A Portable Run-Time System for the Hermes Distributed Programming Language

ABSTRACT

We present our implementation of a portable run-time system for Hermes, a very high level language containing integrated constructs similar to those provided at the system and library level in systems like Mach and SQL.

The Hermes features we will focus upon in this paper are lightweight concurrent processes, the use of ports for typed communication between processes, a typed object store and its implementation via the Unix filesystem, and relational data structures for high-level associative queries.

Introduction

Hermes is a language developed at the IBM T. J. Watson Research Center to support programming of complex distributed systems at a very high level. A prototype implementation of Hermes was recently completed, including a compiler and an associated run-time system, and is being distributed free of charge in source code form.

The Hermes prototype has been implemented on Unix, and was designed to allow easy porting to other systems. One of the challenges we faced in developing the prototype was that of hiding details of the underlying platform from the Hermes programmer. In particular, a number of the features provided by Unix and required by Hermes programs, such as multiple processes, inter-process communication, and persistent data storage do not map easily and securely to the abstract model presented in Hermes. In what follows we discuss these features in some detail, and describe how our implementation bridges the gap between the Unix and Hermes models.

Language Design Goals

Hermes is a general-purpose programming language designed from the ground up for large distributed software systems such as operating systems, hospital information systems, window managers, and network management systems. Such programs share the following characteristics:

- large
- long-lived
- dynamic
- multi-user
- many cooperating implementers.
- maintainers may not have access to the implementers.
- parts of the system will change while it is running.
- multiple concurrent activities.

In our view, a language for this environment must be modular, reconfigurable, and secure. A very high level of abstraction is also required because it enhances portability, readability, and allows more extensive optimization.

This approach requires a more sophisticated compiler. In Hermes, a single construct often has multiple possible implementations, in which case the compiler tries to choose the best one or the programmer annotates the program with a pragma directive. Pragmas may change the performance characteristics of a program, but not its semantics. This allows concerns of correctness and performance to be addressed separately.

There are two major differences between Hermes and other languages aimed at the same type of problems (such as Ada); the first is that Hermes does not have any explicit pointers or aliasing. Instead, relational tables, records, and variants are used to create complex data structures, and copy-on-write techniques are used to avoid aliasing without large performance penalties.

The second difference is that there is no notion of a "main program" which begins execution, computes some result, and then terminates (relying on the operating system to reclaim its resources). The Hermes model is similar to that of an operating system: processes constantly enter and leave the system, and the "program" is simply the sum total of all of the active processes.

Operating systems generally provide protection and resource reclamation for address spaces, which commonly correspond to processes. In Hermes processes also provide protection and resource
reclamation, but without requiring a separate address space for each process. A mechanism called typestate checking does most of this work at compile-time.

In order to reach the widest possible audience, Hermes was implemented as a compiler to a portable abstract machine. A previous version of the language, called NIL or Network Implementation Language\textsuperscript{Str85, Bai89}\textsuperscript{a} shares most of the basic features of Hermes and was implemented as a native-code compiler on the IBM 370. Where appropriate we will mention techniques used in this implementation to create a high-performance compiled Hermes-like language.

The Process Model

Major Features

In the Hermes model, a process contains local variables and ports that provide connections to other processes. There are no shared variables, no aliasing, and no explicit concurrency within a process.\textsuperscript{1} Process creation and termination occur with great frequency, since in Hermes a process is the basic unit of program modularity, serving many of the same purposes as do procedures in other languages. Furthermore, the program text of a process is specified dynamically at process creation time, so new code can be introduced into a running system with ease.

Processes communicate by sending messages over ports. An import is a message queue, and an export is a capability to send messages to a particular import. Ports are discussed in detail in section 3.

A newly created Hermes process has an initialization port (an import) through which it may communicate with its creator (an output port connected to the initialization port is returned to the creator as the value of the create operation). The initialization port may be used to pass parameters and results, and to exchange whatever additional ports may be required.

Hermes processes are secure, meaning that it is impossible for one Hermes process to affect another in a way that the second process could not have anticipated (such as corrupting its storage or causing it to encounter a machine exception). Hermes guarantees security at compile-time through a mechanism called typestate checking\textsuperscript{Str86}\textsuperscript{a}; a process executing a typestate-correct program is guaranteed to be secure, and a program that does not pass typestate checking can never execute because it will be rejected by the compiler.

Because security is guaranteed by the compiler, multiple Hermes processes can be executed in a single address space without any danger that they will destroy each other's data. This is a significant advantage over writing applications with library- or system-level thread packages; experience has shown such programs to be extremely difficult to debug, since there is no address space isolation to protect the threads from each other.\textsuperscript{Con89b} Hermes thus achieves the security of heavyweight processes with the efficiency of lightweight processes.

Implementation

The Interpreter

The basis of the Hermes prototype is an interpreter, written in C and currently running on several variants of Unix (AIX on the IBM RS/6000, 4.3 BSD on the IBM RT, and SunOS on the Sun-3 and Sun-4). The interpreter executes instructions on behalf of running Hermes processes. In addition, the interpreter performs scheduling of multiple logically concurrent processes, and provides centralized memory management.

The "instructions" executed by the interpreter make up an instruction set named LI (for Language of the Interpreter). The LI was designed to efficiently support all the operations of the Hermes language in a form that frees the compiler from adhering to the structure of the original source.

An interpreter-based design was chosen for a number of reasons, including increased portability, ease of code generation, and the ability to investigate architectural features that might facilitate a Hermes implementation. An interpreter is definitely not a prerequisite for language security, as evidenced by the earlier NIL system.

Process Representation

A Hermes process is represented inside the interpreter by means of a process control block (pcb) containing, among other things:

- a pointer to the compiled program (a table of LI instructions) bound to the running process;
- the index of the next LI instruction to be executed;
- a pointer to the function that interprets instructions on behalf of this process; and
- the process' local storage.

An unusual feature of the pcb is the interpreter function pointer. Whenever an instruction is scheduled for execution on behalf of a process, the process' interpreter function is invoked to do the work. For ordinary Hermes processes, the interpreter function is simply one that retrieves the LI instruction indexed by the instruction pointer, decodes its opcode, and then dispatches accordingly. However, "foreign" processes can also be accommodated via this mechanism. In particular, the

\textsuperscript{1}It should be stressed that these statements refer to the conceptual Hermes process model; the implementation is free to make transparent use of techniques like data sharing and concurrent execution for performance or other reasons.
The current implementation includes a small number of processes written in C that provide access to various Unix services such as file I/O and window facilities.

Such foreign processes look like normal Hermes processes to the interpreter and to other processes (since they may own ports), but their pcb’s interpreter function pointers point to code generated by the C compiler. Foreign processes therefore have direct access to operating system services not available via the LI instruction set. Foreign processes can make use of the other pcb fields, including the instruction pointer and local storage areas, to maintain whatever state is required from one invocation to the next.

Clearly, the foreign process mechanism is insecure and therefore potentially dangerous to other running processes. However, the code implementing these processes is linked directly into the interpreter and cannot be loaded dynamically by user processes. We are thereby able to retain strict control on access to foreign processes. We plan to implement a more general framework for interaction with programs written in nonsecure languages.

Procedure-Like Processes

The Hermes process model admits a wide variety of process behaviors, including processes that provide multiple services and save state between calls. In practice, many processes fall into a category that we call “procedure-like,” characterized by the following:

- The process responds to a single type of call.
- No state is saved between calls to the process.

We use the term “procedure-like” because the programs for these processes look much more like the procedures one finds in other languages than do other Hermes processes. The standard sequence is: receive a call, perform some work, return results, and terminate.

One other procedure-like characteristic is made somewhat difficult by the normal Hermes process mechanism, namely ease of repeated calls. If a Hermes program is written in the style just described, the caller must create a new process to run the program before each call. Of course, this can be alleviated by wrapping the procedure body in an infinite loop; the process would then loop back each time it returned a call, ready to receive another. This approach would result in somewhat less readable code, because the loop serves only as an auxiliary role with respect to the function performed by the program.

To make programming of such processes more natural, Hermes provides a special variant of process creation called procedure creation. The creator of a procedure is given an output port just as in normal process creation. In this case, though, the port is not connected to the initialization port of a newly created process. Instead, the port is really a trigger for the creation of new processes; each time a message is sent on the port, a new process is created, and the message is queued on that process’ initialization port. Thus the caller is relieved of the burden of repeatedly creating the processes it needs to call.

Efficient implementation of procedures is important because the facility is heavily used in practice. To avoid incurring the overhead of process creation each time a procedure is invoked, the Hermes interpreter pre-allocates a pcb when the procedure is created. This pre-allocated pcb is called the primary pcb for the procedure. When the first call is made on the procedure port, the primary pcb is used to (cheaply) create a process to service the request. At that point the primary pcb is marked in-use, and any additional calls made on the procedure port will require the creation of new secondary pcb’s. Once the process executing with the primary pcb terminates, the primary pcb is no longer in-use and can therefore be reused when the next procedure call arrives. In practice, most procedures are called in a serial fashion, allowing the primary pcb to be used for all invocations.

Process Scheduling

In the current implementation, the interpreter uses a simple round-robin scheduling discipline by maintaining a circular queue of ready processes. In each scheduling cycle the following steps are performed:

1. Execute one LI instruction for the current ready process.
2. If the current process was made to block by the instruction, remove it from the ready queue.
3. If any blocked processes were revived by the instruction, add them to the ready queue.
4. If an asynchronous I/O event occurred, add any processes that were waiting for it to the ready queue.
5. Advance the ready queue to make the next ready process current.

As described in more detail in section 3, the call operation provides the primary mechanism for interprocess communication in Hermes, and is the closest counterpart to procedure calls in other languages. The operation causes a message to be sent to another process along with an identification of the calling process. The caller then blocks while the callee dequeues the message, acts on it and perhaps alters it, and eventually returns it to the caller. At that point, caller and callee proceed (in particular, the return statement does not cause a process to terminate).

2The process would eventually terminate with an exception on its receive statement when all potential callers had discarded their calling ports.
There are many scheduling policies which may provide superior performance to our round-robin scheduler. Some possibilities include:

- Execute more than one instruction for a process before advancing the ready queue. Much of the scheduling overhead could be amortized over several instructions using this approach, rather than incurring the full overhead on a per-instruction basis.
- Continue executing the current process until it either blocks voluntarily or executes a set number of iterations of a potentially infinite loop. This policy was used in the NIL run-time environment.
- Give an elevated priority to a process executing with the primary pcb of a procedure, thereby increasing the likelihood that the primary pcb would be free when the next call was made to the procedure.

It should be noted that not all context switching requires participation by the scheduler. An example of such a situation is presented below in the discussion of the call/receive mechanism.

Storage Management

The Hermes interpreter manages storage for all processes in a single pool. In the initial implementation, the interpreter used malloc() and free() each time an object was created or destroyed. When profiling showed this to cause a substantial performance overhead, that portion of the interpreter was reworked to use "quick-cell" lists for all allocations of certain sizes determined by profiling. The interpreter now typically uses quick cells for over ninety percent of all allocations. Execution speed improved by a factor of approximately 2.5 as a direct result of this modification.

Each Hermes object is represented internally by a descriptor-value pair. The descriptor identifies the primitive type of the object (integer, record, table, output port, etc.) and points to functions that implement the various ubiquitous operations (copy, finalize, equality test, etc.) for objects of that primitive type. A single descriptor is shared among all objects of a given primitive type, so the descriptors reside in static storage.

While descriptive information is not strictly necessary in the object representation (since the Hermes language is strongly typed), its presence allows our implementation to deal with ubiquitous operations like copying and finalization in a simple, generic fashion. The information has also proved invaluable for debugging the interpreter.

The value portion of an object is an instance of a C union type with a component for each primitive type. Some additional descriptive information is included here in cases where that information is not common among all objects of the given primitive type. For example, the value portion of a record object includes the number of components contained in the record, in addition to the data vector containing the component objects themselves.

The semantics of some of the Hermes primitive types allows storage to be shared easily among multiple copies of objects. An example is the Hermes nominal type, which provides a mechanism for creating globally unique values for use as tags, keys, etc. The only operations that pertain to nominals are creation, copying, moving, equality testing, and finalization. In particular, once created a nominal may not be altered. As a result, storage for multiple copies of a nominal can be shared without concern for future access conflicts; a reference counter ensures that a nominal persists until all copies have been finalized. Similar treatment is given to output ports and (because most operations on programs are as yet unimplemented) program objects.

A more complicated copy-on-write sharing mechanism is employed for table objects. When a copy is made of a table, the storage for the original table is shared and a reference counter is updated. Unlike nominals and output ports, however, a table may be modified. Therefore, each operation that will modify a table must first check whether it is currently being shared (i.e. the reference count is greater than one) and if so, create a new table whose elements are copies of those in the original table (embedded tables will get shared copies as a result). The resulting copy can then be altered without affecting other copies of the original.

The copy-on-write optimization is more difficult for other aggregate types such as records and variants, because operations exist to alter component objects of those types without making reference to the aggregate. Such a modification to an embedded object must be caught and cause the aggregate to be copied first. This problem does not arise with tables because modifications to table elements are disallowed; to update an element it must first be removed from the table, and then reinserted after the update.4

Inter-Process Communication

Major Features

Interprocess communication in Hermes is performed via typed objects called ports. Imports provide logically unbounded queues for incoming messages; outputs are access rights to place messages on import queues. Each output is connected to a single import, and can be used to queue messages only at that import. Multiple outputs can be connected to a single import, and messages from different outputs are merged fairly into the import queue.

4Notwithstanding this restriction, the compiler is free to generate code that performs updates in place where this is possible, for performance reasons.
Both synchronous (call-receive-return) and asynchronous (send-receive) communication are supported in Hermes. In synchronous communication, the calling process is suspended until the message is returned (possibly with alterations). The message data is wrapped in a special type of object called a callmessage, which identifies the calling process.

Asynchronous communication does not block the sender and does not provide for return of the data by the receiver. For both types of communication, the receiver must actively dequeue the message by performing a receive operation on the associated import. Hermes provides an operation to test whether an import's queue is empty and the select operation to multiplex message receipt from multiple imports by receiving a message from any available queue or blocking until a message becomes available.

All Hermes ports are typed, meaning that they are only capable of transmitting data of a particular type and typestate. Not only does this allow the Hermes language to remain secure despite the possibility of interactions between separately compiled processes, it also relieves the programmer from the need to convert data to and from some canonical serialize form such as a byte stream. Integers are transmitted and received as integers, and records as records; even large tables can be communicated without extraneous manipulation. Furthermore, ports are first-class objects, and can therefore be communicated among processes over suitably typed ports as easily as integers. In particular, communication of outputs provides many of the features of capability-based systems [Leb84a, Wal84a], but because Hermes is a secure language, costly run-time validation checks are unnecessary.

Because Hermes processes cannot share data, their only means of interaction is the passing of messages. One result of this is that Hermes programs can be distributed across multiple systems with ease, requiring only that the run-time system support passing of messages between machines. Furthermore, since the granularity of a Hermes process is roughly that of a procedure in other languages, relatively fine-grained distribution can be achieved with little work required by the compiler.

Distribution decisions are abstracted away from the programmer, who structures his programs without regard to machine boundaries. Instead those decisions are made either by the compiler or the run-time system. In contrast, a system like Mach provides tasks, threads, and procedures as units of program decomposition, and if the programmer decides that was previously a single-threaded task should now be a multi-threaded program distributed across several machines, significant rewriting will be required.

Example

To illustrate the constructs for inter-process communication, we present a simple "grep" process. The process is initialized as a server containing the string to grep for. It is given a capability to a line-oriented output function, PutLine, and returns a new capability of the same type which is bound to the grep process.

The callmessage interface definition is shown below; the first two parameters, PutLine and String, are input parameters and are declared as constants. The third parameter, ClientPutLine, is the new capability that is returned. The exit declaration says that the callmessage will be "full" on exit, meaning that all parameters will be fully initialized.

```plaintext
FilterDefs: using (stdio) definitions
  Filter: callmessage {
    PutLine: PutLineFn,
    String: Charstring,
    ClientPutLine: PutLineFn
  } constant(PutLine, String)
  exit (full);
  FilterQ: import of Filter
    {init(PutLine), init(String)};
  FilterFn: output of FilterQ;
end definitions
```

The import declaration includes the type of objects that will be received, namely Filter callmessages, and the typestate (the initialization level) of the parameters on entry to the import. In this case, the first two parameters are initialized, but the final parameter is not.

An example of the use of the grep process is shown below. First the grep process is created, and a capability to its initialization port, of type FilterFn, is returned and assigned to makeGrep. makeGrep is then called, passing the putLine capability from the standard environment and the string "Hermes". The grep process takes these two objects and creates a new port to itself, returning a capability which is assigned to grepPutLine. Finally, the client process iterates over a table of strings and outputs them via grepPutLine, so that only the strings containing "Hermes" will be displayed.

```plaintext
grepUser: using (filterDefs,stdio)
  linking (grep) process (q: mainQ)
declare
  makeGrep: FilterFn;
  grepPutLine: PutLineFn;
  string: Charstring;
  textFile: CharstringList;
  ....
begin
  -- some initialization code goes here
  -- create "Hermes" grepper
  makeGrep := create of process grep;
```
A Portable Run-Time System for the Hermes...

```plaintext

greppedPutLine :=
    makeGrep(stdEnv.putLine, "Hermes");

-- now grep textFile for "Hermes"
for string in textFile | inspect
    call greppedPutLine(string);
end for;
....
end process
end if;
return grepArgs;
end while;
end process

Implementation

The implementation of intra-interpreter communication is straightforward. Imports are represented as linked lists of queued messages, and exports are simply pointers to imports. The only semantic subtlety is that imports must be fair: no client output may be "starved" because other clients are flooding the import with messages. But because imports are FIFO queues, the round-robin policy of the scheduler guarantees fairness.

Although Hermes programs always use message passing with call-by-value semantics at the logical level, several different parameter passing mechanisms are used in the implementation. For single-word quantities such as integers and output ports, parameters are passed by copying the value into the callmessage on call and copying it back on return. For larger objects such as records and imports, a pointer to the object is passed. Since the compiler insures that an object can only be accessed by one process at a time, parameters can be passed by reference within an address space without sacrificing security.

In the NIL compiler, performance analysis showed that very large amounts of time were being spent enqueuing messages for blocked processes. As a result, imports were modified to contain a function pointer to an enqueuing routine. When the process owning an import is blocked waiting for a message to arrive on it, the enqueuing function simply saves the current registers of the process, loads the parameters into the registers, and jumps directly into the called process. The result is an eight instruction context switch between protected processes.

The implementation of inter-interpreter communication is more complex, although the high-level abstractions of Hermes make this task considerably simpler than in other languages. There were a number of Unix limitations that had to be overcome, most notably the limited number of file descriptors and the lack of asynchronous communications facilities.

Distributed Hermes uses Sun RPC and XDR protocols over TCP for communication. For each Hermes data representation, there is an associated XDR encode/decode routine which is automatically invoked by the run-time system for a remote call, return, send, or receive. Because there are no explicit pointers in Hermes, the only pointers are those created by the run-time system to implement the Hermes abstractions, which makes it easy to locate all the portions of a data structure for transmission.

```
Because a Hermes interpreter runs many processes concurrently, it is not acceptable for it to block during a remote procedure call, particularly since the callmessage might never be returned! It is also not acceptable to block during a receive from a remote port, since there may be other runnable processes.

To accommodate these requirements, we created RPC procedures layered on top of the existing synchronous Sun RPC calls. They call the regular routines and then set the input file descriptors to generate SIGIO interrupts when data is available, and set the output file descriptors to suppress buffering of packets.

The small number of file descriptors available in Unix also made it necessary to implement an LRU file descriptor cache so that an interpreter could communicate with more than 15 other interpreters. A connection is always used at least once before being swapped to prevent thrashing.

Both the asynchronous RPC and the file-descriptor caching are part of an RPC enhancement library that is completely independent of the rest of the Hermes system.

In the initial implementation, distribution decisions are made on the basis of locality: each interpreter is created with I/O interface processes and some sort of shell for communicating with a user. New processes are instantiated in the same interpreter as the creating process. Remote creation and migration facilities are under discussion. The obstacles are not technical, but conceptual: what is the proper interface to the programmer or compiler for control over such decisions? This question is currently under (heated) debate.

Outports connected to remote imports are represented as triples containing a machine name (Internet address), interpreter name (RPC version number), and address of the remote import in that interpreter. When an outport is sent to the interpreter containing its import, it is converted back into a local outport.

The Typed Object Store

In most languages, complex data structures created in memory can not be stored on disk without performing some sort of conversion process which marshalls the data structure into a byte stream. Although facilities like XDR and $\text{regen}$ can partially automate these procedures, they generally do not handle pointer-based structures that contain cycles (such as graphs) or other kinds of pointer aliasing, and if some pointers are opaque, user-written routines must be introduced.

In the Hermes language, there are no pointers and no aliasing. Although the compiler represents programmer data structures using both pointers and aliasing, this is invisible to the programmer. Since the compiler has full control over the data structures, it was easy to provide run-time support for converting data structures to and from byte streams.

The same representation is used both for disk storage and for network transmission. As a result, Hermes objects (including compiled Hermes programs) can be shared by heterogeneous systems all mounting the same filesystem.

The same XDR routines described in the previous section on inter-process communication are used for converting Hermes objects into files. Details on conversion of table data types are provided in the following section.

The main disadvantage to using XDR for disk storage is the large amount of redundant information stored: applying the Unix compress utility to these files yields a five- to ten-fold compression. We plan to provide an option to store these files in compressed format, allowing I/O performance to be traded off against disk space.

**Tables**

**Major Features**

The Hermes table is a high level data abstraction for programming with homogeneous collections of objects. The Hermes table is designed to eliminate the need for such traditional user-defined data structures as arrays, pointer-based linked lists and trees, hash tables, and so on. The model is loosely based on the relational data model and allows queries and other operations to be programmed without regard to the underlying physical organization of the data.

Besides creation of tables and insertion of elements into tables, Hermes provides a uniform mechanism for selecting elements of a table to be scanned, removed, or copied out. The selector construct is used in each of these operations to identify, via an arbitrary test expression, which table elements are to participate.

Two optional attributes can be included in table type definitions. The first, ordered, indicates that the table elements exist in a fixed order relative to one another. The position of each element in this ordering is established (relative to the existing elements) when it is inserted into the table. Concepts like “the first element” make sense only when applied to ordered tables. Ordered tables provide indexed access, such as is normally associated with arrays.

The second attribute, keyed, implies that a certain portion of each table element forms a primary key (in the database sense) for that table element. The semantic implication is that no two elements in the same table can have the same key value. Multiple keys can be declared for a table type.
Figure 1 shows some code fragments illustrating the high level of programming abstraction afforded by the Hermes table type. Honesty requires us to point out that the final code fragment would perform very badly if compiled with the current Hermes compiler. It is one of the future goals of our project to demonstrate that high-level transformations can be used to produce extremely efficient code from high-level specifications such as this.

Following are the table type definitions used in the sample code fragments of Figure 1:

```haskell
-- a simple ordered table
charstring: ordered table of char (init);

-- an unordered set of distinct nodes
nodeSet: table of nodeID (init)
  keys (*);

-- an edge with an identifying string
edge: record {
  source: nodeID,
  sink: nodeID,
  label: charstring
};

-- an unordered set of distinct edges
-- with unique labels
edgeSet: table of edge (full)
  keys (source, sink) (label);

-- standard specification of a graph
graph: record {
  nodes: nodeSet,
  edges: edgeSet
};

-- a table that is both ordered and keyed
nodeOrdering: ordered table of nodeID (init) keys (*);
```

Implementation

Internal Representation

A uniform representation and implementation for all table objects would almost certainly perform quite badly, since no single representation will be efficient for all tables under all circumstances. For this reason, the Hermes run-time system supports a number of specialized implementations. The mapping of table instance to run-time representation is made by the compiler, and could be based on various criteria, including table attributes (ordered, keyed), hints (pragmas) supplied by the programmer, and profiling data.

Currently, the Hermes interpreter supports four internal representations for tables:

- Linked list
- Vector
- Character string (packed vector of byte values)
- AVL tree (self-balancing binary tree).

The internal representation of a table object includes, in its value part, one or more table descriptors identifying the representation(s) used for the table. A table descriptor contains, among other things, pointers to specialized functions to perform primitive operations on tables in the given representation. Thus, for example, the code to interpret an `extract` statement makes indirect calls to representation-specific `remove()` functions for each element satisfying the `extract` test. Pointers to the `remove()` functions are found in the table descriptors attached to the table.

The ability to attach multiple table descriptors to a single table means that the compiler can choose to maintain multiple representations simultaneously for a single table. Each representation may be highly efficient for some subset of the operations or queries that are likely to be performed on the table. This is analogous to the capability of maintaining multiple indices for a single relation in a relational database; though such a capability is common in database products, it is quite unusual as an implementation strategy in a general purpose programming language.

Adding a table representation to the implementation consists primarily of the following steps:

1. Design the necessary data structures.
2. Implement each of the required primitive table operations including, as appropriate, operations making use of positions (ordered tables only) or keys (keyed tables only).
3. Make a descriptor for the new representation available to the interpreter.
4. Modify the module of the code generator (which is written in Hermes) that chooses representations for tables, to make use of the new representation where appropriate.

The compiler currently uses a fairly straightforward set of heuristics for choosing table representations. First, a "primary" representation is selected, which is used by any operation that must scan through all the elements of the table:

- If the table is ordered, a vector or charstring representation is used. Charstring is chosen only if the compiler can determine that every potential table element value can be represented in a single byte.
- If the table is unordered and unkeyed, a linked list representation is chosen.
- If the table is unordered but keyed, one of the keyed representations (see below) serves as the primary representation.

After the primary representation has been selected, additional representations are established for every key appearing in the table type definition. AVL trees are currently used for this purpose, though we may soon switch to hash tables.
When a table operation involving a selector is used on a table with multiple representations, the compiler can choose among several access methods to locate the required table elements. Currently, the following strategy is used:

- If the selection test is not a conjunction or a simple test for equality, a loop is generated to scan the table using the primary representation, and each element is tested as it is scanned.
- If the table is ordered and a top-level conjunct in the selector is an equality test involving the position of the element being tested, an indexed vector access is generated. Any remaining tests in the selector are applied to the element retrieved by this probe, if any.\(^3\)

\(^3\)Note that in this case and in the case of keyed access, the indicated method may be disqualified based on the form of the other comparand(s). The details are omitted from this presentation.

```
-- Declarations that would be needed for code fragments
pos: integer;
string: charstring;
junk: charstring;
firstchar: char;
g: graph;
successors: nodeSet;
startNode: nodeID;
unplacedNodes: nodeSet;
sortedNodes: nodeOrdering;
unplacedNode: nodeID;

-- remove initial blanks from string, get copy of first non-blank character
pos := position of char in string where(char <> ' ');
extract junk from char in string where(position of char < pos);
firstchar := string[0];

-- find all immediate successors of the start node
successors := every of node in g.nodes
  where (exists of edge in g.edges
    where (edge.source = g.nodes[startNode] and edge.sink = node));

-- topologically sort a graph
begin block
unplacedNodes := g.nodes;
new sortedNodes;
while (size of unplacedNodes <> 0) repeat
  remove unplacedNode from node in unplacedNodes
  where (not exists of pred in unplacedNodes
    where (exists of edge in g.edges
      where (edge.source = pred and edge.sink = node)));
  insert unplacedNode into sortedNodes;
end while;
on (NotFound)
  call putLine("Graph is not acyclic.");
end block;
```

Figure 1: Sample code fragments illustrating programming with tables.

- If the table is keyed, and at least one of the keys is "covered" by the selection test, a keyed lookup using the AVL tree based on that key is generated, and any remaining tests are applied to the retrieved element, if any. A key is covered if each component key field appears in an equality test which is a top-level conjunct in the selection test.
- If neither positional nor keyed access is applicable, the scanning loop strategy is used.

External Representation
A few special tricks are used in converting tables to an external form either for transmission over a net-
work or for disk storage.

First, descriptive information for table elements is stored once for the entire table, rather than once per element. This is analogous to the elision of individual object descriptors in the internal table representation. This strategy is unfavorable only in the case of an empty table.\(^6\)

Second, even if a table has multiple internal representations, each element is encoded only once. Representation descriptors appear at the beginning of the external representation, so that decoding the table can be accomplished by creating a new empty table with the indicated representations, and then inserting each element as it is decoded.

Finally, special treatment is given to tables with a charstring representation. Normally, each element of a table is encoded by invoking the generic XDR routine to encode an object of the table element's primitive type. Because the XDR specification requires a minimum of four bytes per encoded value, this results in three-to-one ratio of wasted to useful space in the case of charstrings. To avoid this, a special check is made for the existence acharstring representation, and if it is present, a single call to a byte-oriented XDR routine is used to encode the entire table at once.

**Related Work**

Hermes differs from other languages for large, concurrent systems like Ada\(^8\) and Modula-3\(^8\) primarily in its process model and its substitution of tables for pointer-based structures.

Ada and Modula-3 provide support for concurrency within an address space, but require extra-lingual constructs to be used for communicating across address-space and machine boundaries. Because of their shared-memory model, transparent distribution is much more difficult.

These languages also do not provide for dynamic compilation and linking of code.

The Hermes model of processes communicating by ports is very similar to that of Mach\(^7\). However, since Mach is providing an operating system service, it does not perform type checking across ports and must use address space protection to provide security. By providing these abstractions within the language, we can move much of this work into compile-time typestate checking, allowing multiple programs to execute securely within a single address space.

---

\(^6\)We have considered, and may adopt in the future, a special representation for empty tables, both internally and externally. A special internal representation would eliminate the overhead of building data structures at table creation time in the fairly common case that the table remains empty throughout its lifetime.

---

This frees programmers of the necessity to choose between fast, insecure communication (between threads) and slow, secure communication (between tasks), each with its own programmer interface.

**Conclusions**

The Hermes language allows distributed applications to be created more easily than existing languages and systems. This is due in large part to its very high level view of communication and computation.

The high-level view also allowed certain parts of the implementation to be much simpler than that of other languages, for instance the object store and the remote communications. On the other hand, many Hermes constructs are very powerful and require substantially more sophistication on the part of the compiler.

The table data type requires the compiler and run-time environment to provide a large portion of database functionality, and to manage the automatic mapping of a single data type into multiple data structures.

Our initial plan was to quickly build an interpreter-based system and then use it to bootstrap a native-code compiler. While we still plan to follow this path, the effort of creating the interpreter was significant enough that in retrospect, our effort might have been better spent just building a compiler from the start. In addition, the movement toward standardization during the last few years has made many of our original portability concerns, which led us to choose an interpretive design, obsolete.

The Hermes system is designed for exactly the kind of computation environment which is emerging among Unix systems: distributed networks of uni- and multi-processors. The security, dynamics, and powerful high-level operators of Hermes should make it possible to rapidly create applications for this environment. We are making Hermes available in source code form in the hope that others will use it for their own studies of the interaction between computer hardware, networks, operating systems, and languages.

To request Hermes code and documentation, send e-mail to hermes-request@ibm.com.

**Acknowledgements**

The authors wish to acknowledge the many people who have contributed to the Hermes implementation, either through their ideas or with actual code, including Rob Stom, Shaula Yemini, Danny Yellin, Arthur Goldberg, and Joel Auslander.
A Portable Run-Time System for the Hermes...

David Bacon began programming computers at 12 and got his first programming job at 15. He received his A.B. in Computer Science at Columbia University and now works at the IBM T. J. Watson Research Center, Hawthorne, where he was a designer and lead implementer of the Hermes language. His other interests include opera, fencing, bicycle touring, and transparent optimistic transformations for such problems as fault-tolerance, parallelization, and replication.

Andy Lowry received his B.S. and M.S. and is currently pursuing his Ph.D., all in Computer Science at Columbia University. He is currently conducting his research as a full-time employee at the IBM T. J. Watson Research Center. As a member of the Hermes project, Andy has contributed substantially to the implementation of the Hermes language. He has over ten years experience working as a computer programmer. His research interests include optimistic techniques, especially as related to parallelization, database concurrency control, and replication of data in distributed systems.

References


