Published in: Proceedings of the 8th International Parallel Processing Symposium, IPPS'94, April 26–29, 1994, Cancun, Mexico, pages 260–267.

Parallel Evaluation of a Parallel Architecture by means of Calibrated Emulation

Henk L. Muller* Paul W.A. Stallard* David H.D. Warren* Sanjay Raina†

Department of Computer Science, University of Bristol, Queen's Building, Bristol. BS8 1TR, UK.

Abstract

A parallel transputer-based emulator has been developed to evaluate the DDM—a highly parallel virtual shared memory architecture. The emulator provides performance results of a hardware implementation of the DDM using a calibrated virtual clock. Unlike the virtual clock of a simulator, the emulator clock is bound to a fixed fraction of real time so individual processors may time actions independently without the need for a global clock value. Each component of the emulator is artificially slowed down so that the balance of the speeds of all components reflects the balance of the expected hardware implementation.

The calibrated emulator runs an order of magnitude faster than a simulator (the application program is executed directly and there is no overhead for the maintenance of event lists) and more importantly, the emulator is ***inherently parallel****. This results in a peak emulation speed of 16 million instructions per second when simulating a machine with 81 leaf nodes on a 121 node transputer system.*

1 Introduction

Shared memory machines are convenient for programming but do not scale beyond tens of processors. The Data Diffusion Machine (DDM) [1], overcomes this problem by providing a shared memory abstraction on top of a distributed memory machine. A DDM appears to the user as a conventional shared memory machine but is implemented using a distributed memory architecture. This approach is generally known as Virtual Shared Memory, or VSM.

The key issues for evaluating the merits of the DDM are the complexity of the design, the performance, and the scalability of the performance. In this paper, we focus on performance evaluation of the machine design. In general there are several ways to evaluate the performance: using a performance model, a simulator, or a prototype of the architecture. In this paper we use a combination of the last two: a prototype of the machine developed in software (an emulator), enriched with a virtual clock to provide realistic timings, an approach similar to the WWT [2]. The emulator thus provides the complete functionality of a hardware prototype. Although it runs slower than a prototype, it is much cheaper to produce and gives the flexibility to experiment with the architecture. Compared with a simulator, the emulator is slightly less flexible but it runs significantly faster, and in parallel.

The rest of this introduction gives some background on the DDM. In Section 2 we address the evaluation strategy. An emulator implements both the functionality and the timing aspects of a future DDM realisation, and can be used to run benchmark programs, measure their performance, and study various architecture details. The emulator has been validated as is described in Section 3 in which we show that the results of the emulator have reasonable correspondence with figures from existing machines. In Section 4 we show some preliminary results of emulating the DDM, and of the emulator performance.

1.1 Data Diffusion Machine background

The purpose of the DDM architecture is to provide a scalable shared memory architecture to the user. The DDM is not the only architecture that provides virtual shared memory. Other machines such as the DASH [3], KSR-1 [4] and MIT-Alewife [5] also implement virtual shared memory. There are however some essential differences between these machines. The DDM and KSR-1 are described in the literature as cache only memory architectures or COMAs. A data item is stored at the processors where it is needed and does not have a home location where it will always be found. Instead data is located by means of directories. Data addresses no longer correspond to physical locations, but simply represent names (or tags) for data. As the data is free to move around the system the programmer sees a uniform memory access (UMA) system in which all data is equally accessible. In contrast, the DASH and Alewife architectures have the data stored in some home location and cache the data where it is frequently used. In these architectures memory accesses are non-uniform (NUMA) which

*Working at PACT/SRF, 10 Priory Road, Bristol. BS8 1TU, UK.
e-mail: {henkm,paul,warren}@pact.srf.ac.uk.
Work supported by ESPRIT P7249, OMI/HORN.
†Seconded from Meiko Limited, Aztec West, Bristol.

generally means that the programmer is more restricted in how data is laid out in the address space.

The DDM supports a strong consistency model. Like the KSR-1 and the MIT Alewife, the DDM offers the same form of consistency as a machine equipped with a single memory. The DASH provides a weak form of consistency known as release consistency in which the memory is only known to be consistent after explicit synchronisation points. In general, a stronger consistency model complicates the implementation, but simplifies programming.

The DDM is organised hierarchically. The leaves of the hierarchy consist of processors with large set-associative memories that comprise the sole store for data. The nodes above the leaves are directories that keep track of the data items below. Data can be either writable in the memory of exactly one processor, or read-only in multiple memories. When the data required by a processor is not available locally, a request is posted upwards in the tree. The request is propagated upwards until a directory indicates that the data is available in some branch of the tree below. This directory then fetches the data and returns it to the requesting node. If necessary, ordinary caches can be placed between the processor and the DDM memory, or local memory can be attached to nodes to store, for example, program code.

The DDM architecture can be built with many types of interconnect. At SICS in Sweden a DDM is prototyped with a hierarchy of busses [6]. We are particularly interested in a DDM constructed with a point-to-point network, as supported by transputers for example. The protocol used is different, but the basic properties (the latency in all operations scales only with the logarithm of the number of nodes) are identical. The version of the architecture that is described in this paper consists of a single tree, that is interconnected with links.

2 The emulator

To evaluate the performance of the DDM architecture, a software emulator has been developed. The first emulator was described in [7]. The emulator presented in this paper employs a more powerful calibration scheme, giving rise to more reliable results, and more flexibility.

Software emulation has a number of advantages over other strategies, such as simulation or measurements on a prototype hardware implementation. A hardware implementation is expensive, inflexible and takes considerable time to develop whereas a software emulator can provide a flexible experimentation platform as well as accurate results. The emulator runs orders of magnitude faster than a detailed simulator and provides results of comparable accuracy. The emulator itself runs on a parallel machine with near perfect speedup, as opposed to the far from perfect speedup resulting from parallelising simulators. As a consequence, the emulator is capable of simulating real applications with a speed of over 300,000 emulated processor instructions per second per node.

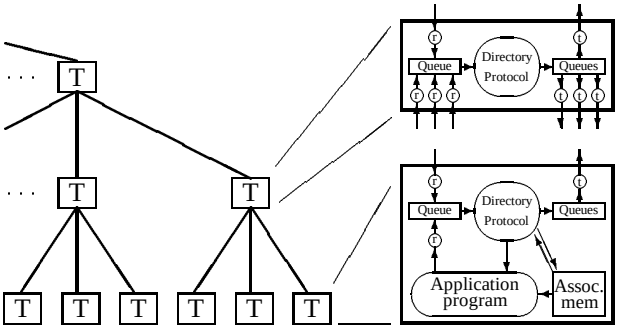

Figure 1: The DDM emulator. The boxes marked 'T' are transputers, the right hand side shows which processes run on each transputer.

The emulator is presented in two stages. In Section 2.1 the process of developing a functionally correct emulator is presented. The calibration scheme employed in the emulator, that causes the emulator to have timing characteristics that are identical to an expected hardware implementation, is presented in Section 2.2.

2.1 Emulation

The basic structure of the emulator is shown in Figure 1. A network of T800 transputers [8] is used for the emulation. The lowest level of transputers emulates both the parallel application program, and the leaf nodes of the DDM hierarchy—the set-associative memory nodes. The transputers higher in the hierarchy emulate the DDM directories.

The right hand side of Figure 1 shows the processes needed for the emulation in more detail. Each interior transputer (in this particular example with a fanout of 3) executes 9 processes: one process implementing the DDM-directory management, four receiving transactions from above and below, and four that are transmitting transactions. The receivers and transmitters decouple the protocol engines of the various nodes, which is necessary since the protocol works with asynchronous transactions while a transputer supports synchronous communication. The communication between the directory and receiving and transmitting processes is implemented by means of shared queues.

The process structure on the leaf-transputers is similar, but these transputers additionally execute the application program as a separate process. This process communicates with the directory process and the (single) receiver process via soft-channels. The data structure containing the set-associative memory is maintained by the directory but

accessible to the application program. The structure of the top node is also slightly different, because there is no level above. Since no transaction should ever try to leave the DDM, a special process is connected to the top directory that generates a fatal error if a transaction emerges from the top-level directory.

The directory processes on each node do the bookkeeping necessary for the DDM architecture. When a transaction enters the directory process, the address carried by the transaction is indexed and matched with the keys of the data items, and when a match is found, a state machine implements the data diffusion coherency protocol. The process also maintains statistics (described in [9]) that are required to evaluate the architecture and applications.

The number of receiving processes at each level depends on the number of nodes in an architecture but can never exceed four—the number of physical links of a transputer. This limits the architectures to be emulated to a fanout of three for the interior nodes, and four for the top node. We plan to remove this limitation in the near future.

The application program is compiled to the transputers. It is thus not simulated, but actually executed on the leaf nodes. To emulate the virtual shared memory each shared memory reference must be trapped and directed through the protocol handler to ensure the data is available locally and in the correct state before the operation is performed. As the transputer does not provide memory management, these references cannot be trapped by hardware so must be intercepted in software. To achieve this the code is annotated after the compilation stage (as in Tango [10] and MiG [11]).

After the compiler has generated transputer assembler, the assembly code is modified so that each non-local read and write operation is replaced by a function call to the protocol handler. The protocol handler first tests if the trapped reference is indeed a shared access and then either returns the data (in the case of an access to local data) or passes the reference on to the protocol engine. The non-local accesses are references to potentially shared data whereas local accesses are in the local stack of a node. In the current emulator we do not allow process migration (in which running processes could move from processor to processor depending on load and data locality) so that a private stack for each processor suffices.

The code annotation process is completely transparent to the user. The user simply compiles the application program with `dcc`—a compiler driver analogous to `cc` which performs the additional code translations after the compilation stage. After linking, the program is executed under the control of the DDM emulator. We provide support for the p4 programming model [12]—a portable library providing parallel programming facilities (such as synchronisation, process control and timing). By implementing a p4 library for the DDM, we are able to compile and run existing p4-based parallel programs on the emulator without modification.

Component	T800 Emulation	Projected Hardware	Ratio
Integer Instruction	50–300 ns	10 ns	5–30
Floating Point	.4–1.6 μs	40 ns	10–40
Local Memory time	33.8 μs	100 ns	338
Protocol Delay	81.0 μs	200 ns	405
Header Latency	24.0 μs	400 ns	60
Message Latency	86.0 μs	1040 ns	83

Table 1: DDM Component Timings of the uncalibrated emulator. The projected hardware column shows the timings expected to be achievable before the end of this decade.

2.2 Calibration

The emulator presented above represents a functional model of the DDM, working exactly like a real DDM except that it is comparatively slow. Unfortunately, it is not simply some factor slower than a hardware implementation because the balance of speeds of the various components are very different from those of a real DDM. This imbalance is illustrated in Table 1. The ratio of speeds of the various components differ widely: the speed of the network is only 60 times slower than the expected implementation, while the protocol engine is 405 times slower.

The imbalance extends to the speed of the T800 itself and its interconnection network—the T800 transputers used in the emulator are much slower than the processors that will be available by the end of this decade. All these differences make the temporal performance of the emulator of little use as a comparison with a hardware implementation. As the scalability of the architecture depends crucially on the actual timing parameters of the system, we have devised a method to make the emulator reflect the hardware timings more accurately.

Our aim is to map the timing model of the DDM emulator on to the model of the timings of the expected DDM implementation. This mapping is achieved using a technique called calibration and results in a system with very similar temporal properties to a real implementation. As we cannot be sure exactly what the future hardware specification will be, the timings are a parameter of the calibration process. The mapping of the functional model to the hardware model is shown schematically in Figure 2. The execution trace of a program accessing data that currently resides on another

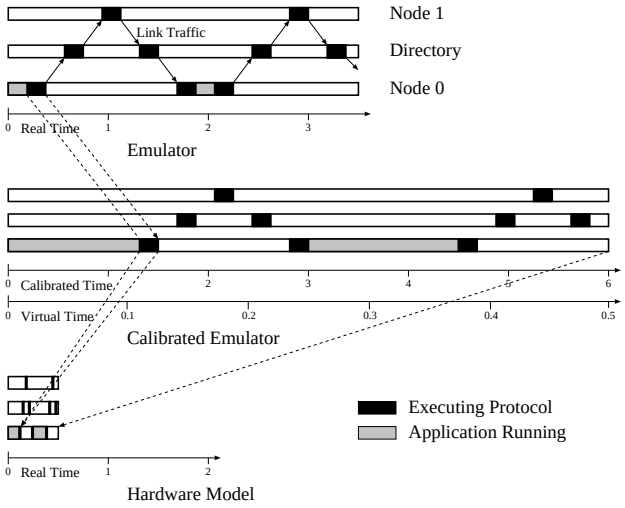

Figure 2: The Calibration Process.

node is shown in the top three lines (denoting the activity of the two nodes involved and the directory in between). The calibration process increases the time spent in certain phases so that the temporal characteristics of the emulator are only a *constant* factor different from the characteristics of the hardware model. This leads to the execution paths shown in the middle three lines. By reducing the time scale we manage to get the execution trace as it would have been on the real DDM architecture, shown in the bottom lines.

The calibration scheme sketched above is implemented by using a *virtual clock*. The virtual clock is defined to run at a fraction of real-time dependent on the worst of the ratios between emulation speed and hardware speed. In the case of Table 1, the virtual clock would run at $1 / 405^{th}$ of real-time. Each element of the emulator monitors the virtual clock and after performing its task, waits for the appropriate virtual time to pass. The protocol in the above example will take 81 μs of real time to complete during which time the virtual clock will have advanced exactly 200 ns. A memory access however will take 33.8 μs of real-time which corresponds to only $33.8 / 405 = 83$ ns of virtual time. This code will therefore block until the whole operation has taken $100 \text{ ns} * 405 = 40.5\ \mu s$, so that the virtual clock will have advanced 100 ns as dictated by the hardware model. In contrast with the approach described in [2], the introduction of an artificial delay means that there is no need to maintain event lists.

To model various processor speeds, we have to adjust the execution speed of the emulation processor to the execution speed of the emulated processor. The calibration scheme is designed so that it adjusts for differences in the instruction set between the emulation processor (the T800) and the emulated processor (we are aiming at the successor of the T9000). Each instruction of the T800 is classified according to the expected execution speed on the target processor. Currently there are three classes of instructions: null-instructions, floating point instructions and all other operations. The null instructions are the T800 instructions which will be executed on the target processor in zero instruction cycles, or which are not required by it. The most prominent member of this class is the prefix instruction that the T800 needs to construct operands. Null instructions are not accounted for by the emulator. The floating point instructions comprise all instructions for performing floating point arithmetic. It is assumed that all these instructions are executed in one single "floating point cycle" (addition and division operations thus run at the same speed). The rest of the instructions (jumps, integer operations, loads, stores) are classified as integer operations. The emulator assumes that they execute in a single integer instruction cycle as long as no memory is accessed. The calibration scheme is flexible enough to extend the number of instruction classes, but we have found that a simple division into null, floating point and other instructions gives reasonable results. It is possible however, as more information on the target processor becomes available to formulate a more accurate model of those components.

The calibration for the processor is achieved by annotating the application assembler code from the C compiler and inserting time costs in the code at the end of each basic block. The number of instructions between jump/call instructions are counted and a delay of the appropriate time is inserted at the end of the block. At that point the program is descheduled until the value of the real time clock and the value of the virtual clock (which has been incremented because of the executed instructions) match again. In this way both the time used by the actual execution of the instructions, and the time that has been used by the protocol (if that had been scheduled in because there was some request from above) are accounted for.

The annotation with the timing calls is performed transparently by the same program that generates the traps for all accesses to shared memory. Using basic blocks as the unit of calibration overcomes the potential code explosion of annotating individual instructions. Because the calibration of long basic blocks at the end may result in less accurate results, the calibration is performed at regular intervals inside large basic blocks as well. Only 10% of code is added, while the errors in the calibration are negligible.

Unlike the interior nodes of the DDM hierarchy that run only the directory protocol engine, the leaf nodes run both the protocol engine and the application itself. The calibration mechanism ensures that the application appears to be the sole consumer of the leaf CPU and that the protocol engine that cohabits the node is also able to operate

unhindered. The calibration scheme decouples these two processes completely and ensures that each one operates seemingly on separate hardware at the speed dictated by the hardware model. This is achieved by deducting the time spent in the directory from the calibration of the time of the application. The calibration factor is increased to ensure that both the application and protocol have enough idle periods to allow each other to run. The extra slowdown depends on the slowdown factors, the instruction mix and the utilisations of processor and protocol but is at most a factor of two. This is an upper bound (as it effectively provides two processors) but in practice one can run with only 20% extra slowdown and a run time check that verifies that the clocks are able to catch up, saving a considerable amount of time.

At this stage we can only estimate the properties of future hardware. We have specified the hardware model in a separate file that allows us to define various machines and compare the DDM performance on these. This configuration file is read by the emulator at start-up time whereupon the emulator measures the speed of each of the components. It then calculates the calibration factors necessary to simulate the hardware model specified and distributes these to each processor. This ensures that the calibration is kept constantly in tune with any changes in the emulator.

The major advantage of the calibrated emulator (from now on referred to simply as the emulator) over an event driven simulator is the speed. The emulator can run in parallel on an arbitrary number of nodes without having to perform any synchronisation. The speed of the emulator however critically depends on the slowdown of each component: one component with a huge slowdown will lead to slow emulation. The lack of synchronisation makes the emulator non-deterministic and for this reason, the emulator has an option to maintain a log for debugging. Lastly, developing an emulator is a bit (but not much) harder than developing a discrete event simulator, an investment that is repaid when emulating large computations.

3 Validation

When an emulator (or simulator) is used to measure performance figures, it must first be validated. This validation will give an indication of the errors in the modelling process, invaluable when judging the merits of an architecture. A complete validation will compare the results of an emulator with the results of the existing machine. As we do not have real DDM hardware to compare with, we have performed three other validations. First the robustness of the emulator and its calibration scheme is verified. Using the calibration mechanism, the emulator can be used to mimic many architectures. Section 3.2 presents a validation against T800 and Sparc/10 uniprocessor systems and Section 3.3 compares the emulator against a shared memory multiprocessor. In general, the errors in the modelling are less than 10%. This is very acceptable as we are unable to predict the specification of future hardware within a margin of 10%.

Access					on	on	on	on
Link			on	on			on	on
Protocol		on		on		on		on
Time (μs)	80.9	79.1	78.9	77.6	78.3	78.5	78.3	78.2
Error (%)		2.2	2.5	4.1	3.2	3.0	3.2	3.3
Emulation	119	168	168	217	197	202	200	222

Table 2: Errors on the robustness test.

3.1 Robustness

The calibration process described in the previous section takes care of the timing of the emulator. This means that the timings should not depend on any delay in the emulator or the underlying hardware. To verify this, we have added delay loops at three places in the emulator, and compared the predicted performance with the performance of the emulator without the delay loops. One program of the test suite was used (Barnes 64) on a three level binary tree architecture topology.

The timings are shown in Table 2. The row labelled "Time" shows the execution time for 8 different settings of the emulator, without any delay loops, or with one or more of the delay loops enabled. The next row shows the error in the performance relative to the case without any delay loops, the last column shows the number of seconds needed for the emulation. Although the time needed by the emulator varies considerably (almost a factor of two), the error is somewhere between 2 and 4 percent, which is very acceptable. Notice that in contrast with a simulator the emulator is non-deterministic. The exact behaviour of the machine, or in some cases also the behaviour of the program, depends on the exact timings of the hardware. In the case of a race condition, the emulator might choose any of them at random. The variation of the figures is about 0.7% for this example.

3.2 Validation against uniprocessors

To test the completeness of the processor modelling, we have compared the emulator with two existing uniprocessor systems: the T800 and the Sparc/10. The emulator calibration files for the T800 and the Sparc/10 are set so that the net speed of instructions and memory correspond with our transputer and Sparc platforms. Four programs from the SPLASH suite [13] were tested: Water, Barnes, Mp3d and Pthor. Water and Barnes are heavily floating

Program	T800 Real	T800 Emu	T800 Error	Sparc 10 Real	Sparc 10 Emu	Sparc 10 Error
Water 64 7	31.8	33.7	6.0	1.65	1.87	13.3
Barnes 128	17.6	15.4	12.5	1.12	0.98	12.5
Mp3d 30000 10	19.6	18.1	7.7	1.76	1.77	0.6
Pthor 500	9.3	7.5	19.4	0.81	0.74	8.6

Table 3: Execution time in seconds and percentage error between the emulator and real world processors.

point (double precision) bound, Mp3d uses single precision floating point, and Pthor is mainly integer and pointer manipulation. The last program has been adjusted to remove a large number of system calls: the emulator allows the clock value to be read without any time overhead; on our UNIX system the clock references accounted for 50% of the total execution time. For the same reason the I/O calls were removed.

The run times as measured on the real hardware and the emulator are shown in Table 3. The errors between the real implementation and the hardware are in the order of 10% with an extreme up to 20%. Given the high level nature of the modelling that is used we consider these errors as quite acceptable.

3.3 Validation against a Sequent Symmetry

The Symmetry is based on 16 MHz 80386 processors that have a CISC architecture: each instruction requires around 4 clock cycles to execute. The instructions of a 386 are more powerful than the instructions of our emulated RISC processors so around half as many instructions are actually required. We have calibrated the DDM emulator to run with a clock cycle of 133 ns, which roughly compensates for both the cycles per instruction and the power of the instructions. Executing two instructions on the DDM takes 266 ns, equivalent to a single 386-instruction of 4 clock cycles on a 16 MHz 80386. The memory is set to respond in one cycle (133 ns), and the network is set to have negligible delay. The results of running Aurora (a parallel Prolog system [14]) on the DDM and on the Sequent are summarised in Table 4. The error is around 5%, indicating that the emulator gives good results in this case as well.

4 Experiments

The emulator has been used to run a number of experiments on a (hypothetical) DDM using a machine with 131 T800 transputers (allowing up to 81 leaf nodes). Each transputer is equipped with 4 MB of memory. 2 MB are used to emulate the DDM memory, 1.5 MB is reserved for

Nodes	Sequent Symmetry	Emulation	Error
2	25.56	27.14	5.8%
3	17.37	18.52	6.2%
4	13.53	14.33	5.6%
6	9.73	10.19	4.5%
8	7.77	8.04	3.4%
9	7.05	7.41	4.9%

Table 4: Execution times in seconds when running Aurora Hamilton on the Sequent and on the emulator.

Component	Proj. hware	Time	Slow down	Calibrated
Integer instruction	0.010	3.0	300	4.86
Floating point	0.040	3.0	75	19.4
Local memory access	0.100	33.8	338	48.6
Protocol transaction	0.200	81.0	**405**	97.2
Header latency	0.400	24.0	60	194.0
Message latency	1.040	86.0	83	505.4

Table 5: The calibration parameters and resulting timings in microseconds.

program code and local data and stack, and 0.5 MB for system and emulator code. The experiments were run on trees with up to 4 levels. A DDM with up to 4 leaf processors is simulated on a single level tree with 5 transputers, up to 12 nodes on a two level tree with 17 processors, up to 36 leaf processors on a three level tree with 53 transputers and a DDM with 81 leaf nodes on a system with 121 transputers organised in a four level tree.

We have measured one instance of the DDM, of which the characteristics are summarised in Table 5. These parameters are a first estimate of the specification of the processor and the network that we expect to use for the DDM. The table also shows the timings of the individual emulator components and the resulting slowdown factors. The annotation of the code (for calibration) makes the emulation of instructions slower than shown in Table 1. The limiting factor is the protocol implementation, all other components run faster. According to the calibration strategy, the whole machine is slowed down to run a factor 486 (405 plus the 20% explained in Section 2.2) slower than real time, leading to the timings shown in the fourth column. Note that this is a factor of 486 slower than the emulated machine, not than the emulating processor: in comparison with the T800, the emulator runs only 20–30 times slower.

We have run the SPLASH benchmark suite [13] and the Aurora [14] and Andorra-I [15] parallel Prolog systems on our emulator. The SPLASH benchmark programs

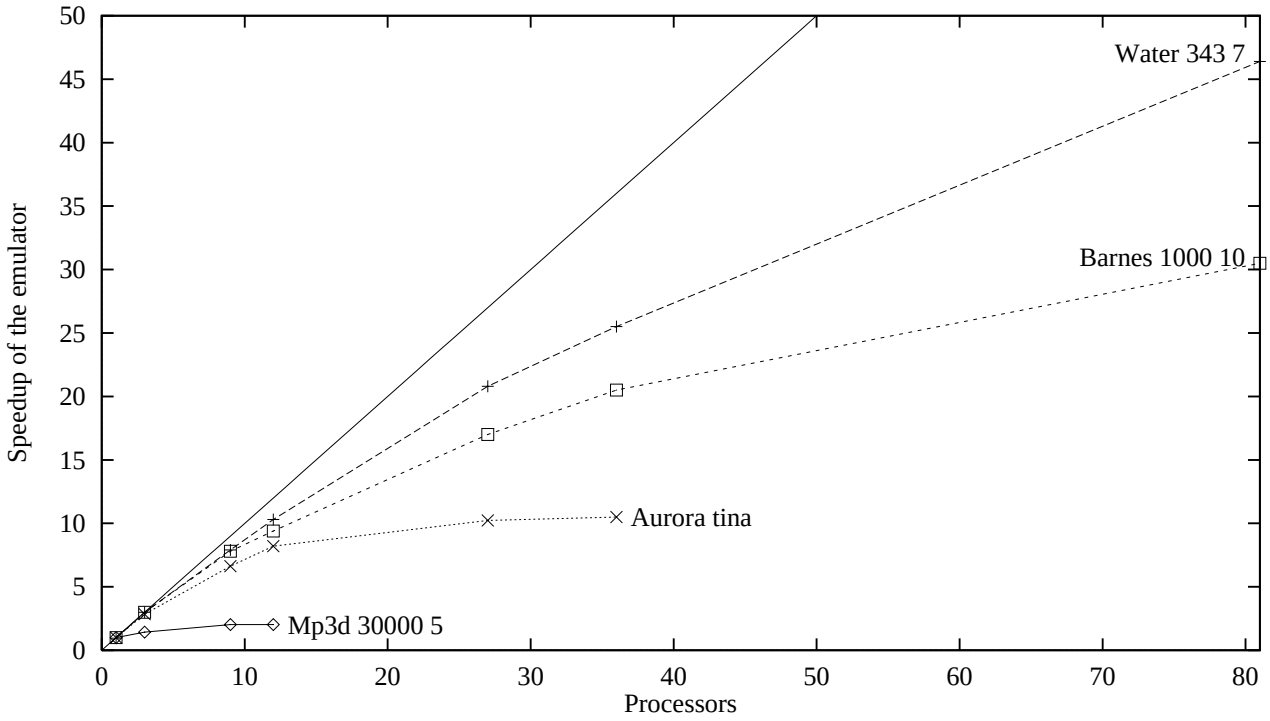

Figure 3: Speedups of the emulator running on multiple processors.

Application program	Parameter settings	Shared Reads 10^6	Shared Writes 10^6	Emulation time hr:mn	Proj. run time secs
Andorra	Fly-Pan	50	1.1	0:13	1.9
Aurora	tina	15	1.9	0:18	2.7
Water	343 7	40	6.0	6:25	57.0
Barnes	1000 10	36	0.5	2:08	19.0
Mp3d	30000 5	3.5	2.3	0:07	1.0
Locus	Primary1	6.8	1.1	0:16	2.4
Cholesky	bcsstk14	7.7	1.0	0:14	2.0
Pthor	risc 500	5.0	0.7	0:06	0.9

Table 6: Running the programs on a single processor.

are written using the p4-parallel programming library, so they are compiled for the DDM without change. Aurora and Andorra-I are written for existing shared memory machines (Sequent Symmetry, BBN Butterfly) using native operating system and compiler primitives for interfacing to the parallel hardware. We have modified the source to use the p4-library calls.

The results of running the programs on a single processor are shown in Table 6. The factor 486 slowdown can be seen in the ratios between emulation time and projected run time. On a single node system, 10^7–10^8 *shared* memory accesses can be emulated within only a couple of hours. Multiprocessor DDMs are evaluated by running the emulator on the desired processor configurations. Notice that the speedup of the emulator itself is identical to the speedup that the program would have on a multiprocessor DDM. The speedups are shown in Figure 3 (only a subset of the graphs from both good and bad performing programs are shown). Mp3d cannot be emulated faster on a multiprocessor system because Mp3d has negligible speedup on the DDM (and on other architectures as well). Barnes shows a speedup of 30 on 81 nodes.

With these parameters it is possible to emulate $\frac{100*81}{486} =$ 16 million instructions per second although in practice the applications cannot fully utilise every processor. Barnes, for example, executes about 6 million instructions per second. The emulation strategy gives performance orders of magnitude higher than that obtained with ordinary simulation. Our emulation performance is similar to that of the WWT [2], even though our emulator is running on processors an order of magnitude slower.

The DDM design that is currently emulated is not yet optimal. No latency hiding is implemented and the fanout of the tree is too low. We are working to refine the DDM protocol and to find the optimal design parameters. When the DDM design is improved, the emulator will run faster as well.

The emulator speed is limited by the component with the highest slowdown factor. At this moment the protocol is the bottleneck, although there is little to be gained from improving this speed. Table 5 shows that two other components, the local memory emulation and the instruction emulation, would otherwise become the bottleneck, with only a small performance improvement.

5 Conclusions

The emulator presented in this paper provides us with a fast and reliable evaluation platform. The emulator can run in parallel without explicit synchronisations over the network, leading to a speedup that is only bounded by the speedup of the architecture and application that we are emulating. The price paid is that parts of the emulator are artificially slowed down. On a single node, a discrete event simulator may run faster than the emulator because it takes discrete time steps. On a multiprocessor, the emulator wins because the simulator needs a synchronisation scheme to keep the clocks synchronised.

The emulation speed critically depends on the match between the timings of the emulator and the architecture to be emulated. For each component the ratio between the speed of the emulator and the speed of the architecture is calculated. The worst of these ratios determines how slowly the emulator will finally run. In the example DDM case the emulator runs only a factor 486 slower than the real hardware. If the expected hardware characteristics change because better estimations of future technology can be made, we will end up with other emulation speeds.

The emulator presented can emulate 16 million instructions per second on a system with 81 leaf transputers (200,000 instructions per second per transputer). This means that the emulator allows us to run a program that executes 10^{10} instructions in about 10 minutes, under the assumption that the program has linear speedup. If the program reaches an efficiency of for example 40%, the emulation time will be a factor 2.5 longer.

The current emulator runs on T800 transputers. We plan to port the emulator to a T9000 (the successor of the T800) based system which should improve the emulation speed by an order of magnitude, hopefully allowing us to emulate over 200 million instructions per second on a 100 node system. Additionally, the T800 imposes a maximum fanout of three which is a serious limitation for the DDM since we expect an optimal DDM tree to have a fanout of 8 to 16. The T9000 processor will make it possible to evaluate architectures with more optimal fanouts.

References

[1] David H. D. Warren and Seif Haridi. The Data Diffusion Machine—A Scalable Shared Virtual Memory Multiprocessor. In *Proceedings of the 1988 International Conference on Fifth Generation Computer Systems*, pages 943–952, Tokyo, Japan, December 1988.

[2] S.K. Reinhardt, M.D. Hill, J.R. Larus, A.R. Lebeck, J.C. Lewis, and D.A. Wood. The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers. In *Proceedings of the 1993 ACM SIGMETRICS conference*. Association for Computing Machinery, May 1993.

[3] Daniel Lenoski, James Laudon, Kourosh Gharacharloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica S. Lam. The Stanford DASH multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[4] KSR. *KSR Technical Summary*. Kendall Square Research, Waltham, MA, 1992.

[5] D. Chaiken, J. Kubiatowicz, and A. Agarwal. LimitLESS Directories: A Scalable Cache Coherence Scheme. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 224–234, 1991.

[6] E. Hagersten and S. Haridi. The Cache Coherence Protocol of the Data Diffusion Machine. In *Proceedings of the Parallel Architectures and Languages Europe, PARLE*, 1989.

[7] Sanjay Raina and David H. D. Warren. Traffic Patterns in a Scalable Multiprocessor through Transputer Emulation. In *Proceedings of the 25th Hawaii International Conference on System Sciences*, pages 267–276, 1992.

[8] INMOS Ltd. *Transputer Reference Manual*, 1988.

[9] Paul W. A. Stallard, Henk L. Muller, and David H. D. Warren. Performance Evaluation of Parallel Programs on the Data Diffusion Machine. In *Performance Evaluation of Parallel Systems, PEPS '93*, pages 94–101. University of Warwick, UK, November 1993.

[10] Helen Davis, Stephen R. Goldschmidt, and John Hennessy. Multiprocessor Simulation and Tracing using Tango. In *International Conference on Parallel Processing*, pages II–99–II–107, August 1991.

[11] H. L. Muller. *Simulating computer architectures*. PhD thesis, Department of Computer Systems, University of Amsterdam, February 1993.

[12] Ewing Lusk, Ross Overbeek, James Boyle, Ralph Butler, Terence Disz, Barnett Glickfeld, James Patterson, and Rick Stevens. *Portable Programs for Parallel Processors*. Holt, Rinehart and Winston, Inc., 1987.

[13] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. Technical report, Computer Systems Laboratory, Stanford University, 1991.

[14] E. Lusk, D.H.D. Warren, and S. Haridi et. al. The Aurora Or-Parallel Prolog System. *New Generation Computing*, 7:243–271, 1990.

[15] V. Santos Costa, D.H.D. Warren, and R. Yang. Andorra-I: A parallel Prolog system that transparently exploits both and- and or-parallelism. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practices of Parallel programming*, pages 83–93, April 1991.