET`	End	**`int`**` is_relaxed,` **`void`**`* dst,` **`const void`**`** src,` `size_t n`
`GASP_UPC_PUT`	Start	**`int`**` is_relaxed,` **`const void`**`** dst,` **`void`**`* src,` `size_t n`
`GASP_UPC_PUT`	End	**`int`**` is_relaxed,` **`const void`**`** dst,` **`void`**`* src,` `size_t n`
`GASP_UPC_CACHE_MISS`	Atomic	**`const void`**`** dst,` **`void`**`* src,` `size_t n`

Continued on next page

Symbolic name	**Event type**	`vararg` **arguments**
`GASP_UPC_CACHE_HIT`	Atomic	`const void** dst,` `void* src,` `size_t n`

Table 5: Direct shared variable access events

These events shall occur whenever shared variables are assigned to or read from using the direct syntax (not using the `upc.h` library functions). The arguments to these events mimic those of the `upc_memget` and `upc_memput` event callback arguments, but differ from the ones presented in the previous section because they only arise from accessing shared variables directly. If the memory access occurs under the strict memory model, the `is_strict` parameter shall be nonzero; otherwise the `is_strict` parameter shall be zero.

The `GASP_UPC_CACHE_*` events may be sent for UPC runtime systems containing a software cache after a corresponding get or put start event but before a corresponding get or put end event. UPC runtimes using write-through cache systems may send `GASP_UPC_CACHE_MISS` events for each corresponding put event.

The `gasp_event_notify` function shall receive all UPC `shared` pointers through an extra level of indirection (i.e., the `void**` pointers above). Tool code wishing to dereference these pointers shall cast them to appropriate UPC `shared` pointers in upcalled code.

4.4.6 Nonblocking shared variable access events

Table 6 shows events related to direct shared variable accesses implemented through nonblocking communication.

Symbolic name	**Event type**	`vararg` **arguments**
`GASP_UPC_NB_GET_INIT`	Start	`int is_relaxed,` `const void** dst,` `void* src,` `size_t n,` `gasp_nb_handle_t handle`
`GASP_UPC_NB_GET_INIT`	End	`gasp_nb_handle_t handle`
`GASP_UPC_NB_GET_DATA`	Start	`gasp_nb_handle_t handle`
`GASP_UPC_NB_GET_DATA`	End	`gasp_nb_handle_t handle`
`GASP_UPC_NB_PUT_INIT`	Start	`int is_relaxed,` `const void** dst,` `void* src,` `size_t n,` `gasp_nb_handle_t handle`
`GASP_UPC_NB_PUT_INIT`	End	`gasp_nb_handle_t handle`
`GASP_UPC_NB_PUT_DATA`	Start	`gasp_nb_handle_t handle`
`GASP_UPC_NB_PUT_DATA`	End	`gasp_nb_handle_t handle`
`GASP_UPC_NB_SYNC`	Start	`gasp_nb_handle_t handle`
`GASP_UPC_NB_SYNC`	End	`gasp_nb_handle_t handle`

Table 6: Nonblocking shared variable access events

These nonblocking direct shared variable access events are similar to the regular direct shared variable access events in Section 4.4.5. The `*INIT` events shall correspond to the nonblocking communication initiation, the `*DATA` events shall correspond to when the data completely arrives on the destination node (these events may be excluded for most implementations that use hardware-supported DMA), and the `GASP_UPC_NB_SYNC` function shall correspond to the final synchronization call that blocks until the corresponding data of the nonblocking operation is no longer in flight.

The gasp_nb_handle_t shall be an opaque type to tool developers defined by a UPC implementation. Several outstanding nonblocking get or put operations may be attached to a single gasp_nb_handle_t instance. When a sync callback is received, the tool code shall assume all get and put operations for the corresponding handle in the sync callback have been retired.

If the tool receives a value of handle equal to GASP_NB_TRIVIAL, the tool shall consider the access a local access that required no communication and shall not receive any *DATA or *SYNC event callbacks.

The gasp_event_notify function shall receive all UPC shared pointers through an extra level of indirection (i.e., the void** pointers above). Tool code wishing to dereference these pointers shall cast them to appropriate UPC shared pointers in upcalled code.

4.4.7 Collective communication events

Table 7 shows events related to collective communication.

Symbolic name	Event type	vararg arguments
GASP_UPC_ALL_BROADCAST	Start	**void**** restrict dst, **const void**** restrict src, size_t nbytes, upc_flag_t flags
GASP_UPC_ALL_BROADCAST	End	**void**** restrict dst, **const void**** restrict src, size_t nbytes, upc_flag_t flags
GASP_UPC_ALL_SCATTER	Start	**void**** restrict dst, **const void**** restrict src, size_t nbytes, upc_flag_t flags
GASP_UPC_ALL_SCATTER	End	**void**** restrict dst, **const void**** restrict src, size_t nbytes, upc_flag_t flags
GASP_UPC_ALL_GATHER	Start	**void**** restrict dst, **const void**** restrict src, size_t nbytes, upc_flag_t flags
GASP_UPC_ALL_GATHER	End	**void**** restrict dst, **const void**** restrict src, size_t nbytes, upc_flag_t flags
GASP_UPC_ALL_EXCHANGE	Start	**void**** restrict dst, **const void**** restrict src, size_t nbytes, upc_flag_t flags
GASP_UPC_ALL_EXCHANGE	End	**void**** restrict dst, **const void**** restrict src, size_t nbytes, upc_flag_t flags

Continued on next page

Symbolic name	**Event type**	`vararg` **arguments**

`GASP_UPC_ALL_PERMUTE`	Start	`void** restrict dst,`
		`const void** restrict src,`
		`const int** restrict perm,`
		`size_t nbytes,`
		`upc_flag_t flags`

`GASP_UPC_ALL_PERMUTE`	End	`void** restrict dst,`
		`const void** restrict src,`
		`const int** restrict perm,`
		`size_t nbytes,`
		`upc_flag_t flags`

`GASP_UPC_ALL_REDUCE`	Start	`void** restrict dst,`
		`const void** restrict src,`
		`upc_op_t op,`
		`size_t nelems,`
		`size_t blk_size,`
		`upc_flag_t flags,`
		`const char* TYPE`

`GASP_UPC_ALL_REDUCE`	End	`void** restrict dst,`
		`const void** restrict src,`
		`upc_op_t op,`
		`size_t nelems,`
		`size_t blk_size,`
		`upc_flag_t flags,`
		`const char* TYPE`

`GASP_UPC_ALL_PREFIX_REDUCE`	Start	`void** restrict dst,`
		`const void** restrict src,`
		`upc_op_t op,`
		`size_t nelems,`
		`size_t blk_size,`
		`upc_flag_t flags,`
		`const char* TYPE`

`GASP_UPC_ALL_PREFIX_REDUCE`	End	`void** restrict dst,`
		`const void** restrict src,`
		`upc_op_t op,`
		`size_t nelems,`
		`size_t blk_size,`
		`upc_flag_t flags,`
		`const char* TYPE`

Table 7: Collective communication events

The events in Table 7 stem directly from the UPC collective library functions defined in the UPC specification. The `vararg` arguments for each event callback mirror those defined in the UPC language specification. For the reduction functions, a character string (such as ”unsigned int”) shall represent the data type used in the reduction.

The `gasp_event_notify` function shall receive all UPC `shared` pointers through an extra level of indirection (i.e., the `void**` pointers above). Tool code wishing to dereference these pointers shall cast them to appropriate UPC `shared` pointers in upcalled code.

4.5 Header files

UPC compilers shall distribute a `pupc.h` C header file with their GAS language implementations that contains function prototypes for the function defined in Sections 4.2 and 4.3. The `pupc.h` file shall be installed in a directory that is included in the UPC compiler’s default search path.

All supported system events shall be defined in the `gasp.h` file. Compilers may add compiler-specific events to this file.

5 Titanium interface

TBD

6 CAF interface

TBD

7 MPI interface

TBD

8 SHMEM interface

TBD

A Motivation for GASP

Global address space (GAS) languages such as Titanium [1], Unified Parallel C (UPC) [2], and Co-Array Fortran (CAF) [3] offer parallel programmers several advantages over languages that require programmers to manually specify communication between nodes. The global address space provides a convenient environment similar to threaded programming on serial machines, but comes at the cost of increased complexity in GAS compiler and runtime systems. This gives parallel programmers a much-needed increase in productivity; however, since GAS compilers handle low-level communication and work distribution, it can be difficult or impossible for programmers to determine how a given program will perform at runtime.

Recent research has indicated that performance tuning is critical for achieving optimal performance in GAS languages such as UPC, especially on cluster architectures [4]. Although recent work has produced several techniques to improve the performance of compilers by taking advantage of specific architectures [5], employing TLB-like lookup tables for remote pointers [6], and applying optimizations like message aggregation [7], the programmer's ability to exploit locality remains the most influential factor on overall GAS program performance.

The importance of performance analysis for GAS programs has also been aggravated by the lack of performance analysis tools supporting GAS languages. The relative newness of GAS languages compared with other programming models such as MPI is partly responsible for the lack of tool support, but tool developers face a major roadblock even if they wish to add GAS support in their tools: there is no standard performance tool interface that can be used to portably gather performance information from GAS programs at runtime. The extensive and almost exclusive use of the MPI profiling interface [8] by MPI performance tools illustrates the usefulness of a common performance tool interface.

To rectify this situation, we developed GASP (Global Address Space Performance tool interface), a performance tool interface for GAS languages. In a nutshell, we are trying to help programmers answer the question "Why does my GAS program have bad performance?" by providing tool developers with a consistent interface so that their performance tools can help users identify and fix performance bottlenecks. We have incorporated many ideas used for the POMP OpenMP profiling interface [9] in our proposal.

A.1 Current challenges for tool developers

MPI and other library-based parallel language extensions generally use a technique called interposition, which make developing "wrapper" libraries very straightforward. In the interposition technique, all library functions are available under two different names, usually through weak bindings. When a tool developer wants to gather information about how much time is spent in a particular library call, they create their own version of the library that invokes the original function by using its alternate name and record how much time is taken for that function call. For example, Figure 1 illustrates a typical wrapper that can be used to record information about calls to `MPI_Send` for MPI programs. MPI tool developers can rely on the existence of a `PMPI_Send` binding in every MPI implementation which has the same functionality as the original `MPI_Send` function. Tool developers may place any code they wish in their wrapper library, which allows them to support both tracing and profiling. In addition, users of MPI tools need only to relink against a profiled MPI library in order to collect information about the behavior of MPI functions in their code.

Tool developers wishing to support GAS languages are not as fortunate. Since the GAS programming model does not merely use library functions, the wrapper approach mentioned above does not present a valid strategy for recording the behavior of GAS programs. Even for GAS language implementations that encapsulate much behavior in library functions (such as Berkeley UPC [10] with GASNET [11]), no standardized form of weak binding or interposition support is currently available in these libraries that could be used to write wrappers. In addition, GAS compilers can use vastly different techniques to translate source code into executables, ranging from source-to-source transformations to more direct compilation techniques. Tool developers are left with two options for collecting information about a GAS program: binary instrumentation and source instrumentation.

At first glance, binary instrumentation seems like a feasible instrumentation strategy. Libraries like DynInst [12] make it relatively easy to instrument a user's program at runtime; since the user's source code needs no modification this reduces the overhead imposed upon users. However, in order to insert instrumentation code, binary instrumentation requires tool developers to explicitly state *exactly* where they wish to insert their code in an executable. Since existing

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm)
{
  int val;
  Record_message(comm, dest, count, buf);
  Timer_start("MPI_Send");
  val = PMPI_Send(buf, count, datatype, dest, tag, comm);
  Timer_end("MPI_Send");
  return val;
}
```

Figure 1: Example wrapper function

GAS compilers use both source-to-source transformations and direct compilation, binary instrumentation will have to be added on a compiler-by-compiler basis after an extensive analysis of how that particular compiler translates source code into executable code. This presents a severe maintenance problem for tool developers: what happens when developers change their implementation internals between versions?

In short, tool developers adopting a binary instrumentation strategy will have to spend a large portion of their time keeping up with the internal workings of each compiler they wish to support. In addition, binary instrumentation makes it very difficult to relate performance information back to actual lines in source code in a platform-independent way. Another drawback to binary instrumentation is the possibility that choosing instrumentation points in binaries may be impossible without proprietary information from developers of commercial compilers. Finally, while DynInst provides great functionality for tool developers wishing to use binary instrumentation, there are no plans to port it to Cray architectures that are not based on Linux, such as the Cray X1. The X1 and X1E provide excellent vehicles for running UPC code [5], so excluding tool support for these architectures would be disappointing.

Source instrumentation affords the most flexibility to the tool developer. With source instrumentation, a tool developer can record any data they wish, provided that the information can be extrapolated from a user's code. However, with the increasing complexity of GAS runtime systems and the use of techniques such as remote reference caching, using source instrumentation alone limits a tool developer's ability to find out when certain events *actually* occur instead of when the tool developer *thinks* they might occur. For example, UPC's relaxed memory model blurs the distinction between the sequence of actions specified in the UPC source code and what actually happens at runtime.

Source instrumentation also places a higher technical burden on the tool writer. Since most GAS languages allow users to treat remote variables in the exact manner as built-in variables, source code instrumentation systems must be able to perform a *full* parse of a user's source code so that remote variable accesses can be differentiated from local variable accesses. Even parsing straight ANSI C, a subset of the current C language specification, becomes difficult due to a few grammatical ambiguities, such as the "declaration/expression" problem [13]. In addition, most "real-world" code also uses compiler-specific extensions such as those supported by the GCC and Microsoft C and Fortran compilers. Creating a parser that supports these extensions has been found to be very time-consuming and error-prone [14].

To give a concrete example of the possible difficulties in parsing a complicated expression, consider the UPC code in Figure 2. Depending on the algorithms used to translate the UPC code into machine code, several different sequences of remote read and write operations are possible. In addition, compilers which can do complicated strength reductions may be able to reduce the first line of `main` to `f = 0; c = 0;`, which drastically reduces the number of remote memory references.

Neither source nor binary instrumentation represents an optimal path for tool developers wishing to add support for GAS languages to their tools. The availability of a standard performance tool interface for GAS languages would allow tool developers to easily add support for any compiler that implements the interface.

A.2 Design goals

In order for a performance tool interface to be effective, it must meet several criteria and strike a balance between functionality for tool developers and ease of implementation for compiler developers. This section outlines a performance

```
#include "upc.h"

shared int c = 44;
shared int f = 2;

int main() {
  f = ((2*c + 3) - (c + 2) == c) ? 4 : (c = 0);
  return 0;
}
```

Figure 2: UPC code with a complicated expression

tool interface for UPC programs that tries to follow these design guidelines:

- *Flexibility* – For the interface to be useful, it must be flexible enough to support several different performance analysis methods. For example, some tools rely on capturing full traces of a program's behavior, while others calculate statistical information at runtime and display it immediately after a program finishes executing. Our performance tool interface should support these two main modes of operation, and should not overly restrict the tool developer's analysis options.
- *Ease of implementation* – It is imperative to have the full support of compiler developers for the performance tool interface. There are many existing compilers that translate code into executables using a wide variety of methods. We do not wish to alienate compiler writers by making our performance tool interface difficult to implement for a small group of unlucky compilers. In addition, many compilers are proprietary, which limits the amount of help we can give in implementing the performance tool interface. Therefore, our interface should be as implementation-neutral as possible.
- *Low overhead* – Our performance tool interface should not drastically affect overall runtime of profiled programs. Performance data collected for programs that do not exhibit similar behavior as their unprofiled counterparts is not very useful to users. Due to the fine-grained nature of most GAS programs, we do expect some perturbation of overall execution time for profiled programs. However, at every opportunity we wish to engineer solutions that minimize the effect instrumentation has on a user's program.
- *Usefulness* – Performance tools need enough information to analyze so that they can present the user with potential problem areas in their application codes. Specifically, we feel it is absolutely necessary to incorporate source code correlation for data reported to the user down to the source line level.

B Revision history

- *Initial revision* – Defined a simple profiling interface for UPC only. Instead of one callback function, several were used for each `upc_*` function. Also suggested use of statically-allocated source location structs, variable type querying, and a function similar to `MPI_Wtime()`.
- *Version 1.1* – Changed name to "performance tool interface." Still specific to UPC. Added distinction between named and unnamed barriers. Included more detailed description of event notification structure. Added user-specified event reporting, more detailed recommendations for instrumentation control, a function for users to instruct tools to record data for specific parts of their code, and timer-related functions for tool developers (based on the Berkeley UPC high-performance timers). Added thread-safe and re-entrancy requirement for tool code. Included user function entry and exit system events.
- *Version 1.2* – No written document other than a presentation given during the UPC workshop. Changed "global timer" functions to be locally-consistent only. Included a event type argument for user and system event

callbacks (start, end, atomic). User event reporting function also takes printf-style arguments. Changed instrumentation control to `#pragma pupc on/off`, `--profile` and `--profile-local` arguments, and `pupc_control`.

- *Version 1.3* – Added support for CAF and Titanium. Generalized interface for GAS languages, changed name to GASP. Converted document to more spec-like format. Several changes to UPC system event arguments.

References

[1] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken, “Titanium: A high-performance Java dialect,” in *ACM 1998 Workshop on Java for High-Performance Network Computing* (ACM, ed.), (New York, NY 10036, USA), ACM Press, 1998.

[2] T. A. El-Ghazawi, W. W. Carlson, and J. M. Draper, “UPC language specification (v 1.2),” June 2005.

[3] B. Numrich and J. Reid, “Co-Array Fortran for parallel programming,” *ACM Fortran Forum*, vol. 17, no. 2, pp. 1–31, 1998.

[4] C. Coarfa, Y. Dotsenko, J. Mellor-Crummey, F. Cantonnet, T. El-Ghazawi, A. Mohanti, Y. Yao, and D. Chavarră-Miranda, “An evaluation of global address space languages: Co-Array Fortran and Unified Parallel C,” in *PPoPP ’05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, (New York, NY, USA), pp. 36–47, ACM Press, 2005.

[5] C. Bell, W.-Y. Chen, D. Bonachea, and K. Yelick, “Evaluating support for global address space languages on the Cray X1,” in *ICS ’04: Proceedings of the 18th annual international conference on Supercomputing*, (New York, NY, USA), pp. 184–195, ACM Press, 2004.

[6] F. Cantonnet, T. A. El-Ghazawi, P. Lorenz, and J. Gaber, “Fast address translation techniques for distributed shared memory compilers.,” in *IPDPS*, IEEE Computer Society, 2005.

[7] W.-Y. Chen, “Building a Source-to-Source UPC-to-C Translator,” Master’s thesis, University of California at Berkeley, 2005.

[8] M. P. I. Forum, “MPI: A message-passing interface standard,” tech. rep., University of Tennessee, Knoxville, TN, USA, 1994.

[9] B. Mohr, A. D. Malony, S. Shende, and F. Wolf, “Design and prototype of a performance tool interface for OpenMP,” *J. Supercomput.*, vol. 23, no. 1, pp. 105–128, 2002.

[10] P. Husbands, C. Iancu, and K. Yelick, “A performance analysis of the Berkeley UPC compiler,” in *ICS ’03: Proceedings of the 17th annual international conference on Supercomputing*, (New York, NY, USA), pp. 63–73, ACM Press, 2003.

[11] D. Bonachea, “GASNet specification, v1.1,” tech. rep., University of California at Berkeley, Berkeley, CA, USA, 2002.

[12] B. Buck and J. K. Hollingsworth, “An API for runtime code patching,” *The International Journal of High Performance Computing Applications*, vol. 14, pp. 317–329, Winter 2000.

[13] B. A. Malloy and J. F. Power, “Program annotation in XML: A parser-based approach,” in *WCRE 2002, Working Conference on Reverse Engineering*, pp. 190–198, October 28 - November 1 2002.

[14] G. C. Necula, S. McPeak, S. Rahul, and W. Weimer, “CIL: Intermediate language and tools for analysis and transformation of C programs,” in *Conference on Compiler Construction*, 2002.