John Poplett and Rob Kurver

The DSI Transputer Development System

Definicon Systems' new Transputer coprocessor board puts concurrency in your IBM PC

New-generation microprocessors like the 80386 and the 68020 have brought the power of a minicomputer onto the desktop. Yet even this power is inadequate for software designers who are working with applications that run mainly on supercomputers, such as ray tracing and other solid modeling graphics.

Increased microcomputing power is emerging from two main areas of technological development: high-speed reduced-instruction-set computer (RISC) technology and parallel processing. RISC technology offers the promise of microcomputer systems that approach today's supercomputers in performance, yet it requires little change in the way current software is developed. Parallel processing offers the promise of systems whose computing power is limited only by the resources of the system designer and the ingenuity of the programmer.

BYTE has arranged with Definicon Systems to offer BYTE subscribers the TG2 multiprocessor board and TCC, a parallel C compiler, at a special introductory price (see the text box on page 250). The board and compiler let you develop software in parallel at a cost lower than previously possible. This board contains two 32-bit INMOS Transputers, a host interface, and a television-quality graphics section on a single IBM PC expansion bus card.

The INMOS T414 Transputer

The idea of yoking together a number of processors in parallel to perform a computationally demanding task is as conceptually simple as it is difficult to implement. The appeal of this approach is that, in theory, you can add more processors to such a system as computational workloads increase. The difficulty is that conventional microprocessor technology makes little, if any, provision for the fundamental, requirement of parallel processing: a mechanism for interprocess communications.

The INMOS Transputer was designed for parallel processing; facilities for interprocess communications are embedded in the chip's silicon. In addition to a 32-bit microprocessor, a dynamic RAM (DRAM) memory controller, and 2K bytes of 50nanosecond on-chip RAM, the Transputer implements four high-speed direct-memory-access engines-dubbed "links" by INMOS-for serial data communication with neighboring Transputers. Transputer links comprise two unidirectional channels each. The links can transfer data in both directions at a rate of up to 20 megabits per second. The four hardware links and the microprocessor can independently access memory simultaneously; this is accomplished with minimal loss in processor throughput.

The links implement a communication protocol in hardware. The Transputer communicates by sending data bytes down the output channel of a link, framing each data byte with a start and stop bit. Once it has sent a data byte, the Transputer waits until it receives an acknowledge message on the input channel of the link. Programmers have access to these hardware links via instructions in the instruction set of the Transputer's microprocessor.

The Transputer supports multitasking in hardware, but it is more instructive, with respect to the Transputer's design, to regard multitasking as virtual concurrency. In a Transputer network, parallel processes may be physically concurrent (running on separate Transputers) or virtually concurrent (multitasking on the same Transputer). In practice, physically concurrent and virtually concurrent processes most likely will be running on a Transputer network.

The Transputer's multitasking capability makes it possible for you to write a concurrent program for a network of Transputers even when the number of computing nodes in the network is unknown or may vary. You could design a program embodying, say, twelve parallel processes to function the same on a single Transputer as it would on a network of two, three, four, or twelve Transputers. Programs designed in this manner can gain an almost linear increase in execution speed as you add Transputers to the network.

TCC, A Parallel C Compiler

TCC is a superset of the Kernighan and Ritchie definition of C that incorporates extensions to support concurrent programming on a network of Transputers. The TCC package consists of a compiler, an assembler, a linker, and a run-time library.

The TCC's run-time library closely follows the emerging ANSI run-time standard. It accomplishes access to the host PC's file system via level-two file I/O functions, such as fopen( ), fread( ), fwrite( ), getc( ), and so on. Functions for low-level access to the host (e.g., bdos( ) and sysint( )) and functions for sending and receiving message packets supplement the standard run time.

You can control all the Transputer's devices directly, including those that support concurrency, by the MPU's microcode. The Transputer's instruction set has instructions to start up processes, to start up interprocess communications, to select a process's priority, and to cause a process to wait on an event., Standard high-level languages (HLLs), including C, Pascal, and FORTRAN, lack any formal constructs that correspond to these operations. Consequently, conventional language implementations would never access the Transputer's concurrent programming resources.

To get at the Transputer's parallel processing capabilities, you have a few options. You could provide a subroutine or function library to extend the language. Another possibility is a distributed operating system or kernel that manages all the parallel aspects of a system through system calls. The option we took in the development of TCC was to extend the definition of the C language. These extensions include the channel data type, the par construct, the alt construct, and the pseudo-variable timer.

Extending the language definition has the advantage of efficiency, since the compiler produces in-line code (instead of a function or system call) whenever a programmer uses one of these parallel constructs. The efficiency gained by this approach is similar to that gained by a compiler that generates in-line code for a floating-point unit rather than calling floating-point functions.

Parallel Extensions

Parallel extensions include the following:

o The channel *data type and interprocess communications.* TCC's channel data type supplies the mechanism for synchronized process-to-process communications. The programmer uses channel variables to send data entities between two processes. These processes may reside on the same Transputer or on two different Transputers connected by a link. The channel data type conducts interprocess communications with equal facility in either instance. Channel variables behave in a manner that is consistent with the C language. They accept the cast operator, the sizeof operator, and type checking.

C's assignment statement is used with channel variables to send and receive data between two processes. When data is "assigned" to a channel variable within a process, the process attempts to send the data; conversely, when a channel variable assigns its contents to a data variable, the process attempts to read data via the channel into the data variable. Communication is synchronized; it occurs only when two processes become ready to communicate via a shared channel. The Transputer automatically deschedules a process that is waiting to communicate.

The following code fragment illustrates how a process might send a message to a host system:

```
static char *msg = "hello world!";
char *ptr = msg;
channel *host = (channel*)Ox80OOOOOO; /*
           host link output */

while(*ptr)
  *host = *ptr++;
```

Whereas this code fragment illustrates how a program might send a character string down a link, the channel data type supports the transmission of a variety of data types, including floats, integers, arrays, and structures. Here is an example using non-pointer channel variables:

```
channel Comm0l;

CalcOl() {
       double result;
       for(;;) (
             .. code ....  /* a nasty
             calculation here */
       Comm0l = result;
       }
}

IOProc0l() {
       double    result;
       for(;;) {
       result = Comm0l;
       printf(“result: %g\n”, result);
       }
}
```

In this example, `Calc0l` and `IOProc0l` run as concurrent processes. `IOProc0l` waits to receive result over the channel `Comm0l`. Upon receipt of `result`, `IOProc0l` displays the value on standard output; `CalcOl` is released to start its next calculation. Pending input from `Calc0l`, the Transputer deschedules `IOProc0l` so that microprocessor unit (MPU) cycles are not wasted on an idle process.

o The par construct. This starts up (spawns) processes on a Transputer. It resembles a compound statement in C:

```
void proc0l(), proc02(), procO3();
     .... code ....

     par {
          proc01();
          proc02();
          proc03();
     }

     .... code ....
```

This example starts three separate processes to call each of three functions. The processes terminate when the functions return. The parent process waits for the three processes to terminate and then resumes execution with the code following the par statement. Processes started with the par construct each have their own stack (or "workspace, " as it is known in Transputer nomenclature). Workspace requirements are calculated by the compiler at compile time and dynamically allocated at run time.

o The timer pseudo-variable. The timer pseudo-variable provides the programmer with access to the Transputer's on-chip timer. By means of this pseudo-variable, you can read the hardware timer or write to it almost as if you were reading or assigning a value from an integer variable:

```
  #include <time.h>
       (int) timer = 0;
  .... code .... /* your favorite
       benchmark here */

       printf("ELAPSED TIME IN SECONDS "
               "%ld\n", timer/CLK-TCK);
```

In this example, the assignment to timer is prefixed with an integer cast operator. The cast operator informs the compiler to initialize the hardware timer only. You can deschedule a process for a specific interval with the use of the timer pseudo-variable:

```
  timer += 1000; /* sleep for a thousand
       clock ticks */
```

o The alt construct. The alt construct provides a software mechanism whereby a process may arbitrate between events. With the alt construct, a process can test the readiness of any of several events, selecting the first event to become ready. An event may be a ready channel, a ready timer, or a Boolean true condition. Each component of an alt construct (i.e., each alternative) uses the keyword guard.

Its syntax and function are not unlike the C switch statement:

```
typedef Boolean int;
channel Comm01, Comm02;
int Result;
Boolean NoTimeOut = FALSE;

.... code ....

TimeOutBegin:

alt
{
  /* boolean event */
  guard NoTimeOut: break;

  /* channel event */
  guard &Comm01: Result = Comm01;
       break;

  /* channel event */
  guard &Comm02: Result = Comm02;
       break;

  /* timer event */
  guard timer += 1000; break;

  /* no event ready just yet */
  default: goto TimeOutBegin;
}
```

This code fragment illustrates how you can use the alt construct to multiplex channels and to time-out a process in the event of a communications failure. This statement evaluates three alteratives: It checks for the readiness of the Boolean flag Timeout, the readiness of channels Comm0l and Comm02, or the readiness of the timer. Alternatives are evaluated in the same order as they are written: from top to bottom. Thus, if two events should become ready simultaneously, the first in order is selected. The first alternative tests the status of the Boolean variable NoTimeout. If true, the alternative is skipped altogether. The next two alternatives check for pending input on one of two channels. Should either channel become ready, an input is performed and the alt statement terminates. (In this instance, input from two processes are multiplexed; in an actual application, an I/O routing process might funnel the output of a number of processes into a single channel.) The third alternative will cause a timeout if either of the two channels should fail to become ready within a specific time interval. Our example merely breaks out of the alt statement on a time-out; you could use this alternative to initiate a recovery strategy or to print an error message.

o The #pragma *macro preprocessor directive.* TCC provides a number of #pragma preprocessor directives to give the programmer greater control over program execution. These directives include- #pragma par, #pragma seq, #pragma fast, #pragma slow, and #pragma stack.

The par and seq directives allow you to control the compilation mode of a C module. You can also select either mode from the command line upon compiler invocation. In the parallel compilation mode, functions call a workspace allocator to a reserve space for local variables. This dynamic allocation of process workspaces ensures that recursion is possible even when multiple processes are executing concurrently. In sequential compilation mode, a single workspace is allocated at program startup. This workspace behaves identically to a stack on a conventional microprocessor. You use the sequential mode when a Transputer program consists of a single process, obviating the overhead of dynamic workspace allocation. You can use sequential mode when single-process programs are run on multiple Transputers.

The stack directive instructs the compiler to reserve a specific amount of memory for a given function or set of functions. This directive takes a hexadecimal value as an argument. It typically prefaces a function definition:

```
# pragma stack 8000

main(argc, argv)
int argc;
char *argv[];
{
  .... code ....
}
```

The fast and slow directives are used in conjunction with the parallel compilation mode. They provide control over the workspace allocator. Functions prefaced by the fast directive will first try to obtain their workspaces from the Transputer's on-chip RAM, while functions prefaced by the slow directive will receive their workspaces from external memory.

Run-time Support for Parallel Processing
The TCC run time provides additional support for the Transputer's parallel processing capabilities, particularly in instances where the compiler's language extensions are ill suited.

The functions msgsend( ) and msgrcv( ) send and receive data packets of arbitrary lengths across channels. Channel assignments work only with data entities whose size is known to the compiler. Consequently, you will often use these functions to send and receive buffers:

```
channel Comm01;
Proc01()
{
  static char *msg = “hello world!\n”;

  Comm01 = strlen(msg);
  msgsend(&Comm01, msg, strlen(msg));
}

Proc02()
{
  char str[MAXSTRLEN];
  int length;

  length = Comm01;
  msgrcv(str, &Comm01, length);
  puts(str);
}
```

The function startp( ) takes a function address as a parameter and starts the function as a separate process. A programmer can use startp( ), albeit with caution, to run multiple processes in a program compiled in sequential mode.

The function resetch( ) initializes a channel variable, taking the address of a channel variable as a parameter. (You must initialize channel variables before you can use them.)

The two functions stpri ( ) and ldpri ( ) pertain to process priority levels, of which, on the Transputer, there are two: high and low. The function stpri( ) sets the priority status of the process that calls it; ldpri( ) returns the priority status of the calling process. A high-priority process will always execute in preference to a low-priority process. However, the Transputer will deschedule a high priority if that process is waiting to communicate or is waiting for a timer to 3ecome re Transputer will then grant time slices to any extent low-priority processes. Should the high-priority process become ready to run again, the low-priority processes will be interrupted.

Programming a Transputer Network with TCC

On a single Transputer, we can start up a number of processes using TCC's par construct or the run-time startp ( ) function. Running multiple processes on a single Transputer compares to the kind of multitasking that operating systems such as Unix perform. In this case, we do not have true concurrency but, rather, virtual concurrency. You can achieve true concurrency on a single Transputer with programs that make use of channel I/O. In such a case, it is possible to employ one or more of the Transputer's links even as the MPU is executing code.

The link facilities of the Transputer make it possible for you to use various schemes of interconnection between multiple Transputers in a network. These interconnection schemes are known as "topologies."

Topology types include two-dimensional arrays, systolic arrays, hypercubes, and trees. Strategies in implementing or choosing a Transputer topology may involve minimizing the distance of link paths between Transputer nodes for efficiency and redundancy of ha paths for reliability. In the programming example that follows, we chose to make a daisy chain of Transputer nodes; this topology has the virtue of simplicity.

Concurrency with Multiple Transputers

Two expedient techniques for concurrent programming on a Transputer network are pipelining and the FARM architecture.

Pipelining sets up stages of a program, with each stage lodged on a separate Transputer. The first stage sends its output to the second stage, the second to the third, and so on. A compiler is a good example of a program that stands to benefit from pipelining. A pipelined compiler, running on a Transputer network, might run its preprocessor in the first stage, the lexical analyzer and parser in the second stage, the code generator in the third, and an output process, which resolves a binary image to a specific link format, in the fourth. Pipelining transforms a program from a single sequential process with multiple phases to an ordered set of concurrent processes. In addition to the performance benefit, a pipelined program imposes the kind of modular design on a program that lends itself to ease of maintenance and team development efforts.

The FARM architecture is implemented by identifying one or more points in a program where a calculation task iterates through a wide body of data. These calculation tasks are then coded as individual processes and replicated on each node in a Transputer network. Typically, a message passer process or processes are replicated on the network along with these calculation processes.

The following represents a simple form for a calculation process on a FARM:

```
1. Wait for input on a given channel or
   link.
2. Input the data.
3. Perform the calculation on the data.
4. Wait for output on a given channel.
5. Output the data.
6. Repeat.
```

The program TDHRY.C is a version of the well-known Dhrystone benchmark modified to run in parallel on a Transputer network. [Editor's note: *TDHRY C is available for uploading from BYTEnet and BIX. The author has also provided another example program-PWC. C, a word-counting program-for uploading. It and its documentation, PWC. DOC, are available on BYTEnet and BIX.]* It uses a rudimentary FARM architecture. Underpinning the TDHRY.C program is a network hooter, NB.LIB, or NB for short. NB performs the fundamental task of booting TDHRY.C on multiple Transputers. The operation of NB, as shown in the list below, is fairly straightforward:

1. A prebooter reads NB into Transputer memory.

2. NB reconstructs a clone image of itself and the prebooter in Transputer memory.

3. Each of the nonboot links is tested for the presence of a Transputer on the other end.

4. The status of each of the four links, including the boot link, is recorded in an array.

5. NB uploads a copy of itself (its clone image) to each of the active non-boot links.

6. NB begins to read in the program proper. As it reads in the program, copies are sent to all the active non-boot links (those that received a copy of NB). Any requisite code relocation, data initiation, and so on, is done at this time.

7. NB reads in the command-line argument count and the argument vector from its boot link. To each of the active non-boot links NB sends out a null argument count. (Thus, only the program on the root Transputer has a non-null argument count.)

8. NB sets up a stack for the program and calls main().

NB acts as a virus, spreading itself and a program throughout a Transputer network. The original copy of the program (that lodged on the root Transputer) differs from its clones in one respect: All clones have a null argument count, while the root program has an argument count of at least one (for argv[0]). argv[0], of course, is a pointer to the program's name. The clone copies of the program are (fittingly) nameless.

NB is limited to a daisy-chain topology; it assumes only one connection between any two Transputers. For other topologies, we would have to incorporate additional logic into NB; namely, NB would have to have a strategy to recognize an active link that has already been booted by another Transputer.

With NB in place, the modifications to the Dhrystone benchmark are slight. However, it is necessary to code in different logic paths that depend on whether the program is a clone copy or the original copy on the root Transputer. The root Transputer is charged with displaying the benchmark results on the console, since it is the only copy of the program with direct access to the host system.

The following piece of code is the initialization code at the start of main():

```
main(argc, argv, LinkArray)
int argc;
char *argv[];
channel *LinkArray[];
{
  int i, NoOfTxxs = 1;

  /* if root copy of DHRY.c */
  if (argc > 1)
  {
    /* get # of transputers */
    NoOfTxxs = atoi(argv[1]);
  }

  /* if clone */
  if (argc == 0)
  {
    /* get # of TXXs */
    NoOfTxxs = *_fromhost;
  }

  for (i = 0; i < 4; i++)
  {
    channel *ChanPtr;

    if ((ChanPtr = LinkArray[i]) != 0)
      *ChanPtr = NoOfTxxs;
  }

  Proc0(argc, NoOfTxxs, LinkArray);
  Exit(0);
}
```

The root copy of the program determines, from a value passed on the command line, the number of Transputers in the network. Clone copies, lacking command- arguments, must determine the number of Transputers by fetching it from their boot link. Then for each active link, the program-be it a clone or the root copy-passes along NoOfTxxs. Eventually, all copies of TDHRY.C executing in the network are informed of the total number of Transputers in the network.

The next point of interest in TDHRY.C occurs in Proc0():

```
loops = LOOPS/NoOfTxxs; /* adjust
         iteration count */
```

Here, the constant LOOPS is replaced by the variable loops, which is equivalent to LOOPS divided by the total number of Transputers in the network. The main iteration loop of the benchmark replaces the constant LOOP with the variable. Thus, a single Transputer running the benchmark will iterate to the full value of LOOP; two Transputers running the benchmark will each iterate to LOOP / 2; four Transputers, to LOOP / 4; and so on . In any case, the number of total iterations performed on the network will equal LOOP, and the results, save the overall execution time, will be the same.

Once the main iteration loop of the program completes, each copy of the program on the network calculates its execution time, collects the execution times of any Transputers it might have itself booted, and forwards this value to the Transputer that booted it. Eventually, the aggregate execution time is collected by the root copy of the program and displayed on the console:

```
if (NoOfTxxs > 1)
{
  for (i = 0; i < 4; i++)
  {
    channel *ChanPtr;
    if (ChanPtr = LinkArray[i]) != 0)
    {
      ChanPtr = ChanPtr&0x10;
      benchtime += *ChanPtr;
    }
  }
}

if (IsRoot)
{
  printf(“Dhrystone(%s) time for “
         “%ld passes = %ld\r\t”,
         version, (long) LOOPS,
         benchtime);

  printf(“This machine benchmarks at “
         “%ld dhrystones/second\r\n”,
         ((long) LOOPS) / benchtime);
}
else
{
  *_tohost = benchtime;
}
```

Theoretically, the parallel Dhrystone should yield almost linear increases in performance as Transputers are added to the network. Communications overhead will, of course, preclude the possibility of strict linear increases. Running TDHRY.C on the TG2 board confirms the theory: On a single 20-MHz Transputer, TDHRY.C achieves about 4500 Dhrystones. Two Transputers boost the execution time to about 8600 Dhrystones-not quite a factor of 2, but close.

Conclusion

The INMOS Transputer's on-chip support for parallel processing make it an efficient parallel-processing engine. The TG2, TCC, and the TMAC macro assembler combined form a cost-effective parallel-processing development system for PC users.

TCC defines extensions to the C language that a high-level language needs to harness the potential of the Transputer in a multiprocessor network. TMAC offers similar capabilities for assembly language programmer. With these tools, software developers can design programs for Transputer, networks of any size and with almost unlimited performance potential.

ACKNOWLEDGMENT'S

The authors of this article wish to acknowledge Trevor Marshall, Ph.D., Christopher Jones, Bernt Roelost, Hans Bieleman, Klaas Wijbrans, Merik Voswinkel, Professor Andre Bakkers, the Twente University of Technology, and Vince Williams, Ph. D., for their advice and support.

BIBLIOGRAPHY

Karplus, Walter J., ed. *Multiprocessors and Array Processors. San* Diego, CA: Simulation Councils Inc., 1987.

Roscoe, A. W., and C.A. R,. Hoare. *The Laws of OCCAM" Programming*. Oxford, England: Oxford University Computing Laboratory Programming Research Group, February 1986.

Shepherd, Roger,. "Extraordinary Use of Transputer Links," INMOS Technical Note 1. Bristol, England: INMOS Ltd., 1986.

"The Transputer Instruction Set-A Compiler Writer's Guide." Bristol, England: INMOS Ltd., May 1987.

Transputer Reference Manual. Bristol, England: INMOS Ltd., January 1987.

The TG2 Microprocessor Board

The TG2 uses two 15-MHz INMOS T414 Transputers. Each Transputer can address I megabyte of supplied DRAM in addition to its 2K bytes of internal memory. You can upgrade the second section of the TG2 to accommodate 4 megabytes of DRAM.

The graphics section provides a bitmapped display of up to 512 by 512 pixels with 24 bits of color per pixel. It can display over 16 million colors simultaneously on low-cost multisync monitors. The display system operates independently of the host PC's display. The on-board Texas Instruments 34061 video RAM controller allows complete programmability of the displayed pixel count and the sync rates (e.g., for PAL operation).

The Transputer board communicates with the host PC for console I/O and other system service requests via a Transputer hardware link. To ease the programming task, the TG2's host interface causes the host to appear as though it were another 'Transputer on the network.

The TG2 provides a row of pin headers for interconnection of Transputer links, affording the developer complete control over network topologies. You can arrange the Transputers in two dimensional grids, binary trees, hypercubes, or systolic arrays. You can network and run multiple TG2s from the host PC, and larger Transputer networks are possible with the use of a PC expansion chassis. The size of the Transputer network is limited only by the number of available expansion slots and power supplies.

You can also use the TG2 with Definicon's T4 multiprocessor board. The T4 omits the graphics section to provide four Transputer sections on a single PC expansion bus card.

A Transputer Macro Assembler
Definicon supplies each TG2 with a full-featured macro assembler, TMAC, to support Transputer program development at the assembly language level. TMAC recognizes the complete instruction set, as published by INMOS, for the T414 and provides extensive pseudo-op codes to facilitate the programming task. These pseudo-op codes allow the programmer to define bytes, words, macros, and constants; to align code and data in memory; to toggle listings; and to include files.

Product Information

Definicon Systems will provide the DSITG2 Transputer graphics board and the TCC development system at the following special prices for BYTE readers. Definicon has not allowed any margin for accounting overhead, so no purchase orders can be accepted for these products. At these prices, documentation is supplied on floppy disk. (You can order a printed copy of the documentation at additional cost.)

To order, contact Definicon Systems (1 100 Business Center Circle, Newbury Park CA 90320, (805) 499-0652). Terms of payment are Visa, MasterCard, or American Express only. There is a 30-day, no-questions-asked, moneyback guarantee. Goods must be returned in "as new" condition in original packaging for full credit.

Software support available is limited to diagnosis and correction of your software problem, The TG2 board has been tested in machines compatible with the EBM XT and AT. If it doesn't work in yours, Definicon reserves the right to either correct the problem (if it pertains to the DSI-TG2) or refund your money.

The DSI-TG2 will operate in a system with floppy disks, but you'll need a hard disk for meaningful program development.

The DSI-TG2 with a single Transputer section costs $945. One Transputer section consists of I megabyte of dynamic RAM and one 15-MHz T414; a host interface section; MS-DOS interface software; and macro assembler-assembled and tested.

The DSI-TG2 with two Transputer sections costs $1595. Two Transputer sections consist of I megabyte of DRAM and one 15-MHz T414 per section; a host interface section; MS-DOS interface software; and macro assembler-assembled and tested.

The DSI-TG2 with two Transputer sections and graphics costs $1995. Two Transputer sections consist of I megabyte of DRAM and one 15-MHz T414 per section; one high-definition graphics section (512 by 512 pixels, 24 bits per pixel); multisync monitor output; a host interface section; MS-DOS interface software; and macro assembler-assembled and tested.

The TCC "Parallel" Development System consists of a TCC compiler, assembler, and linker for 395. The Kernighan and Ritchie definition C compiler has extensions for parallel programming, in addition to many Unix and ANSI extensions.

TG2 and TCC documentation costs $35 and includes typeset, printed material for the DSI-TG2 Transputer board; TMAC, the Transputer macro assembler; and the TCC Development System.