Design and Implementation of a Massively Parallel Multithreaded Architecture: DAVRID

Sangho Ha, Junghwan Kim, Eunha Rho, Yoonhee Nah, Sangyong Han, Daejoon Hwang, Heunghwan Kim, and Seungho Cho

Abstract

MPAs(Massively Parallel Architectures) should address two fundamental issues for scalability: synchronization and communication latency. Dataflow architecture faces problems of excessive synchronization overhead and inefficient execution of sequential programs while they offer the ability to exploit massive parallelism inherent in programs. In contrast, MPAs based on von Neumann computational model may suffer from inefficient synchronization mechanism and communication latency. DAVRID(DAtaflow/Von Neumann RISC hybrID) is a massively parallel multithreaded architecture which takes advantages of von Neumann and dataflow models. It has good single thread performance as well as tolerates synchronization and communication latency. In this paper, we describe the DAVRID architecture in detail and evaluate its performance through simulation runs over several benchmarks.

I. Introduction

Some of the problems which have bothered dataflow machine developers are that a dataflow machine not only needs specialized hardware but also is unfamiliar to common people. In addition, dataflow architectures cause problems of excessive synchronization costs and inefficient execution of sequential programs[1]. To make dataflow machines practical we cannot help combining dataflow computing rule with conventional, von Neumann style, which offers the ability to exploit locality through registers and caches. Several MPAs consisting of a number of high performance microprocessors are recently being developed with various topologies of interconnection network. We believe that a multithreaded architecture elaborately built from conventional high performance microprocessors will give us good single- and multi-thread performance through incorporating program-counter based instruction sequencing into dataflow computer architecture.

Manuscript received July 31, 1995; accepted January 18, 1996.

S. H. Ha, J. H. Kim, E. H. Rho, Y. H. Nah, and S. Y. Han are with Department of Computer Science, Seoul National University.

D. J. Hwang is with Department of Information Engineering, SungKyunKwan University.

H. H. Kim is with Department of Computer Science, Seowon University.

S.H.Cho is with Department of Computer Science, Kangnam University.

The authors have worked together in Parallel Processing Lab. at Seoul National University.

The DAVRID[18], which has been operational with two nodes since June 1993, consists of three functional units: TPU(Thread Processing Unit), SU(Synchronization Unit), and NIMU(Node Interface and Management Unit). Each unit is a program execution unit consisting of a conventional micro-processor, but only the TPU executes the user programs: the TPU performs only pure computational jobs, and the other units help the work of TPU. The SU takes roles of synchronizations, frame allocation/deallocation, etc. and the NIMU takes roles of message routing, global memory handling, and load balancing. In the future, DAVRID will be extended to a collection of clusters each of which consists of up to 4 nodes; the NIMU interfaces among nodes within a cluster.

Token matching problem has been controversial in developing dataflow machines. Now, dataflow researchers do not consider associative waiting-matching store or hardware hashing any more. It has been replaced by explicitly addressed token store since Monsoon[2]. Monsoon uses circular pipeline and allows only short instruction threads using a small set of registers. However, there are pipeline "bubbles" in case of dyadic instructions and the use of registers is very restricted. P-RISC architecture[3] splits the "complex" dataflow instructions into separate synchronization, arithmetic and fork/control instructions. These von Neumann style instructions get together and form longer threads, and some dataflow synchronizations are replaced with conventional program-counter based synchronization.

However, it is only a conceptual architecture. ETL's EM-4[4] uses direct matching scheme, roughly identical to frame-based synchronization, and very short, restricted threads called strongly connected blocks, which provide the ability to reduce matching frequencies and permit the use of a register file.

TAM[5] is the execution model to implement synchronization, scheduling, and storage management under compiler control. Iannucci's Dataflow/von Neumann Hybrid architecture[6] had incorporated dataflow ideas into von Neumann, which affected the computational model of DAVRID. MIT's *T[7] is known to show good single thread performance with a conventional microprocessor. It's a multithreaded architecture with advantages in hiding remote memory latency and efficient synchronization. However, even if it uses a separate coprocessor for message handling, remote memory requests can cause conflicts since global and local data are stored in the unified node memory. In TAM and P-RISC[8], a processor handles messages as well as performs pure computation. Further, there is considerable scheduling overhead in TAM[8]. The DAVRID architecture is mainly inspired by *T. However, we mitigated considerably the computation processor's burden through separation of synchronization and global memory handling. DAVRID has a special synchronization unit called SU, which also takes a role of frame allocation/deallocation. Global memory of DAVRID, called SM(Structured Memory), is not located at any node, instead it is located at each cluster, which is suitable for exploiting locality and reducing the burden of the nodes.

In section 2 we describe the computational model on which our multithreaded architecture is based. In section 3 we introduce DAVRID and describe each functional unit of it in detail. In section 4 we describe how programs are executed on DAVRID. In section 5 we evaluate DAVRID through simulation results over several benchmarks which include scalability, utilization, and data distribution. Finally, in section 6 we remark conclusion and future works.

II. Computational Model

In general, parallel architectures based on von Neumann computational model are known to have performance degradation due to excessive overheads resulted from latency and synchronization costs which increase with the number of processors[9]. In contrast, von Neumann computational model provides efficient execution for sequential and state dependent programs, easy resource management, and portability of enormous extant software[1]. In this paper, we suggest a multithreaded model which is a hybrid of von Neumann model and dataflow model. In the multithreaded model, two fundamental problems[9] of parallel processing environment can be tackled by dataflow computing rule while still preserving the advantages of von Neumann model.

In the multithreaded model, the computation unit is a thread which consists of logically related sequential instructions which never wait for any event. We can easily solve the problems of excessive synchronization overhead and inefficiency of sequential programs in dataflow model, permitting the synchronization only between threads. A thread is sequentially executed, exploiting locality through the use of registers and caches, but no explicit synchronization is required. In addition, we can tolerate long latencies by applying dataflow computing rule to inter-thread execution. Note that threads must be formed by a compiler so that any long latency operation should be executed through split-phase. From these points of view, the multithreaded model is based on the hybrid model which combines advantages of von Neumann model and dataflow model.

In DAVRID, a thread can be scheduled to a processor, TPU, only when all values and signals for the thread are satisfied. Once a thread is scheduled to a processor, the thread is nonpreemptively executed while accessing memory and registers. During execution, the thread may generate messages to remote memory or other threads, if necessary. The processor switches its control to another thread as soon as the thread completes. Since it is not required to save processor status upon switching threads, there is little context switching overhead.

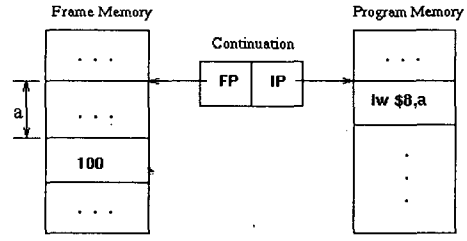


Fig. 1. Continuation.

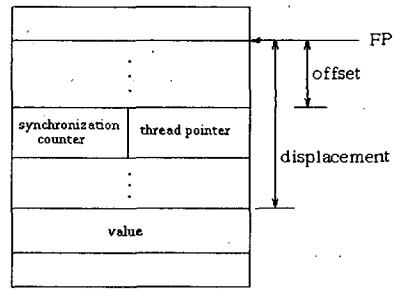


Fig. 2. Frame Structure.

Each thread in DAVRID is identified by a "continuation", <*fp*, *ip*> where *fp*, a frame pointer, is the base address of the frame which is the local data area for a function, and *ip*, an instruction pointer, is the starting address of the thread which is going to execute a part of that function(see Figure 1). To indicate how many events remain to be required for the execution of the thread, we introduced a synchronization counter. An event means the arrival of a value or a signal. Note that a frame is allocated for each activated function or loop. During execution of functions or loops, processors read/write values from/into the frame. In a frame, a synchronization counter and the related thread code pointer(*ip*) are relatively addressed by *offset*, and a value slot is relatively addressed by *displacement* as seen in Figure 2. The synchronization counter is initially the number of events needed for the activation of the thread. When an event occurs, the synchronization counter is decreased by 1, and a value, if exists, is stored at the value slot. If the synchronization counter becomes 0, the corresponding thread is activated.

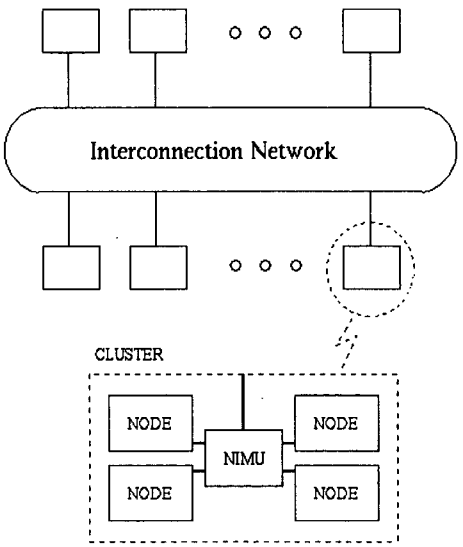


Fig. 3. DAVRID Architecture.

III. DAVRID

DAVRID consists of a collection of clusters, which communicate with each other through a pipelined, message based interconnection network(see Figure 3). Each cluster has nodes up to four and node-to-node communication is done via NIMU(Node Interface and Management Unit).

Each node has its own local memory and may work independently. Local memory is used for frame area, and each frame can be directly handled by its owner node, i.e., the processor in a node can directly load and store data in its own frame. We refer to the local memory as FM(Frame Memory). If the TPU wants to store some data into a frame which is allocated in another node, it should send a message to the node. At that time, of course, global address should be formed by concatenating the node id.

Global data are stored in the memory which is inside NIMU. It must be handled by means of messages and the processor in a node can not directly load or store data from or in it, i.e., global data are always "remote". We refer to the global memory as SM(Structured Memory), since it is mostly used for global data structures like I-structure[10]. The SM is managed by a NIMU and placed in a cluster, not a node. Similar to FM, we can get the global address of SM by concatenation of cluster id. We can tell whether a message is for SM or FM by its header.

1. Node Architecture

A node consists of TPU(Thread Processing Unit), SU(Synchronization Unit), FM(Frame Memory), and message queues(see Figure 4). The TPU is a core unit to execute threads. It consists of a conventional RISC processor and TM(Thread Memory) which contains program codes. The SU is a unit for synchronization. It also consists of a conventional RISC microprocessor and its local memory containing a message handler. These two units communicate using ATQ(Active Thread Queue) and STQ(Setup Token Queue), and share the FM.

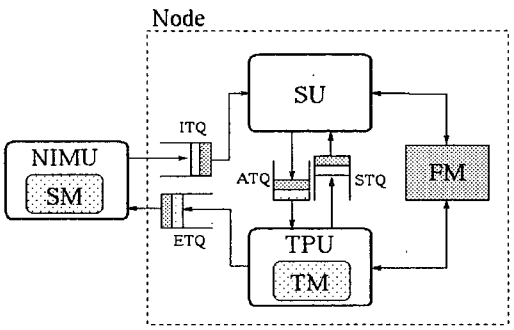


Fig. 4. Organization of the DAVRID Node.

Thread Processing Unit

This unit executes threads in a sequential manner using a conventional RISC processor. It is supplied with the continuation of next thread from the ATQ. Note that it is "multithreading" in the system's view, but it's just a sequence of conventional instructions in the processor's view.

We benefit from using an existing conventional RISC processor in a few points. Existing sequential programs can be used without any modification since there is no inter-thread synchronization and the SU is thoroughly

transparent to those programs. The other merit is that we can exploit the locality among instructions using general-purpose registers and the program-counter based synchronization. The TPU can execute single threaded programs efficiently as well as multithreaded programs.

Thread Switching: Compared with single threaded (sequential) programs, the thread switching may be overhead. To get the performance gain from multithreading, the switching time should be sufficiently smaller than the remote memory latency. While the thread switching occurs implicitly on cache miss in some multihreaded architectures like Alewife[11], the thread switching occurs explicitly in DAVRID. There is no notion of the thread "suspension" but "termination" in our model. When the processor of the TPU meets NEXT instruction, it terminates the current thread and begins a new thread. Thus, there is no need to save the processor status. The TPU just fetches two words - *fp* and *ip* - from the ATQ. The register containing *fp* is used as the base register at the memory address calculation. *ip* is used for branching to the starting point of the new thread. Since NEXT is not a conventional instruction, we should implement by a few instructions which perform the above operations.

Message Forming: There are many cases of generating the messages: synchronizations, global memory accesses, etc. A message is generated by, logically, a special instruction that is composed of conventional RISC instructions. These instructions generate the message and insert it into ETQ(External Token Queue). The message forming overhead may be mitigated by a hardware facility like a message formatter, but we didn't consider the message formatter in our first prototype.

As we described above, some special instructions are required to form messages and switch the threads, e.g., NEXT instruction. In other words, our abstract machine needs its own instruction set to emulate long latency operations, explicit/implicit synchronizations, and other multithreaded features. We refer to the instruction set as DAVRID PML(Parallel Machine Language). The PML defines basic operations of DAVRID, which includes synchronization of threads, global memory access, dynamic memory management, and conventional load/store, arithmetic/logic, and branch(see Appendix). Conventional instructions are defined as MIPS R3000 ISA[12] since our implementation is based on MIPS R3000 family.

As already explained, a PML instruction corresponds to a collection of R3000 instructions. Most PML instructions generate a message containing a "request". In that case, SU or NIMU may serve the request.

The TPU itself can serve directly requests local to its own frame memory without generating messages. Such instructions are as STARTl and STARTln. These instructions can locally activate a thread: when synchronization is completed, a continuation is produced. In that case, STQ(Setup Token Queue) is used for passing the continuation to the SU. Note that the TPU can't immediately execute that thread since current thread does not complete yet. So, the TPU just passes the continuation to the SU through the STQ and then the SU moves it to the ATQ again.

Synchronization Unit

This unit fetches messages from the ITQ(Internal Token Queue) and performs operations indicated by them. There is a message handler having a collection of routines to handle each kind of messages. The message handler is similar to "entry" of P-RISC[8], "inlet" of TAM[5], and synchronization-purpose thread of *T[7], but those codes are generated by a compiler according to applications. Our message handler is independent of application programs, which means that it is set up at system development time. In addition it's not limited to dealing with data or synchronizations. Many kinds of messages which perform various functions are elaborately designed. These messages can be roughly categorized into synchronization and frame allocation/deallocation.

Synchronization: It is an intrinsic function of the SU. Data needed to execute a thread are stored into the FM by the SU. If all the data are stored, the SU enables the corresponding thread, i.e., inserts *fp* and *ip* into ATQ. Data received by the SU are the responses of the SM access or data(or signals) sent by another thread. Though there are many kinds of messages related to synchronization, we describe START message as a typical example. When the message handler meets a START message, it performs the following operations:

```
message: START fp, offset, disp, value
operations: FM[fp+disp] <-- value
        if synchronization counter in FM[fp+offset] = 1
        then ATQ <-- fp
             ATQ <-- ip in FM[fp+offset]
        else
             decrease synchronization counter in FM[fp+offset]
```

START is a message for getting thread synchronized by sending a value to another thread. Such a message can be generated by the TPU, the NIMU, or the SU as a response of the request message.

Frame Allocation/Deallocation: When the SU receives a message requesting frame allocation or deallocation, it allocates/deallocates the frame by the requested size and sends a response message to the source node. There are various kinds of messages in allocating/deallocating frames. The frame allocation/deallocation job belongs to a run-time system, which is executed in the SU and the NIMU. Compared with other run-time systems in which the frame

allocation/deallocation is performed within a processor like TPU, this approach has some advantages: workloads of TPU are distributed, i.e., TPU is free from memory allocation/deallocation and scheduling, and performs only pure computational jobs.

2. The Node Interface and Management Unit

NIMU also handles messages as SU does, but their functions are different in that the NIMU works as a message router and an SM manager, and balances workloads.

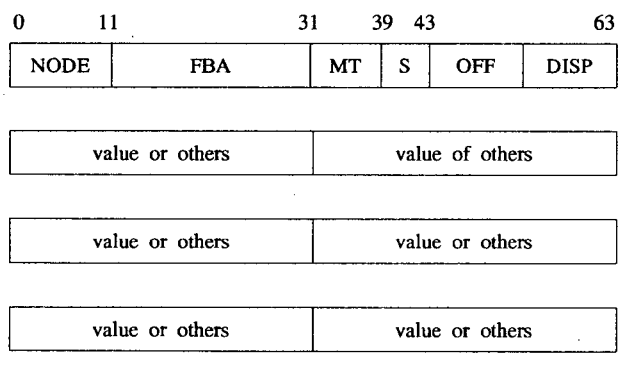
Message Routing: If a received message is destined for a node in the same cluster, the NIMU directly inserts it into ITQ(Internal Token Queue) of the destination node. Otherwise, the NIMU sends the message to the destination node through the interconnection network.

Structured Memory Handling: Another important function of the NIMU is global memory management. In case the message has the address of a cluster instead of a node, it means that the message requests access to SM. In case of normal memory access, the NIMU simply performs normal memory operation with the address extracted from the message. But, I-structure operations need synchronization.

Load Balancing: A message which requests frame allocation is not destined for a specific node. The NIMU chooses a node at which the requested frame will be allocated, and routes the message to that node. Then, the SU of that node allocates the frame and sends the response to the source node. Assuming that total allocated frame size is proportional to workload of the node, the NIMU distributes frame allocations among nodes or clusters. The NIMU maintains a table to keep track of the quantity of allocated frames at each node within the cluster.

3. Message Packets

Messages moving around the DAVRID are generated by not only the TPU, but also the SU and the NIMU. A message generated by the TPU contains "request" to the SU or the NIMU, and the SU or the NIMU generate a "response" message which is destined for the SU of the original node. The request messages are roughly categorized into synchronization, frame allocation/deallocation, and SM access. The "response" message can't be distinguished from the "request" message by their forms. All messages used in DAVRID have the general form described in Figure 5.



NODE: Node Identifier (12 bits) S: Size (4 bits)
FBA: Frame Base Address (20 bits) OFFSET: Offset (10 bits)
MT: Message Type (8 bits) DISP: Displacement (10 bits)

Fig. 5. General Message Packet Structure.

For instance, START message has the packet structure as Figure 6. The bit pattern meaning START message is assigned to the MT field. There are other similar messages: STARTr, STARTn, STARTN, and STARTc.

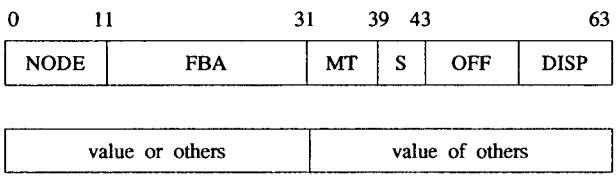


Fig. 6. Message Packet Structure of START.

Note that the above messages are always destined for nodes, so the first field of each message is node id. The message requesting SM access is not destined for any node but a cluster. Thus, the first field of such a message is always cluster id. Figure 7 illustrates the packet structure of ILOAD as an example. The third word - NODE and FBA - indicates the source node context.

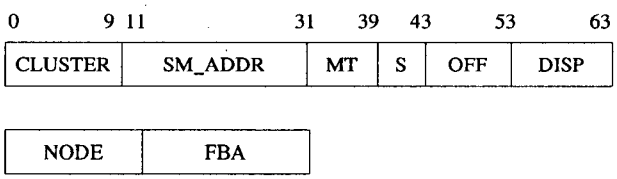


Fig. 7. Message Packet Structure of ILOAD.

4. Implementation

Our implementation is focused on developing the prototype using only conventional microprocessors. We believe that this contributes to showing that multithreaded model based on dataflow computing can be directly realized without specialized hardware.

Each unit of DAVRID consists of a commercial RISC microprocessor and local memory. TPU was implemented using a MIPS R3001 processor. We refer to the local memory of TPU as TM into which multithreaded programs are loaded. SU was implemented using a MIPS R3051 processor. Additionally it has EPROM containing a message handler. The physical implementation of NIMU is similar to

SU. NIMU also has a MIPS R3051 and EPROM containing a message handler, but this message handler differs from that of SU. Local memory of NIMU is used for global data, and we refer to it as SM.

FM and message queues(ATQ, STQ, ITQ, and ETQ) which interface each unit are incorporated into a module referred to as UIM(Unit Interface Module)(see Figure 8). We used commercial FIFO chips as message queues each of which has dual ports and ensures asynchronous simultaneous accesses. Each message queue is "memory-mapped". Thus, each unit can delete/insert an item to the interfaced message queue by a load/store operation with specific address. The FM is implemented by commercial dual-port SRAMs, which ensures fast asynchronous simultaneous accesses by both TPU and SU.

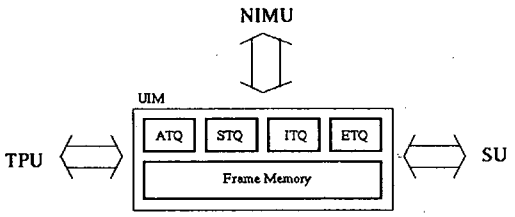


Fig. 8. Unit Interface Module.

DAVRID prototype was implemented as 2-node system which has one NIMU and two nodes. We are considering various topologies as cluster-to-cluster interconnection network but didn't designed one yet.

ETL's EM-4 consists of a collection of specially designed single chips called EMC-R[4]. This is completely orthogonal to our approach. EMC-R consists of Switching Unit, Input Buffer Unit, Fetch and Matching Unit, Execution unit, and Memory Control Unit. Switching Unit, Fetch and Matching Unit, and Execution Unit are roughly similar to NIMU, SU, and TPU of DAVRID respectively. But there is a fundamental difference: units of EMC-R are pipelined and their operations can be overlapped, but not asynchronously concurrent. And each unit of DAVRID can perform more complex and various functions than those of EMC-R. All units of DAVRID are program execution units. In terms of computational model, the Strongly Connected Block of EM-4 is less general than the thread of DAVRID.

MIT's *T is very similar to our approach, but its architecture differs from ours. sP(Synchronization Coprocessor) and RMem(Remote Memory Request Coprocessor) of *T are roughly identical to SU and NIMU of DAVRID respectively. But the functions of NIMU are not only global memory handling but also message routing and load balancing. And NIMU does not belong to any node but to a cluster. SU is functionally a superset of sP of *T. The SU performs memory management as well as synchronization. Moreover, the message handler of SU is fixed, while sP executes the threads generated by a compiler. sP is only logically a separate coprocessor. The separation was physically not pursued while the SU is a physically separate, distinct unit which is designed to pursue concurrency.

IV. Program Execution

We implemented Id and Fortran for DAVRID. Id[21]is a functional core of Id[14] with I-structure and extended to allow user to specify loop unfolding degree explicitly. Each of Id and Fortran compilers has its own front-end but shares a back-end. Our compiler uses dataflow program graph[15] as an intermediate form between front-end and back-end, which has a hierarchically structured form that facilitates high level optimizations without consideration of details on the architecture. Thus programs written in Id or Fortran are all translated into program graphs.

The program graph is transformed into the machine graph for DAVRID. The machine graph[19], called the DAVRID graph, has well-defined operational semantics. So, the DAVRID graph has the same level of abstraction as the machine graph of the compilers for TTDA and Monsoon. The DAVRID graph is partitioned according to its partitioning scheme[19], which mainly makes long latency instructions and their output instructions belong to different partitions each other to deal long latency instructions with as split transactions, and performs several optimizations to minimize communication and synchronization between partitions. Also, loop handling[20] is specially considered under multithreading environment. Loops can be unfolded or sequentialized, depending on the amount of parallelism and the amount of communication and synchronization in their body.

Thread codes for each partition are initially represented in DAVRID PML, and then become input form for assembler by translating each DAVRID PML instruction into a set of R3000 assembly instructions: a DAVRID PML instruction for the memory allocation/deallocation and synchronization among threads is translated into a sequence of R3000 assembly instructions, representing a message formation routine to make a run-time system call. Finally, the thread codes become an executable form for DAVRID real machine through assembler and loader.

The run-time system of DAVRID supports the execution environment of programs and manages resources of system: thread management, memory management, and message management. For fast prototyping of DAVRID, we did not much care about optimizations on run-time systems. Thus, to make DAVRID suitable for MPP(Massively Parallel Processing) systems, it is required to incorporate efficient

thread scheduling, dynamic allocation/deallocation for frame and heap, and efficient message formation in the run-time system.

V. Evaluation

This section evaluates the performance of DAVRID by simulation. Since real machine currently has only two nodes, and is in the early stage of development, it is difficult to evaluate the performance thoroughly with the real machine. So, we developed two simulators: clock cycle level simulator and thread level simulator. We first estimated performance parameters exactly using the simulator with the level of clock cycle, then reflected the results to the simulator with the level of thread. We can evaluate our machine fast and exactly through the above method.

We will evaluate the DAVRID in terms of scalability and utilization. In addition, we will analyze the effects of data distribution on the machine. Four benchmark programs are used for our simulation: paraffins problem, matrix multiplication, Fibonacci function, and LLL1. Paraffins problem[16] describes how to enumerate the distinct isomers of paraffins of size up to n, which is 13 in our evaluation. In matrix multiplication, matrices with dimension 20×20 are used. Fibonacci function is a recursive function with the argument 15. LLL1 in the Lawrence Livermore Loops[17] is also used.

Scalability

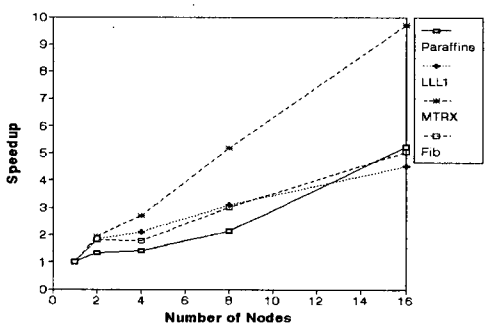


Fig. 9. Scalability.

Figure 9 shows scalability of DAVRID up to 16 nodes. Scalability of matrix multiplication is better than that of other benchmarks. It can be interpreted in terms of computation time, communication time, and properties of problems. There exists massive parallelism in matrix multiplication and LLL1. But the size of loop body of matrix multiplication is larger than that of LLL1. Note that innermost loop of matrix multiplication is sequential and the outer level loops are unfolded. Loop unfolding overhead for LLL1 seems to be too large to exploit parallelism sufficiently due to small size of computation of the loop body. In the Fibonacci function, the function body takes only a little computation time, but a considerable much communication and many synchronization occurs among frames dynamically allocated. Even more, the current runtime system considers only load balancing, not consider the locality of computation. So, the effects of parallel execution for functions called recursively is overridden by its overhead. There exists much consumer and producer parallelism in paraffins problem. So, it causes excessive operations on SM, which usually consists of long latency instructions. From this point of view, it is very important to consider the locality of computation to reduce the overhead of operations on SM.

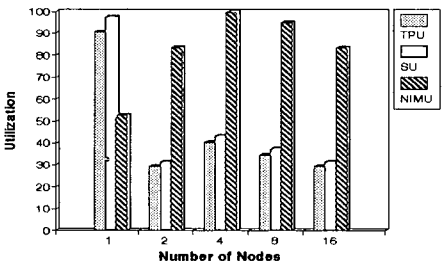


Fig. 10. Utilization(Fib).

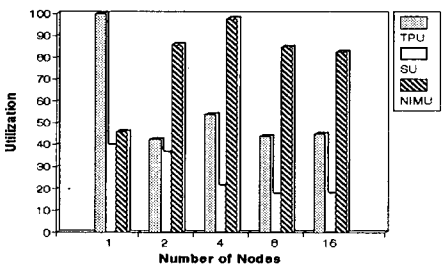


Fig. 11. Utilization(LLL1).

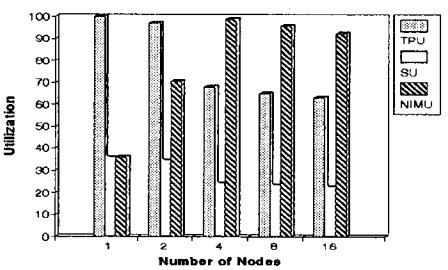


Fig. 12. Utilization(MTRX).

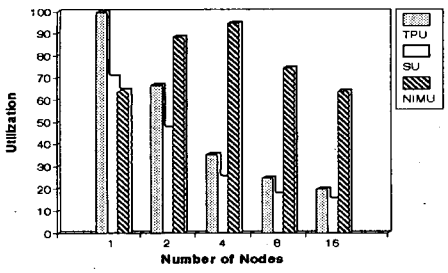


Fig. 13. Utilization(Paraffins).

Utilization

Utilization of each functional unit(TPU, SU, and NIMU) in DAVRID is shown in Figure 10-13. We can see that as the number of nodes increases up to 4, utilization of TPU decreases, and that of SU a little decreases while that of NIMU increases. This situation is explained by the bottleneck caused in NIMU, which is in charge of handling operations on SM, load balancing, and message processing. In other words, as the number of nodes increases up to 4, the number of messages to be sent to NIMU increases linearly, but throughput of NIMU is constant at the bottleneck point. So, under the bottleneck of NIMU, each SU in a cluster receives the decreased number of messages from NIMU. Thus, since fewer tokens are queued in ATQ of the corresponding node, utilization of TPU is reduced. Also, up to 16 nodes, this situation still stay except for a little difference which can be explained in the same way. We are trying to solve these problems by exploiting the locality of computation embedded in programs. For example, we could get the improved results by more than ten times by means of careful handling of data distribution. We will optimize the compiler and run-time system to make the locality be exploited sufficiently.

Data distribution

We will evaluate several strategies on data distribution under 16 nodes(4 clusters) using matrix multiplication, in Figure 14. In the first case('Iarray in a SM'), all arrays of matrix multiplication are allocated on the NIMU of only one of 4 clusters. So, for nodes of clusters where the arrays are not allocated, all array operations on SM result in a considerable communication overhead due to network routing. Thus the loop unfolding for more than 4 nodes makes little effect. In the second case('Global SM'), arrays are allocated on the centralized shared memory as Monsoon instead of NIMU. Although this type of shared memory doesn't exist in DAVRID, it is specially considered to compare the effect of data distribution. Similar to the first case, this case shows that communication can't be overlapped with computation due to the considerable communication overhead and the small size of computation of the loop body. In the fourth case('Distr. Iarray'), unlike the first case, arrays are optimally distributed across 4 clusters; all array elements used in the TPU are localized within the corresponding cluster. So, communication overhead can be considerably lessened since operations on SM don't cause network routing any more. With this strategy, the loop unfolding provides linear speed-up up to 16 nodes as shown in Figure 14. In the third case('Local Iarray'), all array elements used in the TPU are localized within the frame memory associated with the TPU instead of the NIMU. This strategy can be realized by fetching a bundle of array elements by the column or the row at a time. So, the size of threads can be enhanced since only one long latency operation for fetching the bundle of array elements is required. In contrast, in the fourth case, one long latency operation is required for each array element. Thus, the third case provides more improved performance than the fourth case. From this experience, we are developing the data distribution mechanism controlled by user and compiler.

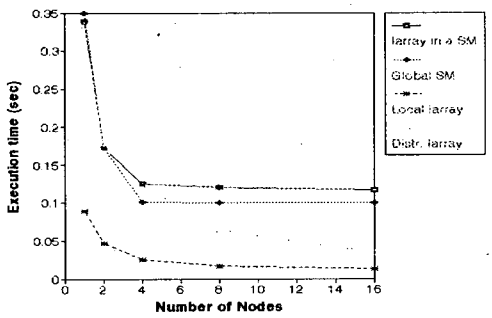


Fig. 14. Comparison of Data Distribution Strategies.

VI. Conclusion

Most of large scale computers inherently have very long processor-processor or processor-memory communication latency. This problem would be solved by global coherent caches or multithreading scheme. Both of them solve the problem through reducing or hiding "remote load" latency. However, global caches do not deal with "synchronizing load" problem which is "when are data ready to load?" We use multithreading with dataflow model to solve this synchronization problem. Dataflow model is also advantageous for representing parallelism and scheduling tasks, however, pure dataflow model needs excessive synchronizations which is a difficulty in developing high-speed processors. Thus, we chose hybrid model which employs the dataflow rule only to inter-thread level. As we just use commercial microprocessors for intra-thread

Instruction	Semantics
thread activation and synchronization	
START fp,off,disp,val	TPU: send a START message SU: FM[fp+disp]←val, decrease sc at FM[fp+off]
STARTr fp,off,disp,r_off,value	TPU: send a STARTr message SU: FM[fp+disp]←val, decrease sc at FM[fp+off], send a signal to r_off
STARTl off,disp,val	TPU: FM[current fp+disp]←val, SU: decrease sc at FM[current fp+off]
STARTn fp,off	TPU: send a STARTn message, SU: decrease sc at FM[fp+off]
STARTN fp,disp,val,off	TPU: send a STARTN message SU: FM[fp+disp]←val, decrease sc at FM[fp+off]
STARTln off	TPU: send <fp, off> to STQ SU: decrease [current fp+off]
STARTd fp,ip	TPU: activate thread <fp, ip>
STARTc closure_addr	TPU: send a STARTc message SU: traverse all closure structures for a function call
NEXT	fetch the next continuation from ATQ
ILOAD smaddr,off,disp	TPU: send an ILOAD message NIMU: if SM[smaddr].tag is full, then send SM[smaddr].val to off:disp, else deferred_list←ILOAD request
ISTORE smaddr,off,val	TPU: send an ISTORE message NIMU: SM[smaddr].val←val, send a signal to off
ISTOREr smaddr,val	TPU: send an ISTOREr message, NIMU: SM[smaddr].val←val
ARRAY_BOUND smaddr,off,disp	TPU: send an ARRAY_BOUND message NIMU: send SM[smaddr+lb_1], SM[smaddr+ub_1], ... to off:disp
memory management	
FALLOC f,size,off,disp,init_cjc,return_sc,off_1, dis_1,off_2,dis_2, ...	TPU: send a FALLOC message SU: allocate FM, send fp to off:disp
M_FALLOC f,size	NIMU: send a M_FALLOC message SU: allocate a FM, start initialization of the main function frame
FDEALLOC fp,size	TPU: send a FDEALLOC message, SU: deallocate FM
AP1 f,size,arg_val,ref_count,off,disp	TPU: send an AP1 message SU: allocate a closure structure, send fp to off:disp
APn closure_addr,arg_val,ref_count,off,disp	TPU: send an APn message SU: allocate a closure structure, send fp to off:disp
APf closure_addr,arg_val,fp_off,fp_dis, return_sc,off_1,dis_1,off_2,dis_2, ...	TPU: send an APn message SU: allocate a closure structure & a function frame, send fp to fp_off:fp_disp,
GETsel1_FRAME K,size,off,disp,init_csc, cleanup_cjc,rv_off1,rv_dis_1,rv_off_2,rv_dis_2, ...	TPU: send a GETsel1_FRAME message SU: allocate K loop frames
PUTsel1_FRAME disp,K,size	TPU: send a PUTsel1_FRAME message SU: deallocate K loop frames
GETsel2_FRAME L,K,size,off,disp,init_csc, cleanup_cjc,ready_sc,setup_csc_p,setup_csc_p, rv_off_1,rv_dis_1,rv_off_2,rv_dis_2, ...	TPU: send an GETsel2_FRAME message SU: allocate a sequential loop frame, send fp to off:disp
LDEALLOC fp,size	TPU: send a LDEALLOC message SU: deallocate a sequential loop frame
GETpl1_FRAME L,K,size,off,disp,init_csc, cleanup_cjc	TPU: send a GETpl1_FRAME message SU: allocate a parallel loop frame, send fp to off:disp
PLDEALLOC FP, size	TPU: send a PLDEALLOC message SU: deallocate a parallel loop frame
HALLOC off,disp,type,n,l_1,u_1,l_2,u_2, ...	TPU: send a HALLOC message NIMU: allocate SM, send smaddr to off:disp
HDEALLOC smaddr	TPU: send a HDEALLOC message, NIMU: deallocate SM
others	
HOST_OUT1 type,val_1,val_2, ...	TPU: send a HOST_OUT1 message NIMU: send a message to the host
HOST_OUT2 $type_1$,val_1,$type_2$,val_2, ...	TPU: send a HOST_OUT2 message NIMU: send a message to the host
HALT	program termination
load/store, arithmetic/logic, branch, ...	same as R3000 assembly instructions[3]

FM:frame memory, SM:structured memory, sc:synchronization counter, K:unfolding degree

Appendix. DAVRID PML instruction set

execution, it isn't required to design a customized processor for that purpose. We, focusing on the above, described components and organization of our architecture, DAVRID, and how to make it work efficiently.

Although we take advantage of multithreading idea to hide latency, if there still exists much node-to-node communication it causes network bottleneck or performance degradation. As we have already seen in section 5, performance may be varied depending on the strategy of data distribution. Optimal data distribution is important to reduce communication overhead and thus has drawn our attention to this topic. Our recent work on data distribution is published in [22].

References

[1] D. D. Gajski, D. A. Papdua, D. J. Kuck, and R. H. Kuhn, "A Second Opinion on Data Flow Machines and Languages," IEEE Computer, pp. 58-69, 1982.

[2] G. M. Papadopoulos and D. E. Culler, "Monsoon: an Explicit Token-Store Architecture," Proc. 17th Annual Symp. on Computer Architecture, pp. 82-91, 1990.

[3] R. S. Nikhil, "Can dataflow subsume von Neumann computing?," In Proc. 16th Annual Int'l Symp. on Computer Architecture, pp. 262-272, 1989.

[4] S. Sakai, Y. Yamaguchi, K. Hiraki, and T. Yuba, "An Architecture of a Dataflow Single Chip Processor," In Proc. 16th Annual Int'l Symp. on Computer Architecture, pp. 46-53, 1989.

[5] D. E. Culler, S.C. Goldstein, K.E. Schauser, and T. von Eiken, "TAM-A Compiler controlled Threaded Abstract Machine," J. of Parallel and Distributed Computing, vol. 18, pp. 347-370, 1993.

[6] R. A. Iannucci, "Toward a Dataflow/von Neumann Hybrid Architecture," In Proc. 15th Annual Int'l Symp. on Computer Architecture, pp. 131-140, 1988.

[7] R. S. Nikhil, G. M. Papadopoulos, and Arvind, "*T: A Multithreaded Massively Parallel Architecture:," In Proc. 19th Annual Int'l Symp. on Computer Architecture, pp. 156-167, 1992.

[8] R. S. Nikhil, "A Multithreaded Implementation of Id using P-RISC Graphs," In Proc. 6th Annual Workshop on Languages and Compilers for Parallel Computing, 1993.

[9] Arvind and R. A. Iannucci, "Two Fundamental Issues in Multiprocessing," In Proc. DFVLR Conf. on Parallel Processing in Science and Engineering, 1987.

[10] Arvind, R. S. Nikhil, and K. Pingali, "I-Structure: Data Structures for Parallel Computing," ACM Transactions on Programming Languages and Systems, vol. 11, No. 4, pp. 598-632, 1989.

[11] A. Agarwal, D. Chaiken, G. D'Souza, et al., "The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor," In Proc. Wkshp. on Multithreaded Computers, Supercomputing, 1991.

[12] IDT RISC, R3000 Family Assembly Programmer's Guide, Integrated Device Technology, 1988.

[13] S. Y. Han, M. S. Park, D. J. Hwang and H. H. Kim, A Study on the Parallel Computer Architecture, ATRC-409-92525, Korean Agency for Defence Development, 1992.

[14] R. S. Nikhil, "Id - Language Reference Manua l(Version 90.1)," MIT CSG Memo 284-2, July, 1991.

[15] K. R. Traub, "A Compiler for the MIT Tagged-Token Dataflow Architecture," TR-370, MIT Lab. for CS, Aug. 1986.

[16] R. S. Nikhil and Arvind, "Id: A Language with implicit Parallelism," A Comparative Study of Parallel Programming Languages: The Salishan Problems, Elsevier Science Publishers B.V., pp. 169-215, 1992.

[17] F. H. McMahon, "The Livermore Fortran Kernels: a Computer Test of the Numerical Performance Range," Lawrence Livermore National Lab., UCRL-53745, UC Livermore, 1986.

[18] S. H. Ha, J. H. Kim, E. H. Rho, and et al., "A Massively Parallel Multithreaded Architecture: DAVRID," Int'l Conf. in Computer Design, Oct. 1994.

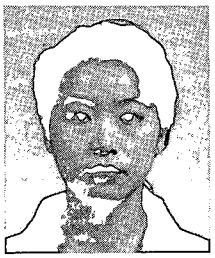
[19] S. Ha, S. Han and H. Kim, "Partitioning a Lenient Parallel Language into Sequential Threads," In Proc. of the 28th Hawaii Int. Conf. on System Sciences, Vol. 2, pp. 83-92, 1995.

[20] S. Ha, H. Kim, and S. Han, "The Efficient Implementation of Sequential Loops in Multithreaded Computation," In Proc. 7th IASTED-ISMM Int'l Conf. on Parallel and Distributed Computing and Systems, Oct. 1995.

[21] E. H. Rho, S. H. Ha, S. Y. Han, and et al, "Compilation of a Functional Language for the Multithreaded Architecture: DAVRID," Int'l Conf. on Parallel Processing, Aug. 1994.

[22] Eunha Rho, Heunghwan Kim, Daejoon Hwang, Sangyong Han, "Effects of Data Bundling in Non-strict Data Structures," In Proc. of the IFIP Working Conf. on Parallel Architectures and Compilation Techniques, pp. 140-148, Jun. 1995.

Sangho Ha has been with Electronics and Telecommunications Research Institute since Sept. 1995, where he works on the development of parallel compiler and ILP compiler. Also, He is a special researcher at RIACT. He received a BS, an MS, and a PhD in computer science from Seoul National University in 1988, 1991, 1995, respectively. While on the graduate student, he conducted several researches in parallel processing, including the DAVRID multithreaded architecture. His research interests are parallel processing, programming language, parallel compiler, and ILP compiler.

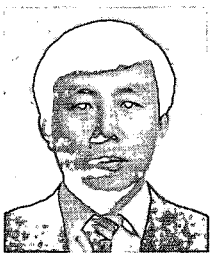


Eunha Rho received her B.S. and M.S. degrees in computer science from Seoul National University, Seoul, Korea in 1989 and 1991 respectively. She is currently working as a doctoral student at Seoul National University. Her research interests include parallel processing, programming languages, and compilers

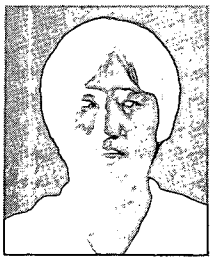
Sangyong Han received the B.S. degree in applied mathematics in 1972 and the M.S. degree in computer science in 1977 all from Seoul National University. He received his Ph.D. degree in computer science from Texas University, Austin, U.S.A. in 1983. He was a full-time lecturer at Ulsan University in 1977 and he has been a professor of Computer Science Department at Seoul National University since 1984. His research interest is parallel processing.

Junghwan Kim received the B.S. degree in computer science in 1991 and the M.S. degree in computer science in 1993 all from Seoul National University, Seoul, Korea. He is currently working toward the Ph.D. degree in computer science at Seoul National University. His research interests include computer architecture, parallel and distributed systems, programming model and compilers.

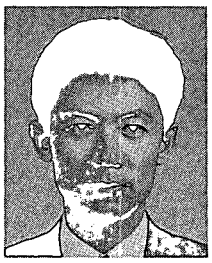
Nah, Yoonhee received her B.S. and M.S. degrees in computer science from Seoul National University, Seoul, Korea in 1989 and 1991 respectively. She is currently working as a doctoral student at Seoul National University. Her research interests include parallel processing, programming languages, and heterogenous computing.



Dae Joon Hwang received his M.S. and Ph.D. degrees in computer science from Seoul National University, Seoul, Korea in 1981 and 1986, respectively. He has been a professor of Information engineering Department at Sung Kyun Kwan University, Korea, since 1987. His research has been much focused on multithreaded computer architecture design, parallel processing into real-time multimedia collaboration. Before joining the SKKU, he esd an assistant and associate professor of the Department of Computer Science at Han Nam University, Taejon, Korea from 1981 to 1987. From 1990 he spent a year working with Prof. Arvind at MIT. Also, he conducted research on multithreaded computer architecture with Dr. K. Ekanadham, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, between November 1993 and March 1994. He is a member of the KISS, the AACE, the ACM and the IEEE and its several SIG's



Heunghwan Kim received his B.S., M.S. and Ph.D. degrees in computer science from Seoul National University, Seoul, Korea in 1985, 1987 and 1990 respectively. He has been an assistant professor of Computer Science Department at Seowon University since 1990. His research interests include parallel computer architecture, computational model and programming languages.



Seung Ho Cho is an assistant professor in the Department of Computer Engineering at the Kangnam University. He received a B.S. degree in the Department of Computer Engineering in 1985, and his M.S. and Ph.D. degrees in the Department of Computer Science in 1989 and 1993, respectively, all from Seoul National University. His research interests include parallel architectures, microprocessor applications, parallel programming and performance evaluation. He is a member of KISS, KITE, IEEE Computer Society, and ACM.