TARMAC: A LANGUAGE SYSTEM SUBSTRATE BASED ON MOBILE MEMORY

Steven E. Lucco
David P. Anderson

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, CA 94720

November 1, 1989

ABSTRACT

Tarmac is a *language system substrate* on which systems for distributed parallel programming can be built. Tarmac provides a model of shared global state called *mobile memory*. The basic unit of state in this model can be viewed as both 1) a block of memory that can be directly accessed by machine instructions, and 2) a logical entity with a globally unique name that may be efficiently located, copied and moved. To support higher-level synchronization models, the movement of a memory unit may optionally enable computations.

Mobile memory is more flexible than models such as distributed virtual memory, shared tuple space, or distributed objects. It avoids the limitations of fixed page size, fixed data placement policy, and type-system or language dependence. This flexibility allows Tarmac to support a wide range of parallel programming models efficiently

1. INTRODUCTION

We are concerned in this paper with distributed parallel programming on a network of computers (uniprocessors, multiprocessors, or both). A variety of possible models exist for this type of programming, each with its own advantages: functional, object-oriented, and dataflow languages, parallelizing compilers for sequential languages such as FORTRAN, parallel database systems, and so on. Our main goal was to identify the functionality common to these models, and implement it in a single system layer, thereby facilitating the implementation (and interaction) of different models. While these models differ in many important ways, they all provide some form of "state" that is shared and communicated among the parts of the computation. The resulting system, Tarmac, is a toolkit for building systems that incorporate shared state. Tarmac is a *language system substrate*: a layer interposed between language systems and operating systems (see Figure 1).

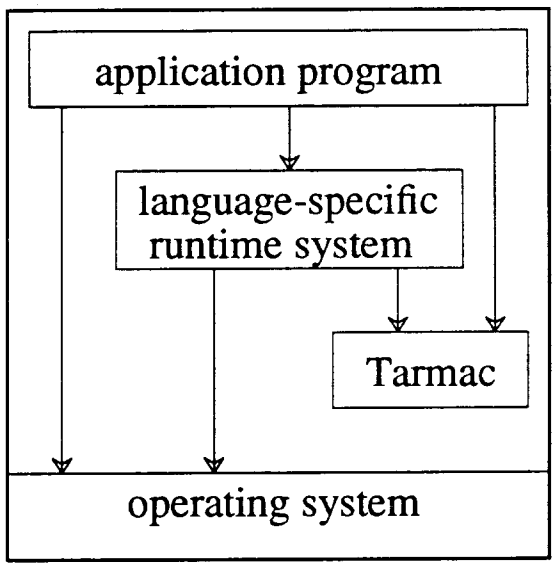

Figure 1: Tarmac is a *language system substrate*
interposed between programming language systems and operating systems.
This figure shows the dependencies among the runtime software components.

Examples of systems for distributed parallel programming include Amber [5], Linda [8], and Ivy [12]. These systems represent different models, but they have two important and desirable properties in common: 1) global shared state is encapsulated and explicitly managed by the underlying system (language, runtime system, and operating system); 2) the system handles the movement of parts of the shared state between computational nodes. These properties eliminate the need for the programmer to design and program message-passing protocols for maintaining and replicating shared state.

The above systems, however, have properties that limit their range of applicability. Shared virtual memory systems such as Ivy have page granularity. Entire pages must be moved between nodes even if only single bytes are referenced, and some global access patterns may cause thrashing. Implementations of the Linda system have embedded policies for tuple transmission and placement. Most object-oriented systems are tied to a single programming language or type system, and cannot easily be used from other languages.

The goal of Tarmac is to provide a facility for managing shared state with the beneficial features of existing systems (encapsulation of small and large state units, mobility, global reference and location, and replication), but without language dependence and system-enforced data placement policies. One of Tarmac's central contributions is that it supports a dual view of state units. Clients can view state both as "black boxes" that can be named and moved, and as virtual memory segments that can be manipulated by statements in arbitrary programming languages.

With this goal in mind, we designed a model of shared state that we call *mobile memory*. In this model, clients can create, access, move and copy arbitrary-size *memory units*. Tarmac provides only mobile memory; any notion of "type", as well as policies for data location, movement, and concurrent access, are left up to the client. Tarmac's level of abstraction is high enough to hide details of communication and allocation (thus simplifying language system development) yet low enough so that few restrictions are made on these systems.

The remainder of this paper is organized as follows: Section 2 describes the Tarmac model, and Section 3 gives examples of its use. Section 4 describes a prototype implementation of Tarmac and gives some performance measurements. Section 5 discusses related work, and Section 6 gives conclusions.

2. Mobile Memory

Tarmac is designed around a *mobile memory* abstraction. Mobile memory is a model of the communication and storage resources available in a distributed computing network. The model involves the following types of entities:

> *memory unit* (MU): a region of memory.
> *habitat*: a set of memory units (generally an address space)
> *label*: a tag, attached to MUs, with client-determined interpretation

Tarmac uses unique identifiers (UIDs) to identify all of these entities. UIDs have an immutable part, used to establish object identity, and a location hint part. UIDs are always passed by reference to Tarmac, so that Tarmac can update their hint part whenever they are used.

A *memory unit* (MU) is a region of memory with a definite size. MUs can contain code or data; there is no Tarmac-defined notion of type. Tarmac provides the following interface for creating MUs:

```
UID-list = create_memory_units(number_of_units, size);
make_immutable(UID);
```

2.1. Labels

A *label* is a client-defined tag for MUs.

```
label_UID = create_label();
bind_label(label_UID, UID);
```

One can associate zero or more labels with a particular MU. Applications can use labels to represent MU types or other information. Copies of the same application on separate hosts must

agree, by some higher level mechanism, on the interpretation of label UIDs. The location hint part of a label's UID refers to the location of an arbitrary MU that has been bound to that label, so that clients can access MUs associatively through labels.

Habitats represent the physical location of a MU. A habitat may be a virtual address space on a particular host, or a file system (only address space habitats are supported in the current design). Processes running in a habitat can also interact with the local operating system to create non-shared data structures. Every MU has a single *current location* which is a habitat. MUs can be moved or copied from one habitat to another, using

```
move(UID, target_UID);
copy(UID, target_UID);
virtual_move(UID, target_UID);
virtual_copy(UID, target_UID);
```

An MU *A* can be moved to any other MU *B*, after which *A's* current location becomes *B's* habitat. If the MU has been designated *immutable*, Tarmac may maintain a copy of the MU in both the source and destination habitats, for more efficient access. *Copy* creates a new MU and then behaves like move. UIDs act as capabilities to MUs; any MU *A* which knows the UID for an MU *B* can specify *B* as a target for a move.

MUs located in the same habitat can reference each other directly, using memory addresses. The memory address of an MU is made available when it is first moved to a habitat (see Section 2.3). If an MU containing direct references to other MUs is moved, it is the responsibility of the language system to patch up its references.

Tarmac supports an *alignment* facility, similar in intent to Emerald's notion of attached objects [10]. If *A* and *B* are aligned, and *A* moves to a new habitat, Tarmac will move *B* to join *A*. Furthermore, the relative addresses of *A* and *B* will remain the same. Tarmac tries to place aligned MUs in the same physical page, increasing the efficiency of moves involving large numbers of small, aligned MUs. The interface for alignment specification is

```
align(UID1, UID2);
```

As an example of the use of alignment, if a user-level computation creates a binary tree in which all nodes are aligned with their parent, a move of any node will cause the subtree rooted at that node to move.

2.2. Lazy Information Transfer

Suppose that one is writing a compiler that will execute in parallel. The compiler breaks a program into separate procedures and has a different processor compile each procedure. However, for convenience, the compiler would like to use only a single symbol table data structure. Because all processors will refer to the symbol table, it should be efficiently accessible from anywhere in the network. To achieve this, the programmer can make a copy of the symbol table on each processor participating in the computation. However, this will result in unnecessary message traffic on the network as each processor will only modify a small, disjoint part of the table. Tarmac provides facilities to transfer only the portions of a data structure actually used, and to re-merge modifications to the data structure from multiple locations. Language systems can use this facility to implement a wide variety of data replication policies.

A *virtual move* of MU *A* to habitat *H* creates a new MU *A** in *H*, but does not actually transfer any state from *A*. *A* is now called the *backing MU* for *A**. When *A** tries to access a location within its state, Tarmac moves that part of the state from *A* to *A**. A *virtual move* of *A** back to *A* causes the modified parts of *A** to be written to *A*. Tarmac may also periodically write modified parts of *A** to *A* if it has no more space in *A*'s* habitat or upon explicit request.

A program can *virtual move* an MU to more than one habitat. In the symbol table example, the compiler could virtual move the symbol table to all habitats participating in the computation. Each processor could act independently. As processors finish their portion of the computation, they *virtual move* their symbol table MU to the backing MU, causing Tarmac to merge their changes into the state held by the backing MU. Finally, a program can move a backing MU to a

new habitat with no effect on the backing MU's semantics. However, if the program moves a backing MU *MB* to an MU *MVM* created by a *virtual move* on *MB*, then *MVM's* changes will be merged with *MB*.

A *virtual copy* on MU *A* creates a new MU *B*. None of *A's* state is transferred to *B* until *B* is referenced or *A* is written (copy-on-write).

A FORTRAN compiler that detects parallelism might generate code that uses these Tarmac primitives to efficiently synchronize data sharing. Suppose that the compiler has two processes, *P1* and *P2*, execute code that modify some common data. Further assume (as is often the case in practice) that the compiler can partition the common data between *P1* and *P2* such that both processes are mostly accessing local storage. Such a compiler can identify critical sections involving the modification of shared data structures. For example, a compiler could detect situations in which *P2* must modify data held in *P1's* habitat. It could then preface each critical section of this type with a *virtual move* to transfer the relevant MUs from *P1's* habitat to *P2's* habitat. Using motion events (see Section 2.3), *P1* would block until the MUs returned. Only those pages of the MUs modified by *P2* would ever be transferred across the network. Further, the compiler could recognize data structures that become read-only after some initial computation phase, and increase efficiency by designating these data structures immutable.

Primitives such as *virtual move* and *virtual copy* are not supported by current object-oriented distributed systems (some operating systems, *e.g.* Tenex [4] use copy-on-write for other purposes). These primitives significantly extend the usefulness of object motion mechanisms; they support such requirements as temporary motion with mutual exclusion and partial replication of large objects. Clients can use these facilities to implement higher level virtual memory models, or to simplify the construction of mechanisms such as efficient call-by-reference in an RPC system. In addition, *virtual move* helps when a processor with few storage resources must refer to a large amount of data, as when a diskless workstation must swap virtual memory pages

across the network. Finally, Tarmac clients can specify the size granularity (in pages) at which sections of an MU will be transferred, to help minimize the total number of messages sent. If a client does not specify a granularity, Tarmac heuristically chooses one (based on page size on the host and network data packet size).

2.3. Events

MU motions can trigger computations in the source or destination habitats. For example, an MU move could correspond to an RPC request. Tarmac clients may indicate which MU moves are to trigger computations by registering *events*.

```
event_UID = add_event(target_UID, label_UID, ...);
remove_event(target_UID, label_UID, ...);
```

An event is an ordered tuple of one or more UIDs. The first field in the tuple names a move target T. The remaining fields are labels. For an MU move to generate an event E, the move target T must match the first field in E and each of the remaining fields of E must be a member of the moved MU's set of label UIDs. The first field of an event can be the special label '*', which matches any move target.

Events are transferred to the Tarmac client either through a queue in the habitat or through a software interrupt. In either case, an event descriptor is passed to the client. Event descriptors contain the event_UID of the event, the virtual address of the MU in the source and destination habitats, and the UIDs of both habitats.

Language systems can base decisions to suspend and resume processes on The event mechanism is sufficient to support any currently popular technique for synchronization.

For example, object-oriented systems can use events and MU motion to implement the invocation of operations on objects. A simple system would define an *operation* object type as an appropriately labeled MU. Moving an operation object O to an MU T would denote invocation of the operation represented by O on the object represented by T. The state of an operation object

could include information such as the type of the target object and an operation number for that type. The language's runtime system would define an event <*, *operation*>, where *operation* is an agreed upon label for MUs representing operations. Occurrences of this event could then trigger the local invocation of the represented operation (plain RPC would work similarly). We plan to implement a version of the object-oriented language Sloop [13] using this general strategy.

3. IMPLEMENTATION

The Tarmac implementation is divided between a *Tarmac server* (one per host) and a runtime library present in each habitat. The server can reside in the operating system kernel or run as a user process. To implement *virtual move* and *virtual copy* the server requires notification whenever a page fault occurs in a local habitat.

Habitats are generally implemented as virtual address spaces loaded by a local mechanism. Through the runtime library, the application registers with its local Tarmac server and can begin interacting with the global mobile memory network. For efficiency, the runtime library handles MU allocation without contacting the Tarmac server. Each copy of the library maintains the mapping between UIDs and virtual addresses for its habitat. Library code can request multiple UIDs from the Tarmac server, so that it does not have to make such a request for every MU allocated.

The Tarmac library matches incoming MUs against a set of currently active event types. The library uses either a software interrupt or a special queue to transmit events to processes running in the habitat.

Tarmac maintains information that allows it to recognize when multiple aligned MUs exist in a contiguous region of memory (part or all of a physical page). When it recognizes this situation, it can transfer the region directly to the destination host. Otherwise it packs the aligned objects into a new page and sends that page. The receiving server always tries to allocate

contiguous memory for the aligned MUs.

Tarmac assumes that given a UID, a language system (or Tarmac itself) may need to locate an MU's host from among thousands of hosts on a network. When an MU *M* moves from habitat *H1* to habitat *H2*, Tarmac holds a forwarding address for *M* on *H1's* host. When the system attempts to move another MU to *M* at its old location, the kernel forwards the move to *M's* new location, and updates the host initiating the errant move. The details of the forwarding algorithm, such as when to update backward hosts along a chain of forwarding addresses, are essentially identical to those of the Sloop [13] forwarding algorithm.

3.1. Performance

We have implemented a prototype Tarmac system on top of 4.3 BSD Unix. The system implements all primitives described above but must simulate external paging by using odd addresses. This enables us to take measurements of the system's performance, but complicates the use of virtual movement primitives in actual applications. We plan to implement a system using DASH [2], which provides external paging as well as useful facilities for network communication.

Our measurements indicate that the prototype system can provide efficient execution of all the Tarmac primitives. Tables 1 and 2 give some specific measurements. Note the speedup given by use of the virtual movement facilities. All timings are given for MUs of 1K bytes, and do not include the cost of Unix IPC.

Table 1: Tarmac Primitive Execution Times

Primitive	Average Execution Time(usec)
Move	187
Copy	213
Virtual Move	107
Virtual Copy	123

Table 2: Memory Unit Location Times	
Location Task	Average Execution Time (usec)
Locating MU Following One Forwarding Link	456
Locating MU Following Three Forwarding Links	1476
Locating MU Corresponding to Label	245

4. RELATED WORK

In this section we contrast Tarmac with other systems for parallel distributed programming. We evaluate these other system in terms of Tarmac's goals: those of providing a language-system substrate for parallel distributed computation in a variety of programming models. The limitations we point out in these systems are relative to this goal, and are not intended as criticisms of the systems in general.

The key property of Tarmac is that it allows a memory unit to be viewed as both 1) an abstract entity that can be named, moved, copied, *etc.*, and 2) an array of bytes that can be directly manipulated by statements of an arbitrary programming language. Other systems do not afford this flexibility.

4.1. Distributed Virtual Memory

Distributed virtual memory provides the abstraction of a consistent virtual address space shared by processes running on separate hosts. A protocol similar to the cache consistency protocols used in shared-memory multiprocessors governs page movement and replication. Examples of systems providing distributed virtual memory include Ivy [12] and Apollo Domain [11].

The distributed virtual memory model has several possible drawbacks as a general substrate. First, because the resolution of page-table mapping is that of a fixed-size page (typically 8KB to 32KB on current machines) the granularity of state operations (movement and replication) is fixed. For programs that put many small data items on a single page, this granularity may be too large, causing increased contention and excessive data movement. Second, the model's

concurrency control mechanism (single-writer, multiple-reader) dictates the data movement policy. Clients have no direct control over data movement, and it may be difficult to prevent "data thrashing" (situations in which a shared data structure is moved frequently between clients, each of which accesses it only briefly). Finally, distributed virtual memory is not scalable. The system cannot keep track of pointers within a page to other pages, so when a non-resident page is accessed the request must, in some cases, be broadcast.

4.2. Shared Tuple Space

Linda [8] is a system based on the abstraction of tuple space shared among multiple concurrent processes. Its goals are similar to those of Tarmac: to provide a multiple-language substrate for parallel computing. The Linda model has the following limitations. First, the client has no direct control over tuple placement or communication pattern. Each Linda kernel implementation dictates a particular policy, determined by when and where *in()* and *out()* messages are sent. A language system substrate should not dictate policy, since a fixed policy cannot work well for all possible applications.

Second, the Linda model requires all shared state to be encoding into tuples. This imposes extra work on some applications. Finally, the model does not scale well in all cases. Linda kernels must resort in the worst case to either broadcast or centralization.

4.3. Master/Slave Systems

In the master/slave model, a single *master* process generates and accepts asynchronous function calls that are performed in parallel on a set of *slave* processors. There is no communication between the slaves. Marionette [14] is an example of such a system. Marionette provides shared global state that is read/write by master and read-only to slaves. Because the master processor is a bottleneck, the master/slave model has a limited range of values of the (grain-size, number of slaves) pair in which it performs well.

4.4. Object-Oriented Systems

Many system have used the object model for parallel distributed computation. Some of these systems, such as Matchmaker [9], and the Apollo Network Computing Architecture (NCA) [7], are intended as a structuring mechanism for permanent storage and client/server interactions, They provide an interface description language to specify both halves of what is essentially an RPC connection. Other systems, such as Clouds [6] and Eden [1] implement objects as separate address spaces, leading to prohibitive performance penalties when many small objects are used.

We focus our attention, therefore, on systems that support small objects, efficient access to both local and remote objects, and object mobility. Examples include Emerald [10], Amber [5], Distributed Smalltalk [3] and Sloop [13]. These systems suffer from several drawbacks relative to the goals of Tarmac:

Parallel object models have difficulty accommodating typical numerical data structures such as arrays. Is one expresses the array as a single object, than all operations on the array must pass through a single processor, which is likely to become a bottleneck in communication or computation. If each of the array elements is a separate object, the latency of individual operations increases. In addition, most object-oriented systems are coupled to a fixed type system, and usually to a single programming language.

The Tarmac location hint forwarding scheme is similar to the Hermes location independent invocation mechanism, with two important exceptions. First, the Hermes system fits object location into an operation invocation mechanism with a fixed request/reply protocol. This precludes higher level systems from expressing higher level requests, such as a search for a whole group of objects or for any member from a group. Tarmac supports such requests. For example, one can use a label_UID to specify the target of a move. The moved MU will end up in the habitat of some object labeled with the label_UID.

Second, the Hermes system requires the program to explicitly specify, for each object *T*, all other objects to which *T* holds references. Hermes uses this interobject reference information to maintain a cache of location hints at each host. The cache contains a hint for each object for which a local object holds a reference. Hermes, however, does not make full use of the interobject reference information. References to objects may be either UIDs or virtual addresses. The latter occurs because objects can share an address space. However, when an object *T* moves from address space *A1* to address space *A2*, objects in *A1* holding virtual address references to *T* will have to replace these references with UIDs (the reverse for reference holders in *A2*). Hermes does not do this reference patching, because its designers wanted to make it a language independent facility. At the same time, it is dependent on higher level language support in that it requires interobject reference information for its location independent invocation mechanism.

Tarmac avoids this dichotomy by keeping a location hint with each UID. This approach makes UIDs larger and causes memory units to contain possibly redundant location hint information. However, it releases language systems from the need to specify anything about a memory unit's internal structure. This makes memory unit creation more efficient, and decouples Tarmac from the type systems of its clients. Further, using this approach does not increase network traffic because each Tarmac server maintains a cache of hint updates.

5. CONCLUSIONS

Tarmac is a *language system substrate*: a foundation on which language systems for parallel distributed programming can be built. Tarmac provides a model of shared distributed state called *mobile memory*. Its client language systems determine the process structure, data placement policies, synchronization, and programming syntax by which this state is manipulated. Simply put, mobile memory provides two views of shared state:

- A state unit is simply a block of memory of arbitrary but fixed size. It can be accessed directly by arbitrary machine instructions, with no prescribed type system.

- All state units are encapsulated as logical entities with unique global names. These entities can be efficiently located, moved between processors, replicated, and so on.

In addition, mobile memory introduces some mechanisms not found in existing systems: virtual movement and copying, motion events, and label-directed data movement. The mobile memory model facilitates the dynamic allocation, redistribution, naming, and organization of the resources of a distributed computing system. Further, it provides language systems with the means to construct and maintain distributed data structures.

We have shown by example that mobile memory is sufficiently flexible to support a wide range of high-level models (object-oriented, dataflow, functional, database, *etc.*). Because it subsumes most details of management and communication of shared state, Tarmac greatly simplifies the implementation of such models. Finally, we have shown that the basic operations of Tarmac are efficient relative to typical network communication.

REFERENCES

1. G. T. Almes, A. P. Black, E. Lazowska and J. Noe, "The Eden System: A Technical Review", *IEEE Trans. on Software Eng. 11*, 1 (Jan. 1985), 43-59.
2. D. P. Anderson and D. Ferrari, "The DASH Project: An Overview", Technical Report No. UCB/CSD 88/405, Computer Science Div., EECS Dpt., Univ. of Calif. at Berkeley, Feb. 1988.
3. J. Bennett, "The Design and Implementation of Distributed Smalltalk", *Proc. of the 2nd ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications* , Orlando, FL , Oct. 1987, 318-330.
4. D. G. Bobrow, J. D. Burchfiel, D. L. Murphy and R. S. Tomlinson, "TENEX, a Paged Time Sharing System for the PDP-10", *Comm. of the ACM 15*, 3 (Mar. 1972).
5. J. S. Chase, F. G. Amador, E. D. Lazowska, H. M. Levy and R. J. Littlefield, "The Amber System: Parallel programming on Network of Multiprocessors", Technical Report 89-04-01, University of Washington, April 1989.
6. P. Dasgupta, R. J. LeBlanc and W. F. Appelbe, "The Clouds Distributed Operating System: Functional Description, Implementation Details and Related Work", *Proc. of the 8th International Conference on Distributed Computing Systems*, San Jose, California, June 1988, 2-9.
7. T. H. Dineen, P. J. Leach, N. W. Mishkin, J. N. Pato and G. L. Wyant, "The Network Computing Architecture and System: An Environment for Developing Distributed Applications", *Proceedings of the 1987 Summer USENIX Conference*, Phoenix, Arizona, June 8-12, 1987, 385-398.
8. D. Gelernter, "Parallel Programming in Linda", *Proceedings of the International Conference on Parallel Processing*, Aug. 1985, 255-263.
9. J. B. Jones and R. F. Rashid, "Mach and Matchmaker: Kernel and Language Support for Distributed Object-Oriented Systems", *Proc. of the 1st ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications* , 1986.
10. E. Jul, H. Levy, N. Hutchinson and A. Black, "Fine-Grained Mobility in the Emerald System", *ACM Trans. Computer Systems 6*, 1 (Feb. 1988), 109-133.
11. P. J. Leach, P. H. Levine, B. P. Douros, J. A. Hamilton, D. L. Nelson and B. L. Stumpf, "The Architecture of an Integrated Local Network", *IEEE Journal on Selected Areas in Communication 1*, 5 (Nov. 1983), 842-857.
12. K. Li and P. Hudak, "Memory Coherence in Shared Virtual Memory Systems", *Proc. 5th Annual ACM Symp. on ACM Conf. on Principles of Distributed Computing*, 1986, 229-239.
13. S. Lucco, "Parallel Programming in a Virtual Object Space", *Proc. of the 2nd ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications* , Orlando, FL , Oct. 1987.
14. M. Sullivan and D. P. Anderson, "Marionette: a System for Parallel Distributed Programming using a Master/Slave Model", *Proc. of the 9th International Conference on Distributed Computing Systems*, Newport Beach, California, June 1989.