ON INTERFACE DESIGN FOR DISTRIBUTED SIGNAL PROCESSING

** Juan C. Díaz Martín * Juan A. Rico Gallego, *Jesús M. Álvarez Llorente, **Carmen Calvo Jurado*
[*]Department of Computer Science, [**]Department of Mathematics,
Escuela Politécnica, University of Extremadura. Avenida de la Universidad, s/n. 10071. Cáceres. Spain
phone: +34 927 257265, fax: +34 927 257202, email: juancarl@unex.es
web: gsd.unex.es

ABSTRACT

Manufacturers of real-time operating systems (RTOS) for DSP computers and multi-computers are mainly concerned on kernel size and performance. These RTOS rely on configuration tools that statically locate the application tasks across the available machines. This work describes IDSP, a distributed middle-ware for DSP multi-computers. It is not a new RTOS, but a framework upon one of them, currently Texas Instruments DSP/BIOS. IDSP proposes and researches process management and MPI-like message passing interfaces that make possible run-time creation of remote tasks and true location-transparent communication, These facilities are not yet present in commercial systems, but a they are a must for achieving more advanced capabilities such as process migration and fault tolerance. We describe the design of IDSP and give performance figures.

1. INTRODUCTION AND GOALS

DSP intensive applications such as speech engines or video processing are -and they always will be- strongly limited by its computational complexity. Distributed computing changes this scenery. Fortunately, most of algorithms and applications can be decoupled and distributed among two or more CPUs. Cooperative work between instances of signal processing algorithms is necessary in order to gain the scalability of present and future DSP developments. The state of the art in DSP multi-computers is well represented by the developments of Motorola ([1]), Sundance ([2]) o Hunt Engineering ([3]). These manufacturers rely on DSP real-time kernels such as DSP/BIOS, Virtuoso ([4]), VxWorks ([5]), OSE ([6]) or 3L Diamond ([7]) to name but a few. The 3L Diamond case study will put our contribution in perspective because its distribution model closely follows our abstract model. Under Diamond, a complete application is a collection of one or more concurrently executing *tasks*. A Diamond task is a separate multithreaded C program, with its own `main` function. Each task has a vector of input ports and a vector of output ports that are used to connect tasks together and that are passed to `main`. Each port is of type "pointer to channel" (`CHAN *`). Fig. 1 illustrates the Diamond message-passing interface over the ports.

```
#include <chan.h>
main(int   argc, char *argv[], char *envp[],
       CHAN *in_ports[], int ins, CHAN *out_ports[], int outs)
{
  int c;
  for (;;) {
    chan_in_word(&c, in_ports[0]);
    if (c == EOF) break;
    chan_out_word(toupper(c), out_ports[0]);
  }
}
```

Figure 1: A Diamond task.

A program called the *configurer* running in the PC host combines task image files to form the executable file. A user-supplied textual *configuration file* drives the configurer. It specifies the hardware –available processors and physical links connecting them, the software –tasks and connections between them, and how tasks are assigned to processors. Note that `chan_out_word (toupper (c), out_ports [0]);` sends the upper character to "the output port 0". No dynamic addressing is involved, what eases programming and yet it makes tasks communication transparent to specific locations. We understand that static configuration solves most of current practical problems, but it fails to face technical challenges such as run-time reconfiguration, task migration or fault tolerance in the DSP world. A software layer usually known as a distributed framework should ease the cooperation between objects running in different processors. IDSP is our contribution in that address (Fig. 2). MPI is the standard API for parallel programming ([9]). The IDSP framework proposes and researches MPI-like message passing interfaces that make possible the dynamic creation of remote tasks and true location-transparent communication, facilities not explored enough in present commercial systems.

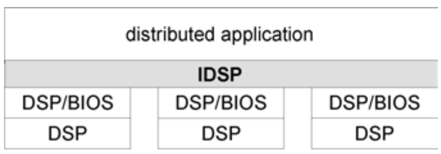

Figure 2: The IDSP framework.

The rest of the paper is structured as follows. Section 2 presents the concepts underlying the IDSP application model and its addressing scheme. Section 3 and 4 studies the process management and communication interfaces respectively, while section 5 shows the internal architecture. Finally, section 6 gives performance figures.

2. DESIGN PRINCIPLES

The key feature of IDSP is the assumption of a model of distributed application that consists of a graph of cooperating DSP algorithms running in one or more machines. A node in the graph represents an algorithm, served by a process that is known as an *operator*. Fig. 3 shows an application of five operators. An arrow represents a data stream.

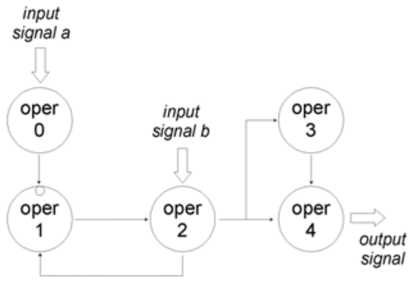

Figure 3. The IDSP application model

Conceived as a building block, a design principle of IDSP is keeping the operator a simple entity. Hence, it has a single thread of execution, currently a DSP/BIOS task. Typically, signal processing leads to an algorithm applied to data streams windows in an infinite loop. In our model, a loop iteration reads inputs in *sequence*, does the computing task, and writes to the output, going back for new input data. This activity pattern is suitable for a single thread. Notwithstanding, for the sake of regularity, IDSP also charges operators with non-DSP services. This is the case of the system servers, for instance.

The addressing scheme is one of the key features of a distributed system. Each operator in the system has assigned an address that distinguishes it from the rest in a global scope. Operators are the end points of a communication. The IDSP address is transparent to the operator location. It consists on the pair [gix, oix] -the *group index*, and the *operator index*. There should not be two groups with the same gix. IDSP provides a service to obtain a unique gix. A random number must be employed otherwise. The operator index, *oix* for short, identifies an operator inside a group, ranging from 0 up to the maximum number of operators in the group.

A data stream is a sequence of messages, usually signal windows. Fig. 4 shows the format of the IDSP message. Four fields compose it. The source and destination address, followed by the number of bytes of the data field and the data field itself. Some messages, notwithstanding, do not carry the signal data, but methods identifiers of the IDSP system RPC servers, their parameters and results.

SRC address	DST address	SIZE	DATA / method

Figure 4: The IDSP message format

Note that both kinds of information are supported by the same message format. Method information or signal data is irrelevant for the IDSP kernel. Hence, from now on, we can refer to them just as the *data* field.

3. THE PROCESS MANAGEMENT INTERFACES

This research has been carried out on Texas Instruments TMS320C6000 processors with DSP/BIOS ([8]). DSP/BIOS is the 25Kbytes sized kernel that Texas Instruments supplies with its DSP systems and it has therefore became one of the better known and more widely used RTOS. Raw DSP/BIOS, however, is not aware of other CPUs in a distributed memory multi-computer environment; hence the purpose of building the new process management interfaces. They use DSP/BIOS for just basic concurrency support and extend it with a run-time process management facility.

Each operator has a system-wide well-known integer name. Of course, all the instances of the same operator share the same name. The so named *operator register* is a module that keeps the features of the operators linked in memory, i.e., the operator name, the body function, the parameters size and the stack size. In some way, this register plays the role of a file system in a conventional computer, which keeps the executable files. The IDSP process management interface is simple:

```
Int    init    (Void);
Int    enrol   (Void);
Void   leave   (Void);
Int    create  (Opr_t *oper, Addr_t addr, Int name, char *param);
Void   destroy (Opr_t oper);
Int    start   (Opr_t oper);
Int    kill    (Opr_t oper);
Opr_t  self    (Void);
```

Init primitive initialises IDSP. Enrol allows a host RTOS task -a DSP/BIOS task, for instance, to become an IDSP operator and therefore invoke its interfaces. Leave has the contrary effect. Create creates a new operator, supplying it with its name and its global address. Destroy stops the operator and liberates its resources. Start schedules the new operator and, finally, Kill “disables” the operator, a state discovered by next or current kernel service and currently used to invoke leave.

The OPR interface manages the distribution of operators by allowing an operator to create another in a given machine, as well as to destroy, start and kill it. OPR is implemented by an RPC system service that exhibits the following interface specification. Note how it fits the kernel interface.

```
Int   OPR_create   (Addr_t addr, Int machine, Int code, char *param);
Void  OPR_destroy  (Addr_t addr);
Int   OPR_start    (Addr_t addr);
Int   OPR_kill     (Addr_t addr);
```

The GRP interface helps on process management by allowing operating on groups. A *group* is a set of related algorithms that cooperate in solving a task and it is known by a single identifier. Groups are created, started and destroyed by using its name. GRP is also implemented by an RPC service, built upon the OPR interface. This is its interface specification:

```
Int   GRP_create  (Int *gix, Int mode, Int *name, Void **parm, Int size);
Int   GRP_destroy(Int  gix);
Int   GRP_kill      (Int  gix);
Int   GRP_start     (Int  gix);
Void  GRP_leave     (Void);
Int   GRP_self      (Void);
Int   GRP_channel  (Int  gix, Int *inCh, Int *outCh, Int size);
```

GRP_create creates a group composed by the operators in name. Operators are assigned to processors following a load-balancing approach. This means, for instance, that group instance *g* can have the operator 4 running on the machine *m*, while the group *g'* can have its operator 4 running on the machine *m'*. When the mode parameter takes the GRP_GEXTGIX value, GRP_create just returns a system unique *gix* identifier. The GRP_GRAPH value creates a new group. Size is number of operator composing the group. GRP_destroy terminates the group by destroying its operators and liberating the group resources. GRP_kill kill the composing operators as above explained. GRP_leave allows the invoking operator to abandon the group. The last one destroys the group.

4. THE MESSAGE PASSING INTERFACES

The kernel shows a simple but yet powerful interface to send and receive messages:

```
Int  send     (Int sync, char *buffer, Int count, Addr_t dst, Int tag,
               Rqst_t *rqst, Uns timeout);
Int  recv     (Int sync, char *buffer, Int count, Addr_t src, Int tag,
               Rqst_t *rqst, Status *status, Uns timeout);
Int  waitany (Int count, Rqst_t *rqst, Int *index, Status  *status);
Int  waital   (Int count, Rqst_t *rqst, Status **status);
```

The sync parameter determines if send and recv operate either in synchronous or asynchronous mode. Send primitive sends count bytes of buffer buffer to dst operator, labelled with the tag tag. The K_E_DISABLED error is returned when the invoking operator has been disabled. The K_E_TIMEOUT error is returned when the rendezvous times out. Recv primitive is similar. The rqst communication object is returned when send and recv are invoked in asynchronous mode. Waitany and waitall suspend the operator until its communication request are satisfied. On other hand, upon these kernel primitives, IDSP build two higher level, user oriented communication libraries, group communication (GC) and remote procedure call (RPC). GC facility is quite similar to MPI. In fact, P4, a parallel library that supports MPI, has been ported to the C6000 architecture upon GC ([10]):

```
Int  GC_send     (char *buffer, Int count, Int dst, Int tag, Uns timeout);
Int  GC_asend    (char *buffer, Int count, Int dst, Int tag,
                  GC_Rqst_t *rqst,  Uns timeout);
Int  GC_bcast    (char *buffer, Int count, Int root);
Int  GC_recv     (char *buffer, Int count, Int src, Int tag,
                  GC_Status *status, Uns timeout);
Int  GC_arecv    (char *buffer, Int count, Int src, Int tag,
                  GC_Rqst_t *rqst, Uns timeout);
Int  GC_wait     (GC_Rqst_t  rqst,  GC_Status *status);
Int  GC_waitall  (Int count, GC_Rqst_t *rqst, GC_Status *status[]);
Int  GC_waitany  (Int count, GC_Rqst_t *rqst, Int *index,
                  GC_Status *status);
Int  GC_test     (GC_Rqst_t  rqst, Int *flag,  GC_Status *status);
```

One important difference between IDSP and DSP/BIOS objects is that the former ones can be in different machines. This fact poses the problem of remote invocation. Remote objects are often operated in distributed systems by using a technique known as RPC (Remote Procedure Call). The RPC system servers of IDSP also fit into its application model. They are implemented as the single instance of a single operator group. There are two main RPC system servers in IDSP: the group server and the operator server. Whilst there is one operator server per machine, there is a single group server in the whole system. The machine hosting the group server is called the *root* machine. RPC syntax and semantics have been inspired in the Amoeba operating system ([11]). Operators use RPC for accessing user or system services such as create groups or operators asking for CPU loads… OPR and GRP stubs and skeletons, for instance, use these primitives:

```
Int  RPC_trans (char *buffer, Int count, Int service);
Int  RPC_send  (char *buffer, Int count, Int dst);
Int  RPC_recv   (char *buffer, Int count, Int src);
```

At the highest level, DSP operators communicate through objects named *channels*. There are two kinds of channels, input channels, and output channels. Inside an operator, channels of the same sense are known by its order number 0, 1, 2, … Thus, channels complete the IDSP application model shown in Fig. 3. The programmer just sends data to output channel, say 2, and data arrives to the connected operators 3 and 4. The GRP_channel primitive supplies a just created group with two connection matrices, one for input channels and another for output channels. The operator creates, reads, writes and destroys the channels it uses by using the *channel interface* (CHN). Built on GC, CHN is a more flexible facility than the before mentioned static Diamond channels:

```
Int   CHN_create   (CHN_t *ch, Uns   mode,   Uns  channelNr);
Void  CHN_destroy (CHN_t  ch);
Int   CHN_send     (CHN_t  ch, char *buffer, Int  nbytes, Uns timeout);
Int   CHN_asend    (CHN_t  ch, char *buffer, Int  nbytes,
                    CHN_Rqst *rqst,
                    Uns timeout);
Int   CHN_recv     (CHN_t  ch, char *buffer, Int  nbytes, Uns timeout);
Int   CHN_arecv    (CHN_t  ch, char *buffer, Int  nbytes,
                    CHN_Rqst *rqst, Uns timeout);
Int   CHN_test     (CHN_Rqst *chRqst, Bool *flag);
Int   CHN_wait     (CHN_Rqst *chRqst, CHN_Status *st);
Int   CHN_waitall  (Int count, CHN_Rqst *chRqst, CHN_Status *st);
Int   CHN_waitany (Int count, CHN_Rqst *chRqst, Int *index,
                    CHN_Status *st);
```

5. A MICROKERNEL SOFTWARE ARCHITECTURE

IDSP rests upon two software engineering techniques that have proved to be a solid foundation for building robust software: layering and objects. An object is a data structure plus a set of operations over such data, also known as member functions or methods. Methods promote the software reusability by applying the principle of information hiding. They hide to the user the internal implementation of the object, allowing that changes in the implementation of the object do not affect the client code. Objects are created, operated on and finally, destroyed. In IDSP, a group is an object an operator is an object and a channel is an object. Every entity in IDSP is an object. Fig. 5 shows the microkernel architecture of IDSP. We can see how services as GRP and OPR have been segregated from the kernel and implemented as user servers that communicate through the kernel message-passing interface.

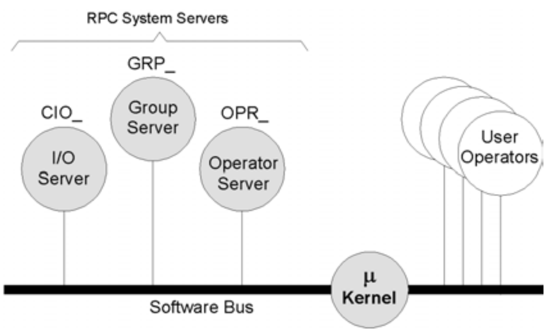

Figure 8: The IDSP architecture

6. MESSAGE-PASSING PERFORMANCE

In spite of its rich semantics (practically the ones showed by the MPI standard) the IDSP message-passing interfaces show reasonable performance.

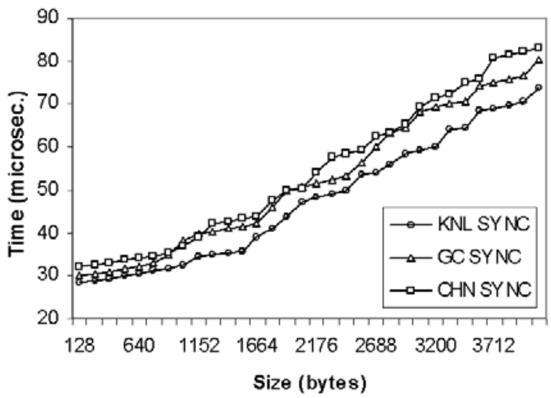

Figure 9: Time to send synchronous messages versus message size

Fig. 9 shows the time that sending a message takes to the synchronous primitives. A Sundance SMT310Q multi-computer board with four TMSC6102 DSP processors has been our test environment. As a reference, we measured that it takes 17 microseconds the highly optimised DSP/BIOS MBX_post primitive to send a short message to a mailbox. Asynchronous primitives show similar performance, being the added cost of further wait invocations around the 20%. The good performance of the P4 port on IDSP ([10]) supports the idea that IDSP is not slow.

7. CONCLUSIONS

A distributed framework for DSP multicomputers has been proposed. IDSP has been implemented on Texas Instruments TMSC6000 processors, but its use of DSP/BIOS makes it quite portable to other architectures. IDSP interfaces have been modelled after the MPI standard, what makes them powerful and flexible, and yet keeping IDSP small (about 60 K) and fast. In our view, its transparent location address scheme makes IDSP a tool for researching on distributed embedded systems. We are currently working on improving the IDSP interfaces and using them to support distributed speech recognition engines and build a MPI port. We plan future work on implementing and testing the MPI/RT specifications on the DSP world.

8. ACKNOWLEDGEMENTS

CICYT and Junta de Extremadura founded this work under the TIC99-0609 (DIARCA) and IPR00C032 projects respectively.

REFERENCES

[1] http://mcg.motorola.com/us/general/CPCIC6400.pdf
[2] http://www.sundance.com
[3] http://www.hunteng.co.uk
[4] http://www.transtech-dsp.com/software/virtuoso.htm
[5] http://www.windriver.com/products/vxworks5
[6] http://www.ose.com/downloads/pdfs/products
[7] http://www.3l.com
[8] www.ti.com/tmwbios
[9] W. Gropp, E. Lusk, N. Doss, A. Skjellum, "A High Performance, Portable Implementation of the MPI Message Passing Interface Standard". Parallel Computing, 22, pp. 789-828 (1996).
[10] J. A. Rico, J.C. Díaz, J.M. Rodríguez, J. M. Álvarez, J. L. García, "Porting P4 to Digital Signal Processing Platforms", in Proc. 10th European PVM/MPI User's Group Meeting (EuroPVM/MPI 2003). Venice, Italy, Sep 29 - Oct 2, 2003. Lecture Notes in Computer Science LNCS 2840. Springer, pp. 362-368.
[11] A.S. Tanenbaum et al., "Experiences with the Amoeba Distributed Operating System". 1990. Comm. ACM, vol. 33, no. 12, pp. 46-63.