A Mechanism for Communicating in Dynamically Reconfigurable Embedded Systems[†]

Mehrdad Hassani and David B. Stewart

Dept. of Electrical Engineering and Institute of Advanced Computer Studies
University of Maryland, College Park, MD 20742
Email: *mehrdad@eng.umd.edu*; *dstewart@eng.umd.edu*; Web: *http://www.ee.umd.edu/serts*

***Abstract**: We present a time-bounded state-based communication mechanism for dynamically reconfigurable embedded systems. The mechanism is a single-processor, low-overhead version of the Chimera state-variable mechanism, that was developed for state-based communication in multi-processor environments. The new design is suitable for execution on low-performance embedded processors, uses less memory, and supports dynamic binding, one-to-one, one-to-many, and broadcast capabilities in a time-deterministic manner.*

1. Introduction

The Chimera Project [4] demonstrated the use of state-based communication to create dynamically reconfigurable real-time software objects for a multiprocessor environment. Its *state variable* (SVAR) mechanism provides the user with operating system services that support transparent time-bounded inter-processor communication for high assurance control systems. The mechanism is fully deterministic, and has guaranteed worst-case waiting and transfer times for shared data. In conjunction with Chimera's port-based object model of real-time software components, it improves predictability and performance by minimizing inter-object dependencies as compared to message-based systems. The mechanism also supports dynamic binding, one-to-one, one-to-many, and broadcast capabilities.

The SVAR model of communication is also suitable for creating dynamically reconfigurable software for embedded systems [6]. The characteristics of embedded systems introduce different challenges into the design of an SVAR mechanism, as compared to Chimera's version. In particular, embedded systems usually use lower performance processors with less flexibility, have significant memory and CPU bandwidth limitations, and timing constraints are often more rigid than those allowed in a multiprocessor environment. The binding of communicating objects must also be fast and time-bounded to support dynamic reconfigurability.

Although the Chimera mechanism is very effective for meeting its goals in a multiprocessor environment, it has several shortcomings when applied to embedded systems. Its usage of memory is inefficient, as the local table is a duplicate of the global table, even though a process accesses only a small subset of it. It has larger overhead for locking the table, due to its multiprocessor support. The analysis of the mechanism is specific to a VMEbus hardware environment. Chimera's mechanism is also specific to its port-based object model, and only supports two pre-programmed operations, one for reading, one for writing.

By redesigning the mechanism, we have made improvements that address all of the above shortcoming, making the SVAR communication practical for embedded systems. The mechanism has also been decoupled from the internals of the Chimera Real-Time Operating System [5], allowing it to be ported to any environment.

2. Background: State-Based vs. Message-Based Systems

State-based communication is preferred in embedded systems to provide higher assurability. In a message-based system, processes are often synchronous and aperiodic, making real-time scheduling analysis difficult. There is significant overhead in sending and receiving messages, and processes waiting for data might block for an undetermined amount of time. Crucial messages can also get lost, as a result of buffer overflow if processes don't all execute at the same frequency. Control systems also have many feedback loops; sending messages in such an environment creates a risk for deadlock.

In contrast, a state-based system uses structured shared memory, such that communication has less overhead. The most recent data is always available to a process when the process needs it. Streenstrup and Arbib developed the port-automaton theory to formally prove that a stable and reliable control system can be created by only reading the most recent data [3]. Costly blocking is eliminated by creating local copies of shared data, to ensure that every process has mutually exclusive access to the information it needs.

Converting control systems from message-based communication to state-based communication is generally straightforward. We demonstrate this by example.

Example: *A train control system has independent control of every brake to maximize train handling. To minimize stopping distance when coming to a full stop, all the brakes on the train must be applied together. The input/output (I/O) logic for each brake is handled by a separate process; the control module must inform each brake module to turn on the brakes.*

Figure 1 shows the solution using a message-based system (e.g. [1]). The controlling unit sends a message, "apply brake", to every brake process. This approach has high communication overhead, potential loss of messages if tasks execute at different frequencies, non-deterministic blocking, a separate copy of the message for every process, and there exists the possibility of deadlock. Due to the dependencies among processes, it creates a real-time system that is difficult to analyze and is not suitable for reconfigurable systems [4].

In a state-based communication mechanism, the brake system is represented by a global state variable in shared memory. It can be either OFF or ON, as shown in Figure 2. Each brake module executes periodically, and monitors this vari-

[†]Research presented in this paper is supported, in part, by Pulse Electronics, Inc., a division of Westinghouse Air-Brake Company, and by the Dept. of Electrical Engineering and the Institute of Advanced Computer Studies at University of Maryland, College Park.

able to update the state of its own brake I/O. Since processes are periodic, a schedulability analysis is easier. Processes only need to bind to a single element in the state variable table, thus eliminating direct dependencies between processes. Communication through shared memory also incurs less overhead as compared to a message-passing system.

To ensure the integrity of the data, it is necessary to synchronize access to the shared memory. A common solution is to use semaphores, as shown by the *mutex* variable. However, using semaphores significantly increases overhead, potentially incurs additional blocking which lowers the schedulable bound of a real-time task set, and if not carefully used, can lead to priority inversion, starvation, or deadlock [2]. The use of semaphores thus eliminates the advantages of using a state-based system as opposed to a message-based system.

The priority ceiling protocol [2] can help minimize the blocking time and prevent deadlocks. Implementation of the protocol, however, still has significant overhead and potentially long blocking times for lower-priority processes.

The Chimera Project addressed these issues in the design of its SVAR mechanism. Chimera's SVAR mechanism uses a two-level shared memory structure, as shown in Figure 3. Shared data is stored in the global table. Processes, however, only access the data in the local tables. At the beginning of a process's cycle, the most recent data is always available, as per the port-automaton theory [3]. Only at the end of each cycle is newly created data updated in the global table. Transfers between the global and local table occur only when the process is not using the local data. The software framework for Chimera's port-based object, as detailed in [4], ensures that the communication occurs at the proper times.

Integrity of the data is achieved by using a combination of spin-locks on a global lock, and locking the local CPU, rather than using semaphores. The solution is suitable for a multiprocessor environment, and spinning on a lock eliminates the high overhead blocking time, replacing it with small amounts of waiting time. From a theoretical perspective, locking the CPU would cause implicit priority inversion. However, considering the practical aspects of real-time operating systems, where the CPU is often locked for extended amounts of time (e.g. for context switch), this method does not reduce the predictability of a system when data transfers are small. A detailed real-time analysis of SVAR communication for task sets which use this mechanism is given in [4]

3. SVAR Mechanism for Embedded Systems

The SVAR mechanism for embedded systems is similar to Chimera's mechanism shown in Figure 3. Our key contributions are a new design which uses less memory, eliminates the spin-lock in a single-processor environment, and decouples the mechanism from the real-time operating system (RTOS), allowing it to easily be integrated with any RTOS.

An architectural view of the SVAR mechanism is shown in Figure 4. In order to demonstrate the use of the mechanism, an example from Chimera is shown in Figure 5 (from [4]). Port-based objects communicate using state variables to form a control loop for a robotic manipulator. A more detailed look at the contents of the global and local tables is shown in Figure 6. No synchronization is needed to access the local table, since only a single process has access to it. Mutual exclusive access is only needed when a process exchanges information between its local table and the global table.

In a single-processor environment, the synchronization can be obtained by locking the CPU, assuming the amount of data to be transferred is small. It has been argued that locking the CPU leads to possible missed deadlines or priority inversion. This would be true in the ideal case where a CPU has no operating system overhead. However, considering the practical aspects of preemptive real-time operating systems, it is not unusual that a real-time microkernel, on an embedded low-

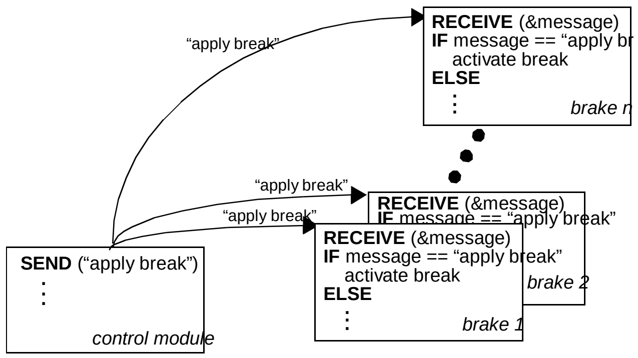

Figure 1: Example of a message-based system

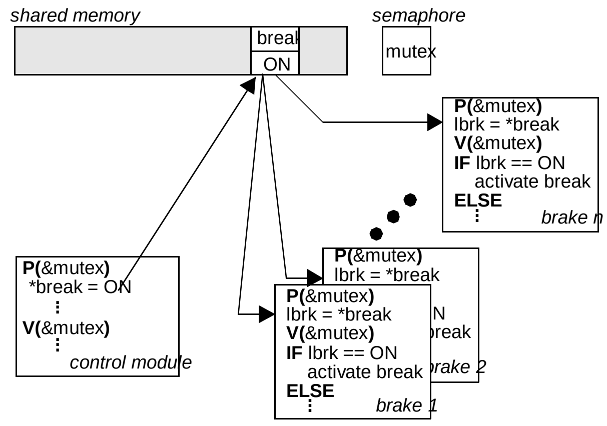

Figure 2: Example of a state-based system using shared memory and semaphores

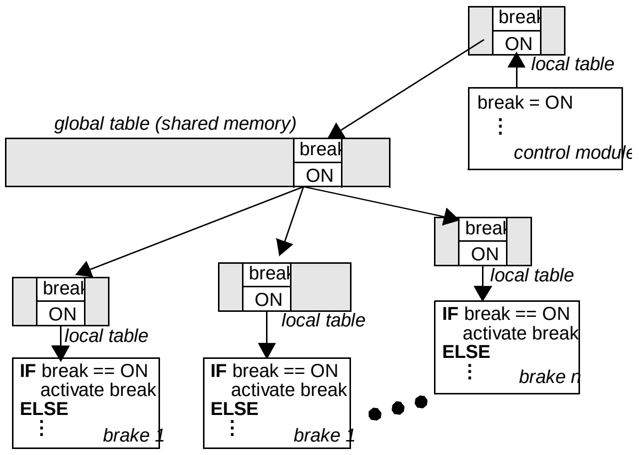

Figure 3: Example of state-based system using an SVAR

performance processor, locks the CPU for 100 μsec in order to perform a system call such as a full context switch [5]. If the total time that a CPU is locked in order to transfer state variables is less than the worst-case locking of the microkernel due to operating system functions, then there is no additional effect on the predictability of the system. Only the worst-case execution time of that task must be increased by the transfer time, and that can be accounted for in the scheduling analysis. As shown in Section 4, the worst-case locking and transfer times are not only bounded, but can be estimated accurately based on the number on inputs and outputs for a module.

An assumption that we make is that the volume of data stored in by the SVAR is small, which is the case in most embedded systems. One notable exception in which this assumption does not hold is for images. Vision applications can easily require several megabytes of data. The SVAR mechanism is not suitable for transferring such large volumes of data. However, once a vision subsystem detects specific points, edges, or objects, that low volume data can be transferred using the SVAR mechanism.

The remainder of this section provides details of the programmer interface, internal data structures of the mechanism, and details for creating and initializing the mechanism.

3.1 Programmer Interface

The contents of the global state variable table are specified in a configuration file, called the *.svar* file. This file can either be created manually by the programmer (generally for testing purposes) or automatically generated by a higher level interface based on the needs of an application. The information from this file is condensed and stored in binary and stored on the embedded system's EPROM or flash memory. A sample entry in the .svar file for the variable x_r is shown in Figure 7.

The state variable is created using *svarCreate()*, with the .svar file data passed as an argument. A global shared memory segment is dynamically allocated. Any task can then call *svarAttach()* to attach to a previously created table, at which time the data structures for an empty local table is created.

The *svarTranslate()* routine is called by a process to add a variable to the local table and translate its symbolic name into physical pointers to the data. The programmer can then use the local pointers indiscriminately during the main body code of the process. The *svarTranslate()* routine is typically only called during the initialization of a process.

The data in the local SVAR table can be updated at any time by performing either *svarRead()* for a single-variable update or *svarMultiRead()* for a pre-programmed multi-variable update. Similarly, the *svarWrite()* and *svarMultiWrite()* routines can be used for updating the global table based on the values in the local table. For the multi-variable updates, *svarProgram()* routine must first be used (usually during initialization) to program the list of the state variables that should be copied to or from the local table before using *svarMultiRead()* or *svarMultiWrite()* routines.

There is no error checking inside the *svarRead()* and *svarMultiRead()* routines. The *svarWrite()* and *svarMultiWrite()* routines can optionally perform range checking if the minimum and/or maximum values of a state variable are provided in the .svar file.

Table 1. Data structure fields of global SVAR table

Field	Description
name	symbolic name of the table
nvar	number of variables defined in global table
totalsize	total size of the table in bytes
varoff-set	offset to the variable segment in bytes

Table 2. Data structure fields of local SVAR table

Field	Description
name	symbolic name of the table
nvar	number of variables
varlist	pointer to the variables data structure
shmtabl e	pointer to the global table
copylist	list of pre-programmed copies
nalias	number of aliases defined
aliaslist	pointer to the list of aliases

Binding is performed using the symbolic names of SVARs. If two modules want to communicate, but internally they used different names, an SVAR *alias* can be created, which simply gives an a new name to an existing variable. This feature is necessary for component-based software assembly, where the input and output ports of different modules could be defined independently, even if they must communicate. Aliases are created using the *svarAlias()* command.

When a local SVAR is no longer needed, *svarUntranslate()* is called to remove it from the local table. When a task no longer needs access to the SVAR table, it performs an *svarDetach()*, which frees the memory used for the local table. The global table is only freed when all processes have detached, and a process performs an *svarDestroy()*.

3.2 Data Structures

The design of the data structures for the local and global table are one of the keys to ensuring minimal overhead, bounded transfer times, and efficient dynamic binding for the embedded SVAR mechanism.

The data structures used for storing data in local and shared memory are designed especially to support dynamically reconfigurable software. All declarations, memory allocation, and bindings are performed at run-time, so that software components can be created independently of each other, yet still be able to communicate.

The data structures are shown in Figure 8. (a) shows the structure for the global table, (b) shows the structure for the local table, and (c) shows the structure of an individual SVAR. Definitions of the fields of the data structures are given in Tables 1, 2, and 3 respectively. Each structure is described in more detail below.

3.3 Global SVAR Table

The data structure for the global SVAR table is stored in shared memory. It contains the union of SVARs of all the modules that can be configured into the system. The table consists of a header section and variable segments. The header provides information about the table, and each vari-

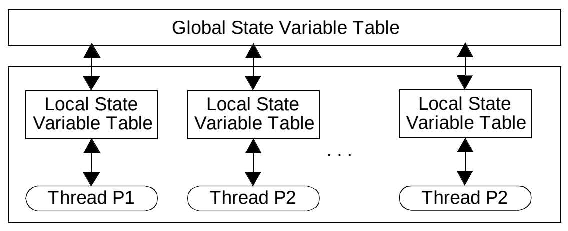
Global State Variable Table
Local State Variable Table
Local State Variable Table
Local State Variable Table
. . .
Thread P1
Thread P2
Thread P2

Figure 4: Structure of state variable table mechanism

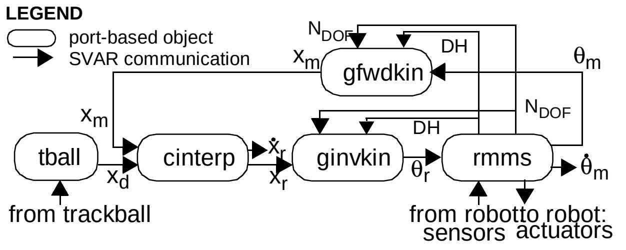

Figure 5: Example of component-based design using port-based objects

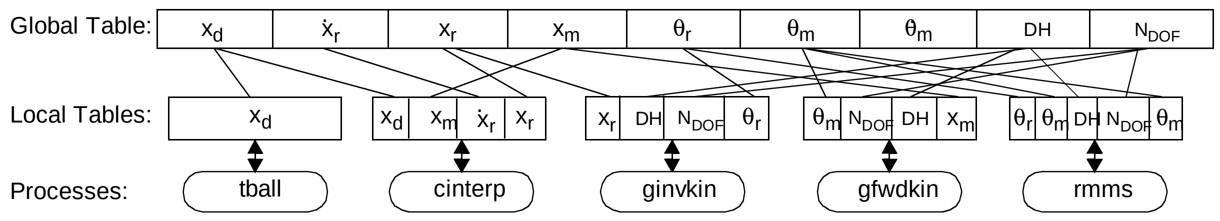

Figure 6: Contents of global and local tables for sample configuration shown in Figure 5.

NAME	**X_R**
TYPE	**float**
DESC	**Reference Cartesian Position**
UNITS	**meter**
NELEM	**6**
INIT	**5 5 6 7 2 4**
MIN	**-10 -10 -10 -10 -10 -10**
MAX	**14 12 15 15 18 12**

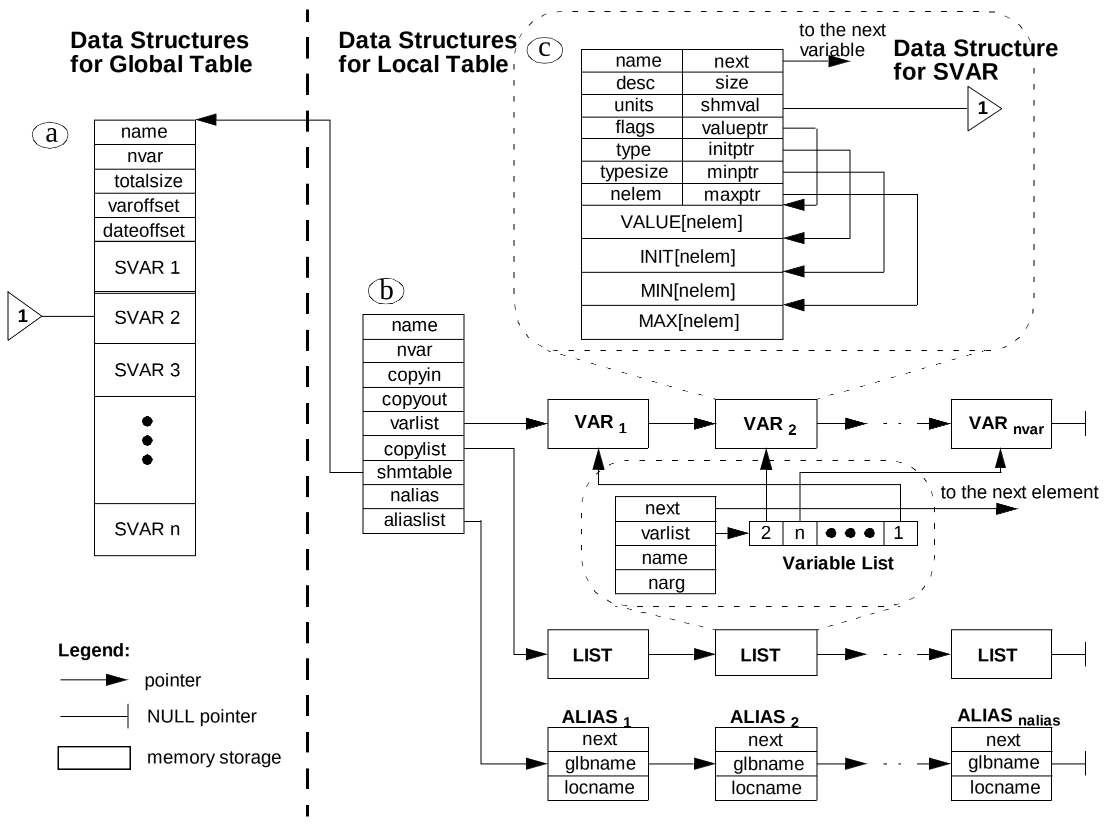

Table 3. Data structure fields for state variables.

Field	Description
name	symbolic name of the variable
desc	a symbolic description of the variable
units	unit of the variable
flags	specifies the type of check facility
type	type of the variable
typesize	sizeof(type)
nelem	number of elements of the variable: 1 scalar, 1< vector
next	pointer to the next variable
size	total size occupied by all the elements of the variable
valueptr	pointer to the value of the variable
initptr	pointer to the initial value specified in .svar file

able segment contains the global copy of an SVAR. The data in the variable segment is the same as the copy that is stored in the local table, and is described in Section 3.5

The global table is created dynamically when *svarCreate()* is called, as a contiguous block. Its structure remains unchanged during the lifetime of the table. Other processes in the system use the symbolic name of the table to attach to it using *svarAttach(),* which returns a pointer to the local table. A process cannot directly access the global table.

If the underlying RTOS does not support a uniform address space among all processes, then the shared segment might be mapped to different portions of each task's address space. As a result, all of the pointers within the global structure are stored as offsets, relative to the base of the shared segment.

The global structure is very similar to the data structure used in Chimera. The local structure, however, is very different.

3.4 Local SVAR Table

Unlike the global table, the local table is dynamic: different parts can be added and deleted during its lifetime.

In Chimera the local table is a duplicate of the global table, although each task accesses only a small subset of SVARs. To minimize memory requirements in our new design, *varlist*, *copylist*, and *aliaslist* are initially empty. *SvarTranslate()* adds a node to *varlist*; *svarProgram()* adds a node to *copylist*, and *svarAlias()* adds a node to *aliaslist*.

When *SvarTranslate()* creates a new node, it copies the data from the global table, and returns a pointer to the newly created local data. The process can then use the local data at any time. The *svarRead()* and *svarWrite()* routines are used to update the local and global tables respectively.

Nodes in varlist are also passed to *svarProgram(),* to preprogram a multi-variable copy. A *copylist* pointer is returned by *svarProgram(),* which is used by *svarMultiRead()* or *svarMultiWrite()* to initiate a multi-variable transfer.

Varlist, *copylist*, and *aliaslist* are generally created during the initialization of a process, thus the time to create the data structure is not a primary issue. The different nodes on these lists, however, provides quick access to the information needed during time-critical execution, to minimize the communication overhead. An analysis of the overhead, and sample performance benchmarks, are given in Section 4.

3.5 State Variables

As shown in part (c) of Figure 8, each SVAR has a common set of header information, followed by the data. In addition to the current value, both the local and global tables can also store the initial, minimum, and maximum values of that particular SVAR. This information is useful primarily for error checking and handling. Our embedded SVAR mechanism supports automated range checking, but due to space limitations it is not described in this paper. *Valueptr*, *initptr*, *minptr*, and *maxptr* are all available to the process, so that it can have direct access to the SVAR's data.

Header information such as *type*, *desc*, *units*, and *nelem*, are available to processes so they can obtain information about the SVAR to which it is attached. A process can verify these fields during initialization to ensure that the variable is indeed the correct one. Incorrect bindings can occur if the configuration manager or application designer uses the same variable name in two different processes to mean two different things. The problem can easily be remedied by using *svarAlias()* to locally change the name of one of them.

4. Analysis of SVAR mechanism

The SVAR mechanism was implemented and tested on an embedded system running QNX RTOS. The only RTOS-dependent code is to *lock()* and *unlock()* the CPU, and an interface to create a single shared memory segment. Since these features are available on most RTOS, porting the mechanism is simple.

To ensure predictable communication, the time required for transferring data between the local and global tables for each task must be computed. Let t_I be the time required to transfer input variables from the global to the local table, and t_O be the time required to transfer output variables from the local to the global table, assuming no waiting for another process that has already locked the CPU. t_I and t_O are computed as,

$$t_I = T_{\mathrm{ov}} + n_I \cdot T_{\mathrm{xfer}} + \sum_{i=1}^{n_I} R(x_i) \qquad (1)$$

$$t_O = T_{\mathrm{ov}} + n_O \cdot T_{\mathrm{xfer}} + \sum_{i=1}^{n_O} R(y_i) \qquad (2)$$

where T_{ov} is the overhead for locking and unlocking the CPU, excluding waiting time for the lock; T_{xfer} is the overhead of reading/writing each additional variable; n_I/n_O are the number of input/output variable of the task; x_i/y_i are the number of transfers required for input/output variables of the task; and $R(x)$ is the time required for x transfers.

T_{ov}, T_{xfer},, and $R(x)$ are dependent on the speed of the hardware. These values can be measured initially for each type of hardware supported to estimate the communication times. As an example, T_{ov}, T_{xfer}, and $R(x)$ were measured for both an embedded 100 MHz 486 and, to demonstrate the performance on a low-performance processor, on an original IBM PC with 4.77 MHz 8088. The results of our measurements are shown in Table 4.

Note that for the 486, the value of $R(x)$ is non-linear. This is due to the structure of the *blockcopy()* routine, which transfers multiples of 4 bytes in 16-byte blocks. If the transfer is not a multiple of block size, an additional overhead results for the

Table 4. Breakdown of Transfer Times and Communication Overheads

Operation	Execution Time (µsec) 8088 4.77MHz	Execution Time (µsec) 80486 100MHz	Variable name in Equations
locking CPU	3	0.6	
unlocking CPU	4	0.6	
block copy subroutine overhead	38	3.9	
raw data transfer, 1 float	19	0.5	***R(1)***
raw data transfer, 6 floats	115	1.9	***R(6)***
raw data transfer, 32 floats	557	4.8	***R(32)***
raw data transfer, 256 floats	4550	25.0	***R(256)***

Table 5. Comparison of Estimated and Actual Transfer Times on 100MHz 486 for Configuration in Figure 5

Module	n_I	x_i	t_I (µsec) Estimated	t_I (µsec) Measured	n_O	y_i	t_O (µsec) Estimated	t_O (µsec) Measured
tball	0	0	5.1	2.1	1	6	9.4	8.5
cinterp	2	6	11.1	9.5	2	6	13.7	12.4

incomplete block. This overhead is incorporated into the time for the raw data transfer time. Thus, the *blockcopy()* routine has a better time per data transfer for larger blocks of data. On the 8088 with an 8-bit bus, however, the block-copying does not provide any additional advantage for larger transfers. The overhead per variable on a multi-variable transfer, however, is still much lower on the 8088 as compared to doing only single-variable transfers.

Execution time for different transfer sizes can be estimated through interpolation, and measurements of *R(x)* for different values of *x* can give more accurate results. However, for purpose of discussion, the above values are sufficient.

The values in Table 4 can be substituted into equations (1) and (2) to estimate the transfer times for each process. We conducted an experiment using the sample configuration that was shown in Figure 5 to verify the above analysis. As can be seen in Table 5, the estimated and the actual values are sufficiently close to use the estimates in real-time scheduling analysis. This aspect is important since it is not desirable, and perhaps not feasible, to measure the execution time of communication for every module on every type of hardware.

Until now, all measurements and analysis assumed the ideal case where there is no contention for accessing the global table. If multiple tasks are attempting to obtain the lock, the one with the highest priority succeeds. Only one task can request the lock at once. Next, we will compute the worst-case waiting time for locking the CPU by any task.

Let L_{pj} be the time that task p with priority j locks the CPU. Thus, $L_{pj} = max(t_I, t_O)$. During this time the interrupts are disabled and task p can transfer data without any interruption. After releasing the lock, the remaining tasks compete to obtain the lock.

Let W_j be the worst-case waiting time for a task with priority j. Since this is the waiting time not blocking time (a *waiting* task is in the running state, a *blocked* task is suspended) W_j can be added to the worst-case execution time of a task. It is computed as

$$W_j = W_{jLO} + W_{jHI} \qquad (3)$$

where W_{jLO} is the maximum waiting time for a task with lower priority, and W_{jHI} is the maximum waiting time for tasks with priority j or higher.

W_{jLO} is computed as the largest time any single task with priority lower than j may hold the lock. Therefore,

$$W_{jLO} = max\left(L_{ki}\Big|_{n=1,\forall k}^{j-1}\right) \qquad (4)$$

To compute W_{jHI}, in the worst-case scenario, all tasks with priority j or higher may require the lock at the same time. Thus, task p have to wait for all other tasks to finish their data transfer to/from the global table, before it can obtain the lock. W_{jHI} is computed as the sum of the waiting times for all tasks with priority of j or higher

$$W_{jHI} = \sum_{i \geq j} \sum_{\forall k, k \neq p} (t_{I,ki} + t_{O,ki}) \qquad (5)$$

The notation $t_{I,ki}$ is the same as t_I, where k is the task ID and i is the priority of the task.

The computation of W_j shows that it is preferable for tasks producing high volumes of data to be assigned the lowest priority since it significantly reduces W_{jHI}. This is usually not a problem, because in many embedded systems, the highest frequency tasks produce the least amount of data, and are executed at highest priority according to the rate-monotonic algorithm.

5. Summary

In this paper, we present a time-bounded mechanism for state-based communication in dynamically reconfigurable embedded systems. Predictability, low overhead, dynamic binding and high reliability are among the contributions of the state variable table mechanism we developed. Detailed analysis and performance measurements are also provided.

6. References

[1] W.C.Athas and C.L. Seitz, “Multicomputers: message-passing concurrent computers,” *Computer*, v.21, n.8, pp. 9-24, 1988.

[2] R. Rajkumar, *Synchronization in Real-Time Systems– A Priority Inheritance Approach*, Kluwer Academic Publications, 1991

[3] M. Steenstrup, M. Arbib, and E.G. Manes. Port Automata and the Algebra of Concurrent Processes, *Journal of Computer and System Sciences*, v. 27, n.1, pp. 29-50, Jan. 1983.

[4] D.B. Stewart and P.K. Khosla. The Chimera Methodology: Designing dynamically reconfigurable and reusable real-time software using port-based objects, *Intl. J. of Software Engineering and Knowledge Engineering*, v.6, n.2, pp. 249-277, 1996.

[5] D.B. Stewart, D.E. Schmitz, and P.K. Khosla, P. K. “The Chimera II real-time operating system for advanced sensor-based robotic applications”, *IEEE Trans. on Systems, Man, and Cybernetics*, v. 22, n. 6, pp. 1282-1295, Nov/Dec 1992.

[6] D. B. Stewart and G. Arora, “Dynamically reconfigurable embedded software - does it make sense?”, in *Proc. of IEEE Real-Time Applications Workshop (RTAW)*, Montreal, Canada, pp. 217-220, Oct. 1996.