Design and simulation of an MIMD shared memory multiprocessor with interleaved instruction streams

Thomas R. Stiemerling

Ph.D.

University of Edinburgh

1991

Abstract

The design of the Eppl MIMD shared memory multiprocessor is described, and its performance evaluated by simulation. The Eppl has a dancehall architecture with p instruction interleaved RISC processors connected to p shared memories by a packet switched, combining, indirect binary n-cube multistage network composed of $\frac{p}{2}\log_2 p$ 2×2 crossbar switches. There is no processor cache or local memory, and no paged virtual memory. Memory addresses are low order interleaved across the memories. The fetch-and-add instruction is used for inter-process synchronisation, and the switches support the combining of load and fetch-and-add memory requests. Simulation results of a single Eppl processor with varying interleaving level and instruction mix are presented, and of an isolated network with varying queue size and network load. A distributed time-driven, instruction level simulator of the Eppl design has been implemented in Occam, and runs on a transputer based, distributed memory multiprocessor. Three parallel benchmark programs: matrix multiply, bitonic merge sort and Moore shortest path, have been written in the processor assembly language, and are used as workloads in the simulations. The programs use the fetch-and-add instruction to implement process control primitives. A number of simulation experiments have been carried out using the Eppl simulator which investigate the effect on performance of increasing the system size (speed-up), varying the switch queue and wait-buffer size, increasing the combining level, increasing the interleaving level, and varying the network and memory speed relative to the processor. These experiments are repeated for each benchmark program, and detailed execution statistics are presented for each simulation. A dynamic execution profile for each benchmark program is also presented.

Dedicated to Trish and Benjamin, and in memory of my grandfather Apa.

Acknowledgements

I would like to thank my supervisors Nigel Topham and Roland Ibbett for getting me started and helping me along, and Peter Osmon for enabling me to finish. Many people have provided moral and technical support during the past few years, so thanks also to: Tim Lees, Pawel Paczkowski and Richard Eyre-Todd for putting up with me in the office; Mikeee Norman, Greg Wilson, Lyndon Clarke, Steve Booth and Mike Brown for help with the ECS; Angus Duggan for PostScript and Latex hackery; and Steve Renals for being my buddy and flatmate. The final thanks go to STC Research Ltd. and Dad for their financial support.

Declaration

I declare that this thesis was written by myself, and that the work described herein is my own.

Quotations

"A general purpose multiprocessor should be scalable, *i.e.*, show higher performance when more hardware resources are added to the machine. Architects of such multiprocessors must address the loss in processor efficiency due to two fundamental issues: long memory latencies and waits due to synchronisation events." [1]

"It seems that enough research has already been done in evaluating interconnection networks in isolation. We strongly feel that more work is needed at the system level that includes the interconnection network as a major component." [2]

"An extant microprocessor was chosen as the PME processor. We feel that this was an appropriate design decision in that we were able to concentrate on other design issues instead of designing yet another microprocessor." [3]

"Fully-fledged, cycle-by-cycle simulation of the multiprocessor is very tedious, especially if the goal of the analysis is to understand the performance of the multiprocessor for very large configurations and very large problem sizes." [4]

"Precise estimates of the interaction between components of such a system can only be obtained by simulation at or below the memory reference level, requiring excessive computing resources. Results of detailed simulation will depend strongly on the specific algorithm being simulated, providing results that are not easily generalised." [5]

Contents

Chapter 1

Introduction

An MIMD shared memory multiprocessor consists of a number of processors connected by a network to a number of memories. Each processor can access all the memories, and executes one or more independent processes. The processes communicate and synchronise using variables in the shared memory. Such multiprocessors are attractive for a number of reasons: similar programming and operating system techniques to those found in multiprogrammed uniprocessors can be used, the homogeneity of the processors allows load balancing to be easily achieved using self-scheduling [6], the low inter-process communication cost allows the execution of fine-grain processes, and message-passing interprocess communication can be implemented on top of the shared memory. To be scalable, *i.e.*, to maintain performance with increasing size, shared memory systems must be able to overcome the increasing cost of memory access and inter-process communication as the system size increases. The interconnection requirements of small shared memory systems, with a maximum of 16 processors say, can be satisfied by the combination of a fast shared bus and "snoopy"-caching [7]. For larger systems it is necessary to use a higher performance, more scalable network such as a multistage network [8].

This thesis describes the design and simulation of an MIMD shared mem-

ory multiprocessor system which is named the Edinburgh parallel processor One (Eppl). The Eppl multiprocessor has a conventional "dancehall" architecture with processors on one side of a multistage network connected to an equal number of memories on the other side. The processors have a reduced instruction set (RISC) architecture, and support the concurrent execution of multiple processes (the terms "process", "context" and "instruction stream" are used rather inter-changeably in this thesis) by replicating register files and other resources in each processor. An atomic fetch-and-add instruction is provided for inter-process synchronisation. The network is packet switched, and consists of 2×2 switches connected in an indirect binary n-cube topology. The switches support the combining of load and fetch-and-add memory requests directed at the same memory location. There is no processor cache, local memory, or paged virtual memory, and all memory requests traverse the network to one of the shared memories.

The Eppl design exists on paper and as a simulation program only, and is used as a didactic tool for investigating the components of the design and their interaction in the whole system. No implementation of the design is intended. The main features of the design are the interleaving of processes on each processor, the combining of memory requests in the multistage network, and the use of the fetch-and-add instruction as the synchronisation primitive. These features have been separately included into other MIMD shared memory multiprocessor designs (Section 2.5), but they have not previously been included together in the same system.

The performance of the Eppl design is evaluated by detailed simulation of the complete design at the instruction level. The availability of a powerful computing resource, the Edinburgh Concurrent Supercomputer (ECS), allows the simulation of large Eppl configurations (up to 256 processors) within reasonable time scales. The ECS is a distributed memory multiprocessor constructed from transputers, and the Eppl simulator is written in Occam to execute in parallel on the ECS. The simulated Eppl processors execute instructions from programs

that are loaded into the simulated memories. The programs are written in the Eppl assembly language and assembled into Eppl machine code. Benchmark programs are used to provide workloads for the simulation experiments, these are assembly language implementations of three well known parallel algorithms: matrix multiply, bitonic merge sort and Moore shortest path. The matrix multiply program multiplies two integer matrices, the bitonic merge sort program sorts a sequence of integers, and the Moore shortest path program finds the shortest path from a source node to every other node in a weighted directed graph. All process control and synchronisation in the programs is implemented using the fetch-and-add instruction.

The simulation experiments are performed by executing the benchmark programs with various values of the Eppl system parameters. The simulator records the program execution time and the system execution statistics, and any changes in these can then be related to changes in the system parameters. The simulation experiments described here investigate the effect on performance of: increasing the system size while keeping the program size constant (relative speed-up), varying the switch queue and wait-buffer size, increasing the combining level, increasing the interleaving level, and varying the network and memory speed relative to the processor. A dynamic execution profile of each benchmark program is also presented which shows how the execution statistics vary during the execution of the program.

1.1 Original work

The following original work has been carried out as part of the research leading to this thesis:

- The context-flow processor design of Topham (Section 2.1.4) was extended to a 32-bit instruction set including the fetch-and-add instruction.

- Simulations of a single Eppl processor were carried out to find the performance bounds of the processor, executing synthetic instruction mixes, with increasing interleaving level and memory latency.

- Simulations of the isolated Eppl network were carried out to find the performance bounds of the network, using randomised addresses, with varying switch queue sizes, network loads and hot-spot percentages.

- A parallel simulator of the complete Eppl design was implemented in Occam on a transputer based distributed memory multiprocessor, allowing the simulation of a 256 processor Eppl system on 128 transputers.

- An assembler of the Eppl instruction set was written to produce machine code executable by the simulator.

- Three benchmark programs were written in Eppl assembly code to implement parallel matrix multiply, bitonic merge sort and Moore shortest path algorithms. Existing fetch-and-add based process control and synchronisation mechanisms were used in the programs.

- Simulations of the complete Eppl design executing the benchmark programs were carried out to observe the effect of various system parameters on performance, and to collect dynamic execution traces of the programs.

1.2 Chapter overview

The chapters in the thesis are ordered so that the Eppl design, the Eppl simulator, and the benchmark programs are described before the main results, gathered from the simulations of the complete Eppl design, are presented.

Chapter 2 describes the relevant background and previous research. First, instruction interleaved processor design is introduced, and examples of previous

single processor instruction interleaved designs are described. Second, multi-stage networks are introduced, and aspects of their design including topology and combining are described. Third, synchronisation using fetch-and-add is introduced, and example algorithms for implementing semaphores, barriers, and parallel queues are described. Fourth, performance evaluation of multiprocessor systems is reviewed, and parallel time-driven simulation methodology is introduced. Finally, a number of example MIMD shared memory systems are described.

Chapter 3 describes the Eppl design in three sections on the processor, network switch and memory. The processor section includes a detailed description of the structure and function of the processor pipeline, a description of the instruction set, and simulation results showing performance characteristics of a single processor. The network section includes a description of the network switch functions, such as routing and combining, and simulation results of an isolated network with randomised addressing.

Chapter 4 describes the Eppl simulator. First, the Occam language and the transputer are reviewed briefly. Second, the Edinburgh Concurrent Supercomputer is introduced, and the structure of its transputer domains described. Third, the operation and function of the simulator is described, including how the simulation is parallelised and how the parallel modules communicate. Fourth, the user-interface and commands available are described. Fifth, the clocking of the modules is described. Finally, performance results of the simulator itself are presented.

Chapter 5 describes the three benchmark programs. First, the choice of algorithms is discussed. Second, the process control mechanisms are described. Each program is then described in turn, showing how it was implemented in the assembly language of the Eppl processor.

Chapter 6 describes and presents results from the simulation experiments which investigate the effect on system performance of system size, queue and

wait-buffer size, combining level, interleaving level and relative processor speed. A dynamic execution profile for each program is also presented.

Chapter 7 summarises the research and results described in the thesis. Improvements to the design and simulator, and further research topics, are suggested.

Appendix A contains Eppl assembly code listings for the benchmark programs which are described in Chapter 5.

Appendix B contains the detailed execution statistics from the simulation experiments which are described in Chapter 6.

Chapter 2

Background

This chapter provides the relevant background information for the chapters which follow. The first section introduces instruction interleaving, and gives examples of processor designs that implement instruction interleaving. The second section introduces packet-switched multistage interconnection networks, and discusses the topology, routing of packets, and contention in such networks. The third section introduces synchronisation using the fetch-and-add instruction, and describes some synchronisation algorithms implemented with fetch-and-add. The fourth section discusses performance evaluation of multiprocessor designs, and describes parallel time-driven simulation. The final section briefly describes some example MIMD shared memory multiprocessor designs.

2.1 Instruction interleaving

Pipelining increases processor throughput, defined as the number of instructions executed in unit time, by reducing the machine cycle time and allowing concurrent execution of a number of instructions. Generally processor pipelining is divided into two levels: instruction pipelining, and function or arithmetic pipelining [9, 10, 11]. Instruction pipelining divides the normal instruction execution sequence

into a number of stages, typically: instruction fetch, instruction decode, operand fetch, instruction execution, and result store. Current microprocessors are usually pipelined at this level. To achieve maximum performance particular function blocks, such as a floating point arithmetic unit, are further pipelined. Function pipelining is found in vector supercomputers for example.

2.1.1 Single instruction stream pipelines

In a conventional instruction pipeline, which is executing a single instruction stream, each stage of the pipeline contains an instruction from that stream (or is empty). To utilise the pipeline efficiently there must be a sufficiently long sequence of instructions to fill the pipeline. Due to discontinuities in the instruction stream only pipelines with a small number of stages can be utilised efficiently. Three possible causes of discontinuities are: branch dependence, data dependence and memory latency.

When a branch instruction occurs and the branch is taken, assuming that instructions are not prefetched from the branch address, then instructions in the pipeline after the branch have to be flushed from the pipeline and new instructions fetched from the branch address. To reduce the loss of performance caused by branch dependence a number of methods such as branch prediction, branch target buffers and delayed branching are used [9, 12, 13, 11].

Sequential instructions in an instruction stream are likely to have data dependencies between them. These can lead to reduced utilisation because reading of operands may have to be delayed to satisfy the dependency. Such data dependence conflicts occur particularly in processors with a long pipeline, or with multiple pipelines of differing lengths in which the order of instruction completion must be maintained. To ensure that the dependencies are satisfied methods such as register scoreboarding, internal forwarding and compiler data dependence analysis are used [9, 11, 14].

Pipeline throughput is limited by memory latency, as the pipeline can only

execute as fast as the instructions are supplied. In a single processor system the memory latency is usually equal to the memory access time. In multiprocessor systems memory requests generally have to traverse an interconnection network, so the memory latency is higher due to network delay and contention. Methods of reducing memory latency include the use of caches, local memory, separate data and instruction busses (Harvard architecture), and decoupling memory access from instruction execution [15, 16].

One way of side-stepping these dependency and latency problems, and increasing the utilisation of the pipeline, is to implement a multiple instruction stream pipeline.

2.1.2 Multiple instruction stream pipelines

In an instruction interleaved pipeline instructions from more than one instruction stream are executed concurrently. If only one instruction from each instruction stream is allowed to be executed at any one time, then each stage of the pipeline contains an instruction from a different instruction stream (or is empty). Effectively the pipeline switches context to a different instruction stream every cycle. Instruction interleaving is also variously known as: micro-multiprogramming [17], virtual-processing [18], context-flow [19], circulating-contexts [20], state multiplexing [21], and multi-threading [1].

The main aim of instruction interleaving is greater pipeline utilisation. If each instruction stream is restricted to executing a single instruction at a time, then the dependency problems described above do not occur. Because each stage of the pipeline contains an instruction from a different instruction stream, there are no branch dependencies and thus no branch penalty. Also, because each instruction in an instruction stream is completed before the next one is issued, there are no data dependencies. So two possible causes of discontinuities in the pipeline have been removed, and as long as sufficient instruction streams are provided the pipeline can operate at maximum throughput. The multiple functional units

may also be better utilised if the instruction streams execute different instruction types.

Although instruction interleaving does nothing to reduce memory latency, it does allow the pipeline to tolerate higher memory latency. This is because memory access and execution can be overlapped, since while some instruction streams wait for memory accesses to complete others can be executed. Memory access and instruction execution are thus de-coupled. If the memory system is also pipelined, for instance in shared memory multiprocessor systems that have pipelined multistage networks, then this effectively extends the length of the execution pipeline and more instruction streams must be executed to achieve maximum pipeline utilisation.

To implement instruction interleaving starting with a conventional single instruction stream processor requires the addition of extra hardware (registers and other function blocks) to the pipeline. Each instruction stream requires extra state information, such as a process number in addition to the usual registers and flags, and all the state information must be replicated for each instruction stream. Extra logic is required to control the multiple instruction streams, but conversely none of the logic to handle branch and data dependency is required. To maintain a high pipeline utilisation the processor must support a sufficient number of instructions streams to fill the pipeline, and the operating system and applications programs must be sufficiently parallel to keep all the instruction streams busy.

Data-flow systems [22] have similarities to instruction interleaved systems, and also use the arguments for greater processor utilisation given above [23, 1]. A process (instruction stream) in a data-flow machine is typically a single instruction, and processes are not associated with any one processor. An advantage of data-flow systems over the instruction interleaved systems described here is that they also provide implicit synchronisation between processes. The hardware implementation and programming of data-flow systems is not straightforward though [24].

2.1.3 Performance of instruction interleaved pipelines

Assuming a simple linear pipeline with P stages executing p instruction streams ($p \leq P$), then (ignoring start-up time):

pipeline throughput $= p/P$ instructions per cycle

instruction stream throughput $= 1/P$ instructions per cycle

The pipeline throughput is the number of instructions executed per cycle, and the instruction stream throughput is the number of instructions executed per stream per cycle. The maximum pipeline throughput is achieved when $p = P$. If the number of streams is greater than the number of stages ($p > P$), then the throughput per stream will be reduced as there will be contention for the pipeline.

The performance of the HEP processor (Section 2.5.1), which supports multiple instruction streams, has been characterised using the r_∞ (maximum throughput) and $n_{1/2}$ (half-performance vector length) parameters that have previously been applied only to vector and array processors [25]. The half-performance vector length in an instruction interleaved system is the number of instruction streams required to achieve half the maximum throughput. Topham [26] also describes a simple analytical model of a context-flow processor. These models provide similar results to the actual performance results of a single HEP processor [27], which show that pipeline throughput increases linearly as more instruction streams are added, up to the maximum defined by the pipeline length. Generally, the effective pipeline length is greater than the actual pipeline length because of memory access latency.

A number of single processor designs which support instruction interleaving are summarised below.

2.1.4 Example instruction interleaved processor designs

In the **shared resource processor** design [18], up to 32 concurrently executing instruction streams share 7 pipelined functional units and a memory interface.

Each instruction stream has its own instruction decoder, register set and instruction and data caches. Contention for the functional units is minimised by allocating time-slices during which each instruction decoder can issue an instruction. The instruction set includes a spawn instruction for starting a new instruction stream and fetch-and-add instructions for synchronisation. Simulations of the design indicate that a high processor throughput and functional unit utilisation can be achieved.

The **P*** or multiple stream registerless shared resource processor design [28], consists of fetch, decode and execute pipeline stages combined with special associative queues which buffer and merge instruction streams. Each instruction stream state consists of a program counter, instruction register and status bits. Instructions are memory to memory and there are no explicitly addressable registers. There are five memory ports which can all make concurrent requests, so multiport memory is required. The design provides high utilisation of individual processor components, decouples processor throughput from memory access time, and the additional hardware cost per instruction stream scales well.

The **virtual multiprocessor** design [29], is a pipelined processor supporting multiple instruction streams. Various pipeline layouts and control strategies were compared for this design. A straight through pipeline gives the best cost/performance ratio for varying degrees of multiprocessing. A single chip implementation of the virtual multiprocessor was also considered [30]. The use of a central register file was compared to incorporating the registers into the pipeline stages. The latter allows the use of dynamic logic and minimises interconnections, but the cost of the processor will depend heavily on the size of the register set, since it is replicated in each pipeline stage. Instruction traces were used to find an optimum register set size for a central register file, and adding registers was shown to be substantially less effective in improving performance than increasing the level of multiprocessing.

The **cyclic pipeline computer** design [31], is a pipelined processor support-

ing multiple instruction streams which is to be implemented in Josephson-junction technology. Because Josephson-junction gates are self-latching there is no need for latches between pipeline stages, and this allows the pipeline to be clocked using only a single-phase clock. The memory is also to be implemented using Josephson-junction devices, and has a four stage pipeline. A silicon version of the processor (FLATS-2) is being implemented.

The **VMP** or virtual multiprocessor design [32], is based on the IBM370 CPU and uses multiprogramming at the instruction level. Each virtual CPU (instruction stream) has a separate register set, but shares all other processor resources. The optimum number of virtual CPU's in the design is dependent on the average instruction completion time.

The **Context-flow** processor design [19, 26, 33], is a VLSI pipelined RISC processor supporting multiple instruction streams. A proposed implementation [34] is based around an off-the-shelf pipelined floating-point ALU, with a multiport register file, three merging queues, and some custom VLSI implementing the control unit. Up to 64 concurrently executing instruction streams can be spawned on demand. The original context-flow proposal also included a message-passing interprocess communication mechanism in which processes communicate through special channels in memory. The Eppl processor described in Chapter 3 is based on the context-flow processor design.

Because instruction interleaved processors can execute efficiently even with high memory latencies they are obvious candidates for multiprocessor systems. Two shared memory multiprocessor designs that support instruction interleaving, the HEP and CCMP, are described in Section 2.5. There are other multiprocessor designs which although not strictly instruction interleaved do support fast context-switching of processes within the processor [35, 36]. There are also a number of hybrid multiprocessor designs which combine features of message-passing and data-flow systems [37, 38, 39], and have similarities to instruction interleaved systems.

This concludes the background section on instruction interleaving; the next section introduces multistage networks.

2.2 Packet-switched multistage networks

A multistage network consists of a number of stages of small crossbar switches in which the outputs of one stage are connected to the inputs of the next stage according to some permutation. There are many flavours of multistage network [8], and only rectangular, two-sided, uni-directional, packet-switched, blocking networks are considered here. Such a network with n inputs and $k \times k$ switches has $\log_k n$ stages with n/k switches in each stage, the same number of outputs as inputs (rectangular), the outputs are on the opposite side of the network to the inputs (two-sided), messages are transmitted across the network in packets using store-and-forward routing (packet-switched), and there is only one path between each input and each output in the network (blocking). The indirect binary n-cube [40] and the omega [41] networks, shown in Figure 2.1, are examples of this type of network. A number MIMD shared memory multiprocessor systems use this kind of multistage network to connect the processors to the memories, for example the RP3, HEP, CHoPP and Monarch (see Section 2.5), and also the Eppl design described in the next chapter.

To restrict the discussion further only relevant aspects of multistage network design and performance are discussed here. In particular, VLSI implementation [42, 43], and fault-tolerance/error-correction [44, 45] are not reviewed.

2.2.1 Network topology

There are many different classifications of multistage networks. Some of these describe classes of networks such as SW-banyan [46], delta [47] and cube [44]; and some describe a particular network topology such as the indirect binary n-cube, omega, baseline [48] and flip [49]. A number of these network topologies

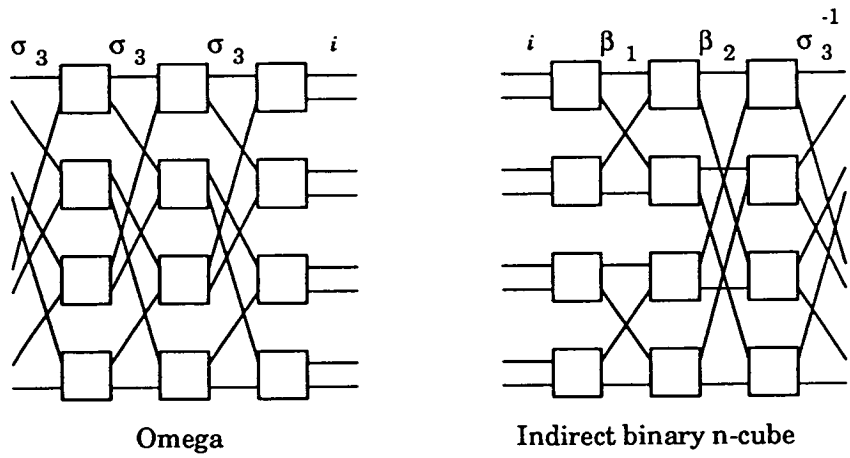


Figure 2.1: Omega and indirect binary n-cube multistage network topologies

have been shown to be isomorphic [48].

The topology of a multistage network can be defined by the degree of the switches in the network, and how the switches are connected. These connections can be conveniently described using permutation functions [14], some of which are defined below. The argument to each function is the binary representation of a network port input number x.

$$x = \{b_n, b_{n-1}, \ldots, b_{k+1}, b_k, b_{k-1}, \ldots, b_2, b_1\}$$

The **identity** permutation is i, where:

$$i(x) = x$$

The **shuffle** permutation σ_k is defined as the cyclic left shift of the k least significant bits of x.

$$\sigma_k(x) = \{b_n, \ldots, b_{k+1}, b_{k-1}, \ldots, b_1, b_k\}$$

The **inverse shuffle** permutation σ_k^{-1} is defined as the cyclic right shift of the k least significant bits of x.

$$\sigma_k^{-1}(x) = \{b_n, \ldots, b_1, b_k, \ldots, b_2\}$$

The **butterfly** permutation β_k exchanges the first and k^{th} bits of x.

$$\beta_k(x) = \{b_n, \ldots, b_1, b_{k-1}, \ldots, b_k\}$$

As shown in Figure 2.1, the stages of the omega network are connected by repeated shuffle permutations (where $k = \log n$), and a final identity permutation. The stages of the indirect binary n-cube are connected by butterfly permutations (where k varies with the stage), and a final inverse perfect shuffle.

In a shared memory multiprocessor system with a dancehall configuration, two of the multistage networks described above are required: one (the request network) to route the request packets from the processors to the memories, and one (the acknowledge network) to route the acknowledge packets back from the memories to the processors.

2.2.2 Packet routing

Packet switching allows packets to be pipelined through the network, and the network cycle time is only limited by the time it takes to transmit a packet from one stage to the next (routing and wire delay). A packet typically consists of a number of fields including: a type field, a source address field, a destination address field and a data field. In a memory request packet the type field defines the type of memory access, the source address field contains the number of the requesting processor, and the destination field contains a memory number and an offset within that memory. In a memory acknowledge packet the destination field contains the processor number to which the acknowledge packet should be routed back.

Routing is controlled at each switch by a number of bits from the destination address [50]. If the network has $k \times k$ switches, then $\log_2 k$ bits are required in each switch to specify where the packet is routed. Therefore $\log_2 k \times \log_k n$ bits are required to route a packet through an n input network. To maintain the same

numerical ordering of inputs and outputs, the routing bits must be interpreted from the most significant end of the destination address in an omega network, and from the least significant end in an indirect binary n-cube network. After the packet has been routed in the switch, the destination address can simply be shifted $\log_2 k$ places to discard the used bits. The return address can be included in the request packet before it is transmitted, or more conveniently, it can be generated as the packet moves through the network [51].

2.2.3 Routing conflicts

If two packets in a switch are addressed to the same output, then some conflict resolution strategy is required. Possible options are to block, re-route or discard one of the conflicting packets. Blocking is discussed below. Re-routing was proposed for the Burroughs FMP network [52, 53], and implemented in the HEP network. The choice of which packet proceeds depends on the priority algorithm used, for example: random, round-robin, or adaptive priority [54].

Packets that have been blocked must be queued in the switch until they can be routed onwards. Incorporating queues in a switch improves network performance because it allows packets from a preceding switch to progress even when a packet in the current switch is blocked. The queues can be positioned either at the inputs or outputs of the switch, or incorporated into the crossbar [55]. The network throughput increases as the queue size is increased, but the network latency increases as well, and therefore small queues of length 4 to 8 packets are optimal [56, 57].

2.2.4 Hot-spot contention

Apart from the existence of routing conflicts, network performance can be further reduced by contention for the memories which occurs when a number of memory requests are addressed to the same memory bank. So that sequential memory

requests are not directed at the same memory, logical memory addresses can be interleaved or hashed to produce the physical address. The RP3 and Monarch designs both use hashing, which has the effect of randomising the memory access pattern. This method does not reduce contention when the memory requests are addressed to the same location however.

Memory locations which have a statistically significant higher percentage of requests directed at them than other locations are termed hot-spots. As the number of requests addressed to the hot memory location increase eventually a point is reached where more packets arrive than the memory can service per cycle, and the requests are queued. Ultimately an effect called tree saturation [58] occurs, in which the switch queues from the hot memory back to the network inputs are all full. Then, not only the traffic directed at the hot-spot is blocked, but all other traffic which must pass through those switches. A hot-spot can arise from access to shared variables [58, 59, 60], or from block transfers [61]. Simulations show that network performance degrades soon after a hot-spot becomes active, and that the effects of the hot-spot remain for some time after it is no longer active [62].

A number of ways to reduce hot-spot contention have been proposed. A method of combining memory requests to the same location is described in the following section. Other methods include moving the location of the hot-spot between memories [63]. Such a roving memory location (RML) is implemented as a special packet which moves from memory to memory, satisfying any requests for the hot-spot which have been queued in the memory. Instead of implementing hardware combining, accesses to shared variables could be implemented in software [64]. Each shared variable is then replaced by a tree of variables which are evenly distributed among the memories. A different network architecture could also be used, for instance the circuit-switched multistage network of the BBN Butterfly [65, 66]. This network is non-blocking, and therefore no situation analogous to tree saturation occurs, although there is still contention for access

to the network.

2.2.5 Combining

The Ultracomputer, RP3, Monarch and CHoPP multiprocessor designs all include some form of combining. Only a brief description of the combining mechanism is given here as the details depend on the particular switch design [67, 68, 69, 70]. Also the combining mechanism used in the Eppl design is similar to that used in the Ultracomputer and RP3, and is described in detail in Chapter 3.

Combining [71, 72] occurs in the network switches. If two request packets addressed to the same memory location meet in a switch they can be combined. The combining operation involves updating the packets' data fields, depending on the memory request type, and then one of the packets is forwarded and the other packet is stored in the switch. When the returning acknowledge packet passes back through the switch from memory, the stored packet is decombined. Again, the specific decombining operation depends on the request type.

A request packet may be combined more than once on its way to memory. The number of times a packet can combine in each switch is given by the combining level [73], and depends on the switch implementation. With a combining level k, or k-way combining, a packet can combine $k-1$ times in each switch. For example with 2-way combining, also called pairwise combining, a packet can only combine once per switch. An initial performance analysis of pairwise combining [58] showed that combining improves network performance, and that tree-saturation can be prevented. Subsequent research [73, 74] suggests that, for large networks and heavy loads, pairwise combining is not enough to reduce tree saturation effectively, and 3-way combining must be used.

Combining is particularly effective when used in conjunction with the fetch-and-add operation described in the following section, and was originally proposed in the Ultracomputer design to allow simultaneous fetch-and-add memory accesses to be carried out efficiently [51].

2.3 Synchronisation with fetch-and-add

In a shared memory multiprocessor interprocess communication and synchronisation occurs using shared variable's. Synchronisation primitives, such as mutual exclusion and condition synchronisation [75], are implemented using atomic read-modify-write operations such as fetch-and-add or test-and-set.

The fetch-and-add instruction has been used in single processor machines such as the IBM370, and in shared memory multiprocessor designs such as the Ultracomputer, RP3, Monarch and CHoPP. The instruction `f&a(v,x)` adds the given value `x` to the variable `v`, and returns the variables previous value. Fetch-and-add has efficiency advantages over the simpler test-and-set instruction, for example it can be used directly to update a shared variable, so removing the need for locking. A number of synchronisation and other algorithms using fetch-and-add have been developed in conjunction with the Ultracomputer and RP3 designs [76, 77]. Versions of some of these algorithms implementing semaphores, barrier synchronisation, test-modify-retest functions, and a parallel queue are described below. These algorithms are used subsequently in the Eppl benchmark programs described in Chapter 5.

2.3.1 Semaphores

Implementations of the `p()` and `v()` semaphore operations [75], are shown in Figure 2.2 using a pseudo-C syntax. The function `f&a(s,x)` behaves as defined above. The address of the semaphore is passed in argument `s`.

The `p()` function first waits for the semaphore's value to be greater than zero. The semaphore is then decremented using f&a, and if the returned value is greater than zero the function returns. Otherwise the semaphore is incremented using f&a, and the function loops back to the beginning. This implementation of the `p()` function is blocking, since the process calling the function will busy-wait until the function succeeds. The `v()` function increments the semaphore using

```
int p(s)
int *s;{
   int t;

   while(1){
      while(*s<=0);
      if((t=f&a(s,-1))>0)
         return(t);
      else
         f&a(s,1);}}

int v(s)
int *s;{
   return(f&a(s,1));}
```

Figure 2.2: P and V semaphore functions

f&a, and returns.

The test of the semaphore's value at the start of the p() function, before the first f&a, is necessary to avoid the possibility of livelock. Assume a p() function without this initial test. If a large number of processes all executed p() on a semaphore in parallel, then the semaphore would have a large negative value after the f&a. If another process executed v() at the same time, the semaphore would still have a negative value. Therefore all the processes executing p() would be locked out, perhaps indefinitely.

The main application of these semaphore functions is to implement mutual exclusion, for instance to lock a shared variable during an update. If the shared variable has an associated semaphore s, which is initialised to 1, then a process calls p(s) to lock and v(s) to release the variable. If the update is simply an addition, then it may be possible to use f&a directly on the shared variable without explicitly using a lock.

```
void join(count,maximum)
int *count,maximum;{
   f&a(count,1);
   while(*count<maximum);}
```

Figure 2.3: A join function

```
void barrier(count,maximum)
int *count,maximum;{
   int t;

   t=(*count<maximum);
   if(f&a(count,1)==(2*maximum-1)) *count=0;
   while((*count<maximum)!=t);}
```

Figure 2.4: A barrier function

2.3.2 Barrier synchronisation

A barrier synchronisation is used to ensure all parallel parts of a computation have completed before the next is scheduled [75, 78]. If the barrier is used only once then a simple join (which is counterpart to a fork) can be used, such as the function shown in Figure 2.3. The count variable is initialised to zero. Processes calling `join()` increment the count, and are blocked until the count equals the maximum.

If the barrier is to be re-used, for instance if it separates iterations of a loop, then care must be taken that there is no possibility of processes from different iterations overtaking each other and thus causing deadlock. The barrier algorithm shown in Figure 2.4 is a re-usable implementation [79], allowing the same counting variable to be used in subsequent barriers. The count variable is initialised to zero. As processes call `barrier()` they are blocked until the count equals the maximum, and then `maximum` processes are allowed to pass. When the processes reach the barrier again in the next iteration, they are blocked until the count equals `2*maximum-1`. Then `maximum` processes are again allowed to pass, and the

```
int tir(semaphore,increment,maximum)
int *semaphore,increment,maximum;{
   int t;

   while(1){
      while(*semaphore>=maximum);
      if((t=f&a(semaphore,increment))<maximum)
         return(t);
      else
         f&a(semaphore,-increment);}}
```

Figure 2.5: Test-increment-retest function

```
int tdr(semaphore,decrement)
int *semaphore,decrement;{
  int t;

  while(1){
     while(*semaphore<=0);
     if((t=f&a(semaphore,-decrement))>0)
        return(t);
     else
        f&a(semaphore,decrement);}}
```

Figure 2.6: Test-decrement-retest function

count is reset.

2.3.3 Test-modify-retest functions

The test-increment-retest (TIR) and test-decrement-retest (TDR) functions are used frequently in fetch-and-add based algorithms [76, 71]. Blocking implementations of these functions are described here, and are used in the parallel queue algorithm described below.

The `tir()` function shown in Figure 2.5 has three arguments: the semaphore, the value of the increment, and the maximum the semaphore can reach. The function first tests the value of the semaphore to prevent livelock (as explained

above), and then increments the semaphore. If the returned value is below the maximum then the function succeeds. TIR can be used to spawn a number of processes, giving each one a unique index number (the value returned by the function). This index number can then be used for accessing parts of a shared data structure such as an array.

The `tdr()` function shown in Figure 2.6 has two arguments: the semaphore and a decrement. If the value of the semaphore is greater than zero, the semaphore is decremented and the function returns, otherwise it is blocked. TDR is similar to the `p()` function defined above, except that the decrement is passed as an argument.

2.3.4 Parallel queue algorithm

Blocking implementations of parallel insert and delete functions for a parallel FIFO queue [76] are presented here. The queue is implemented using a circular buffer, and consists of a header and an array of entries as defined below:

```
struct{
   int insert, delete, entry;} entrystruct

struct{
   int size, upper, lower, insert, delete; /* header */
   struct entrystruct entries[];} queuestruct
```

The queue header contains five fields: the queue size, an upper bound, a lower bound, an insert pointer and a delete pointer. The upper bound contains the number of items in the queue plus the number of active insertions. The lower bound contains the number of items in the queue minus the number of active deletions. The insert pointer points to the tail of the queue. The delete pointer points to the head of the queue. Each queue entry has three fields: an insert semaphore, a delete semaphore and a data value. The insert semaphore indicates whether the data field is empty and available for insertion. The delete semaphore indicates whether the data field is full and available for deletion. These semaphores are

```
void insert(queue,value)
struct queuestruct *queue;
int value;{
   int t;

   tir((queue->upper),1,(queue->size));
   t=f&a((queue->insert),1)%(queue->size);
   p(queue->entries[t]->insert);
   queue->entries[t]->entry=value;
   v(queue->entries[t]->delete);
   f&a((queue->lower),1);}
```

Figure 2.7: Insert function

```
void delete(queue,value)
struct queuestruct *queue;
int *value;{
   int t;

   tdr((queue->lower),1);
   t=f&a((queue->delete),1)%(queue->size);
   p(queue->entries[t]->delete);
   *value=queue->entries[t]->entry;
   v(queue->entries[t]->insert);
   f&a((queue->upper),-1);}
```

Figure 2.8: Delete function

necessary to ensure correct synchronisation when there is a concurrent insertion and deletion of the same entry, or when there are two concurrent insertions of the same entry.

The `insert()` function appends the given data value to the tail of the queue. The `tir()` function is first called on the upper bound, and blocks until there is room in the queue for insertion. Then the insert pointer is incremented, and the offset of the entry in the queue calculated. Next `p()` is called on the entry's insert semaphore. When this succeeds the data value is written, `v()` is called on the delete semaphore, and the lower bound is incremented.

The `delete()` function removes the entry at the head of the queue. First the `tdr()` function is called on the lower bound, which blocks until an item is available to delete. Then the delete pointer in incremented, and the offset of the entry in the queue calculated. Next `p()` is called on the entry's delete semaphore. When this succeeds the data value is read, `v()` is called on the insert semaphore, and the upper bound is decremented.

The queue can be used as a parallel data structure or to schedule processes for example. For process scheduling each queue entry can point to a process control block, and processes are removed from the queue when a processor becomes free (self-scheduling). Only individual processes can be scheduled this way, which is not efficient for scheduling a large number of identical processes. To do this efficiently a multi-queue algorithm can be used, in which each queue entry also contains a multiplicity count [76]. Unfortunately the parallelism of the multi-queue has to be restricted, compared to the above algorithm, to ensure correctness.

2.4 Performance evaluation of multiprocessors

To evaluate the performance of multiprocessor designs the same techniques used for uniprocessor designs are applied [80], but can be complicated by the size, parallelism and synchronisation inherent in the design to be evaluated. Performance evaluation methodologies are divided into analytical and simulation based techniques. The application of parallelism to increase simulation speed has required the development of new simulation algorithms.

2.4.1 Analytical models

Analytical models try to capture the behaviour of a system in a set of equations, and are frequently based on a stochastic analysis of the system's changes of state (for instance using queueing network theory) [80, 81]. Because such models are

quite general they have a wide applicability, and may be used to evaluate similar systems without much change. Assuming closed-form equations can be derived for the model, then it can be solved efficiently and will not require excessive computing resources, although many iterations may be needed before the model settles down and produces reliable results. The size of parallel systems complicate the solution of queueing network models due to the combinatorial explosion in the number of possible states, and the correct modelling of synchronisation is difficult because of the loose-coupling assumed in some of the models. Because of their formulation, analytical models can generally only provide information about the average steady-state behaviour, or performance bounds, of the system being modelled. To validate an analytical model also usually requires comparing the model to a more detailed simulation of the same system.

Various stochastic models including queueing networks have been used to model multiprocessor network performance [47, 57, 2]. These models assume that memory addresses are uniformly distributed and that all addresses are independent, and provide results for best-case network throughout and delay under these conditions. To evaluate the effect of a memory hot-spot the models have been extended to account for a fraction of the memory requests, the hot-spot percentage, being directed at the hot-memory [58, 73, 74]. Stochastic Petri nets have been used to model bus-based multiprocessor systems [81]. The RP3 project developed a spreadsheet program which used queueing analysis to show network performance for various configurations [5].

2.4.2 Simulation models

Simulation models try to capture the behaviour of a system in the code of the simulator program. A simulation model is more specific to the system being evaluated, and is also more computationally intensive than an analytical model. There is no restriction on what can be modelled, allowing dynamic effects to be investigated, and simulation models can also be used for functional verification

as well as performance evaluation.

Simulation models are generally divided into instruction, register-transfer and gate level models. As the model becomes more detailed the amount of structural information in the model increases. General simulator systems allow the modelled design to be specified using a hardware description language, which may be a high level description such as ISP [82] and VHDL [83], or in terms of function blocks and registers, or logic gates, depending on the level of the simulation. Custom simulators are usually implemented using a high-level language in the form of a behavioural description. Specific language features may guide the choice of language, for example using Occam to explicitly describe the parallelism in a system [84, 85].

The input to instruction level simulators consists of either synthetic test data, which may be stochastically generated, or actual program code, or traces of instructions or memory accesses taken from existing machines. The latter is traditionally referred to as trace-driven simulation. To use a trace-driven simulator the appropriate traces must first be available, which may not be the case if the system to be simulated uses novel instructions for example. A number of systems to produce multiprocessor traces have been developed [80, 4, 86, 87]. An example is PSIMUL which is a simulation tool developed in conjunction with the RP3 project [87]. PSIMUL simulates the execution of a shared memory machine using instruction traces collected from the execution of parallel applications on a uniprocessor. The memory reference traces generated by PSIMUL have been used to analyse memory reference behaviour [88], to drive multiprocessor and network simulations [89], and to analyse multiprocessor cache performance [90].

A number of projects involving the design of a multiprocessor system have developed simulators for performance evaluation. For example the NYU Ultracomputer was first simulated using the Washcloth simulator [91] which ran on a CDC6600. Rather than simulating a processor, the CDC processor itself was used, and programs could be compiled from a high level language and executed

directly. Special instructions were added to implement fetch-and-add. A number of different machine models were simulated, including the paracomputer model, and a model with a multistage network. An extension to the simulator included tag-bits in each memory word, to compare the efficiency of tag-bits to fetch-and-add synchronisation.

2.4.3 Parallel simulation

To increase simulation speed, and allow larger systems to be simulated, parallel simulators have been developed to run on multiprocessor machines. Peacock *et al* [92] have defined a taxonomy of distributed simulation (which could also be applied to shared memory simulation), in which simulators are divided depending on how simulation time is treated. In an event-driven simulation the simulated system changes state at varying, unpredictable simulation time intervals, whereas in a time-driven simulation the state changes occur at constant time intervals. The simulation is further defined as being tight if each component of the simulation has exactly the same value of simulation time at any instant, or loose if the components may have differing values.

In an event-driven simulation the simulated system changes state in response to events. Because these events can arrive out of order they must first be ordered into the correct sequence. There has been much research on algorithms for improving the efficiency of distributed event-driven simulators [92, 93, 94, 95, 96]. The generality of event-driven simulation systems allows them to be used in simulating a variety of different systems. Parallel logic gate simulators have been implemented using distributed event-driven simulation on transputer networks [97, 98].

Algorithms for time-driven simulation have not received an equivalent amount of attention to those for event-driven simulation. This may be because time-driven simulation is less general, and has been used mostly for bespoke simulations. A shared memory time-driven simulation has been used to simulate a

multistage interconnection network on a shared memory multiprocessor [99, 100]. After each network cycle the processes simulating the network switches are synchronised using a barrier. In a distributed time-driven simulation the simulator modules synchronise using message-passing. This allows more parallelism than a shared memory time-driven simulation using barrier synchronisation, because connected modules synchronise directly by the exchange of packets. The Eppl simulator described in Chapter 4 uses a loose distributed time-driven simulation methodology.

2.5 Example shared memory multiprocessors

A number of MIMD shared memory multiprocessor designs have been referred to and are now described briefly below. These designs are relevant because they include either combining multistage networks and/or instruction interleaved processors. The descriptions are brief and only include gross architectural features such as processor and network configuration, and synchronisation mechanisms.

2.5.1 Heterogeneous element processor (HEP)

The HEP [101] is a shared memory MIMD multiprocessor machine consisting of up to 16 instruction interleaved processors connected to memories by a packet-switched multistage network. Each processor, shown in Figure 2.9, can support up to 1024 instruction streams, and contains 8 pipelined functional units, 2048 general-purpose registers, 4096 constant registers and 4 Mbytes of local instruction memory. The network is constructed from bi-directional three port switches, and can re-route blocked packets. Each processor also has a direct connection to one of the memories (called its home memory). Synchronisation is through full/empty bits in the memory and processor registers, using blocking load and store instructions.

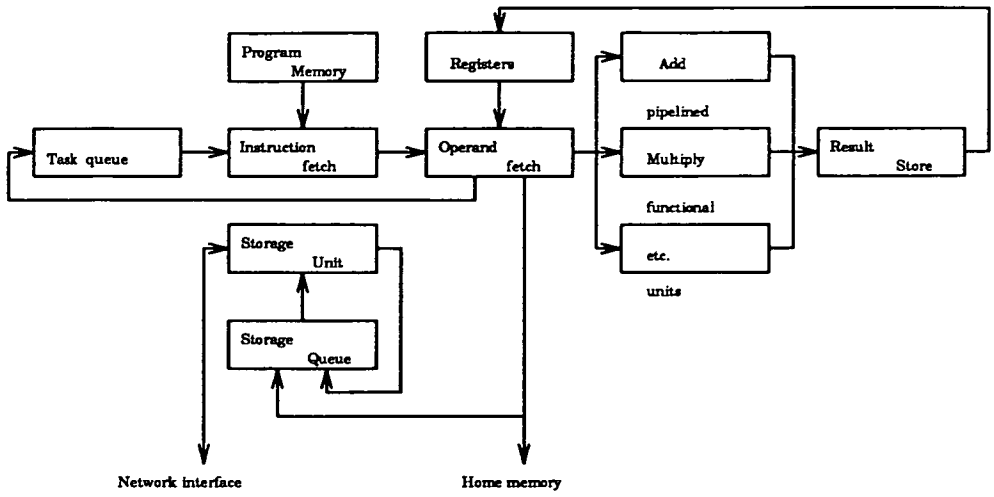


Figure 2.9: Block diagram of HEP processor

2.5.2 Circulating context multiprocessor (CCMP)

The CCMP [20, 102, 63] is an MIMD shared memory multiprocessor design supporting extensive pipelining and multiple instruction streams (processes). The CCMP design shown in Figure 2.10 is split up into home modules, memory modules and execution modules which are connected by buffered singlestage cube networks. The instruction counter and other context of each process is stored in the process's home module. The execution modules contain an instruction decoder and an execution pipeline, and can execute instructions from any process, allowing automatic dynamic load balancing. Instruction execution starts with a process fetching an instruction from memory and then the operands specified in the instruction, then the process is routed to one of the execution modules and the instruction is executed with the result being stored in memory or a register in the process's home module. Synchronisation is based on an atomic swap operation.

2.5.3 Columbia homogeneous parallel processor (CHoPP)

The CHoPP-1 [36] is an MIMD shared memory multiprocessor machine which consists of up to 16 processors connected to a "conflict free" memory. The

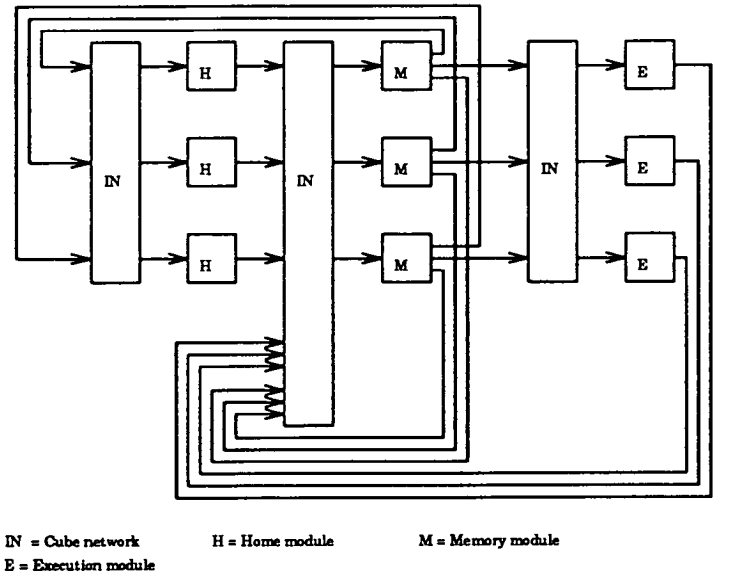


Figure 2.10: Overview of a closely coupled CCMP configuration

CHoPP-1 design is more closely related to the GEM architecture of Cohn [103], rather than the original CHoPP proposal [104]. The processors have a very long instruction word (VLIW) architecture, with each instruction being 256 bits long. Each processor can store the context of up to 64 processes, and a context switch within the processor takes three cycles. The execution units are not instruction interleaved however. Synchronisation is achieved using 2-bit memory tags and special load and store instructions. Processor registers are also tagged, allowing asynchronous memory accesses. There is also a compare-and-swap instruction and a fetch-and-add instruction, with fetch-and-adds being combined in the network.

2.5.4 BBN Monarch

The Monarch [105] is an MIMD shared memory multiprocessor design consisting of up to 65K processors connected to memories by a synchronous hybrid multistage network. The processors are 64-bit with a RISC architecture, have an instruction cache, and overlap memory access with instruction execution to mask memory latency. The network is non-rectangular (more processors than memory

modules), with multiple paths between each source and destination, and uses a hybrid switching strategy. Memory requests are submitted synchronously by the processors, allowing efficient implementation of read combining. Memory words and processor registers have an 8-bit tag, which is used to implement garbage collection algorithms and interprocess synchronisation.

2.5.5 NYU Ultracomputer

The Ultracomputer [106] is an MIMD shared memory multiprocessor design consisting of up to 4K processors connected to memories by a combining omega multistage network. The major contribution has been the development of the fetch-and-add synchronisation primitive and network combining mechanisms. Other areas of investigation include synchronisation algorithms using fetch-and-add, operating system issues such as process scheduling and memory management, and application code development. A number of bus-based prototypes of the Ultracomputer have been constructed using off-the-shelf processors, and are used for software development, particularly a parallel UNIX[1] implementation. The VLSI design of combining switches is also in progress. Much of the design has been realised in the RP3 described below.

2.5.6 IBM Research parallel processor prototype (RP3)

The RP3 [107] is a shared memory MIMD multiprocessor machine consisting of up to 512 processor-memory elements (PME's), connected by two multistage networks. Each PME, shown in Figure 2.11, contains a 32-bit RISC microprocessor, a 32 Kbyte cache and 4 Mbytes of memory. Memory stores are executed asynchronously allowing overlap of memory access and execution. Two networks are provided, one is a high-bandwidth bipolar non-combining banyan network with 128 inputs, and the other a low bandwidth CMOS combining omega network with

[1] UNIX is a trademark of AT&T Bell Labs.

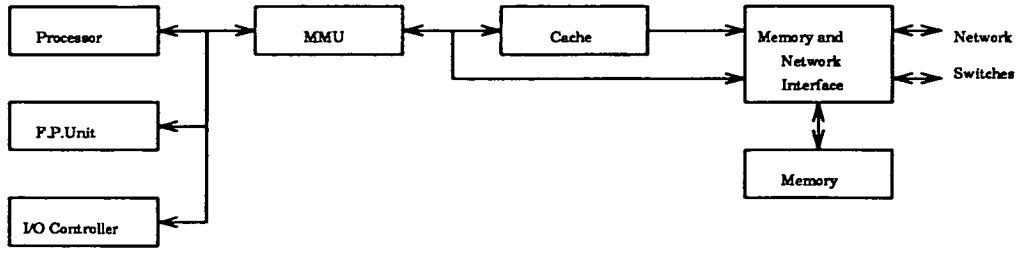


Figure 2.11: Block diagram of RP3 processor-memory element

64 inputs. The latter is used exclusively for transmitting fetch-and-add requests. The PME memory has a soft division into local and shared memory, and shared memory addresses are hashed to reduce contention. A number of fetch-and-op functions are provided for interprocess synchronisation.

2.6 Comments

The areas of processor instruction interleaving, packet-switched combining multistage networks, and fetch-and-add synchronisation have all been introduced in this chapter. Existing shared memory multiprocessor designs which include one or more of these features were briefly described. The Eppl multiprocessor design described in the next chapter combines all these features in a single design.

Chapter 3

Epp1 multiprocessor design

The Epp1 MIMD shared memory multiprocessor design consists of p instruction interleaved RISC processors connected to p memories by a packet-switched combining indirect binary n-cube multistage network constructed from $\frac{p}{2}\log_2 p$ switches. The processors have no local memory or caches, and there is no virtual memory support. Figure 3.1 shows an 8 processor Epp1 system with 12 network switches and 8 memories.

The following sections describe in detail the design of the Epp1 processor, network switch and memory. The processor section includes a description of the processor pipeline, the instruction set, and the results from the simulation of a single processor. The network section includes a description of the combining mechanism, and the results from the simulation of an isolated network.

3.1 The processor

The Epp1 processor supports the concurrent execution of a number of interleaved instruction streams (or processes), and should be implementable as a single-chip VLSI pipelined RISC microprocessor (although implementation issues are not discussed here). This design is an extension of Topham's context-flow processor

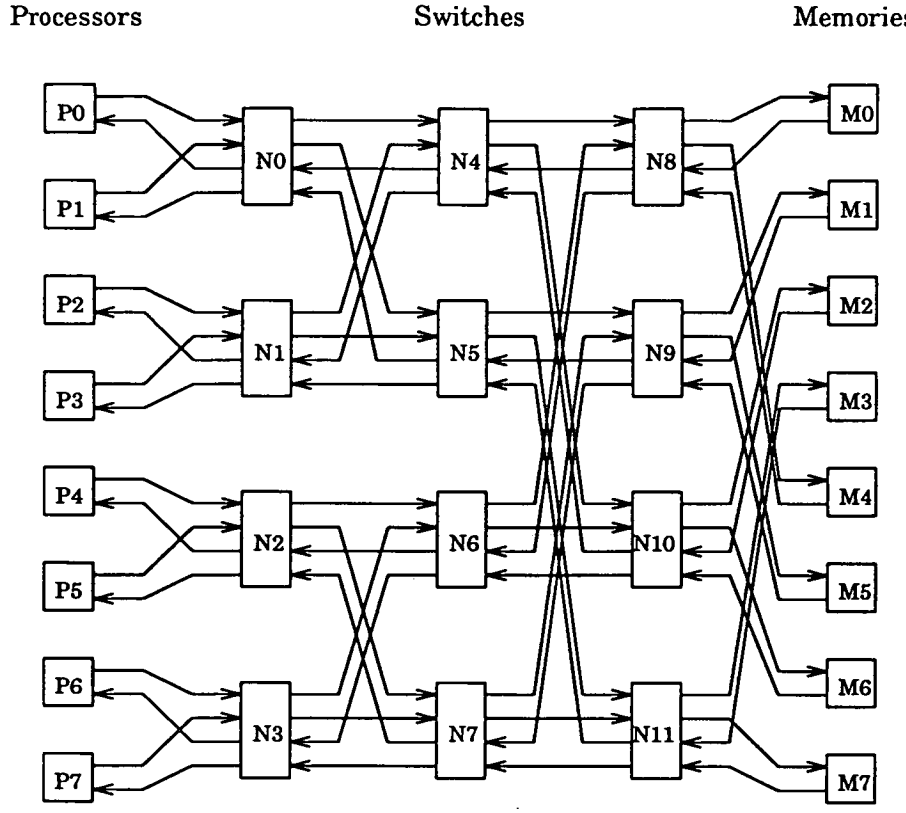
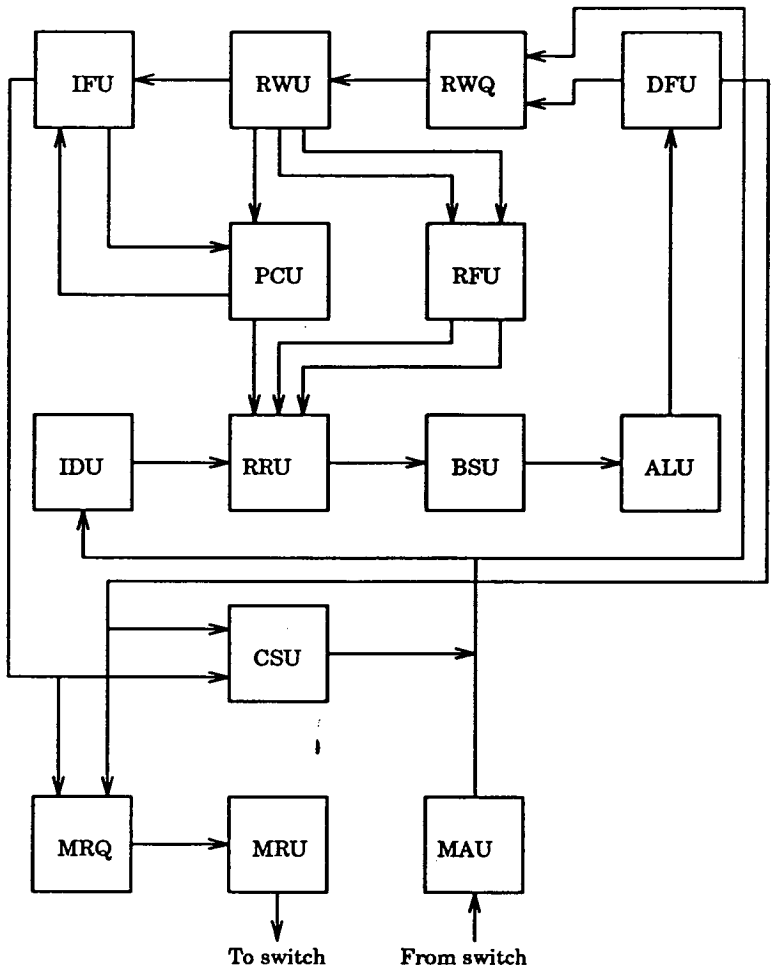


Figure 3.1: Components of an 8 processor Epp1 system

(Section 2.1.4), and has been deliberately kept simple and idealised. Some of the features required in a real processor implementation, but not necessary for the simulations, have been left out to simplify the processor design. For example floating-point support, supervisor mode and exception/interrupt handling. Some of these issues are discussed in [19, 26]. This section effectively describes the Epp1 processor as it is simulated in the Epp1 simulator.

The processor has a general purpose instruction set with a single 32-bit instruction format. Instruction decoding is hard-wired, and there is no microcode. All instructions complete in one instruction cycle, but require a number of clock cycles. Most instructions have a three address register-to-register format (two sources, one destination), and load/store instructions are used for memory access.

Figure 3.2 shows a block diagram of the processor which consists of an 8



IFU = Instruction fetch unit
IDU = Instruction decode unit
RRU = Register read unit
BSU = Barrel shift unit
ALU = Arithmetic logic unit
CSU = Context store unit
MRQ = Memory request queue

DFU = Data fetch unit
RWQ = Register write queue
RWU = Register write unit
PCU = Program counter unit
RFU = Register file unit
MRU = Memory request unit
MAU = Memory acknowledge unit

Figure 3.2: Block diagram showing pipeline units of Epp1 processor

stage circular execution pipeline, three register files, two merging queues, and a memory interface. The pipeline stages all have a 1 clock cycle latency and are clocked continuously and simultaneously by the same clock signal. The number of processes that are executed concurrently by the processor depends on the implementation, and on reset all the processes start executing together. In the processor simulations described in this chapter up to 64 processes are executed per processor. Each process has only a single instruction in execution at any one time, and each instruction requires one pass around the execution pipeline to complete. Each stage of the pipeline contains either an instruction from one of the executing processes or is empty (contains a null process).

3.1.1 Process context

Each process has a context in the processor (apart from any other context in main memory associated with the program being executed). This context is shown in Table 3.1 and consists of: a process identification number (*pid*), a register set, some condition flags, and various operands and control signals used in instruction execution. These are described further below.

Some of the context of a process will move around the pipeline with the process and is called dynamic context, while the rest will be grouped together and only accessed at certain stages, and is called static context [33, 108]. Examples of dynamic context are the *pid*, condition flags, operands and control signals, which move with the process from one pipeline stage to the next. The registers are examples of static context, they are accessed only at certain pipeline stages and it is more area efficient to group them together. Some of the dynamic context, such as the operands and control signals, may exist only for a few pipeline stages. When an instruction is decoded by the instruction decode unit (IDU) a number of control signals are generated, which move from stage to stage with the process. This is called data-stationary control [109, 110] because the control signals move with the data.

context	size (bits)	description
r0–r13	32	general purpose registers
pc (r14)	32	program counter
sr (r15)	32	status register
valid	1	valid process bit
pid	6	process number
carry	1	carry flag
minus	1	minus flag
overflow	1	overflow flag
zero	1	zero flag
instruction	32	instruction code
immediate	32	immediate operand
opbsel	1	operand B select
read1enable	1	enable register read port 1
read1address	4	read port 1 register address
read2enable	1	enable register read port 2
read2address	4	read port 2 register address
write1enable	1	enable register write port 1
write1address	4	write port 1 register address
write2enable	1	enable register write port 2
write2address	4	write port 2 register address
opa	32	operand A
opb	32	operand B
result	32	result
bsufunc	2	barrel shift control
alufunc	4	arithmetic unit control
dfufunc	2	data fetch unit control
isifetch	1	instruction fetch
status	2	memory access type
daddress	32	destination address
saddress	32	source address
data	32	memory data

Table 3.1: Process context

The register set of each process consists of 14 general purpose registers (*r0-r13*), which are stored in the register file unit (RFU). The RFU has 2 read-ports and 2 write-ports, and contains a register set for each process. Register *r14* refers to the program counter (*pc*), and *r15* to the status register (*sr*). The *pc* is stored separately from the other registers in the program counter unit (PCU). The PCU also has 2 read-ports and 2 write-ports, and contains a *pc* for each process. The *sr* does not explicitly exist since the condition flags that make up the status register are part of the process's dynamic context. Only the bottom 4 bits of the *sr* are used.

A process is identified in a pipeline stage by a *valid* bit and the *pid.* The *valid* bit indicates whether the pipeline stage contains a real process or a null process. The *pid* is an integer process number, and is used to index into the register sets and to identify returning memory acknowledge packets.

Each process has four condition flags: *carry*, *minus*, *overflow* and *zero*. Shift instructions can set *carry* in the barrel shift unit (BSU). All the flags can be set by the arithmetic and logic instructions executed in the arithmetic logic unit (ALU).

The remaining context consists of either control signals or operands, and is not user accessible. *Instruction* contains the current instruction, and if it includes an immediate constant then this is sign extended into the 32-bit *Immediate* by the IDU. *Opbsel* indicates whether *opB* should be a register value or an immediate, and *readXenable* and *readXaddress* enable register access and specify a register number respectively. *OpA* and *opB* are the two register operands, and *result* is the result of the ALU operation. *Bsufunc*, *alufunc* and *dfufunc* define the operation of the relevant pipeline stages. *Ifetch* indicates whether the current memory access is an instruction or data fetch. The *status*, *daddress*, *saddress* and *data* fields make up a memory access packet.

3.1.2 Merging queues

The memory request queue (MRQ) and the register write queue (RWQ) are both two-input, one-output FIFO queues. They are used to merge and buffer process contexts going to and returning from memory, so that there is a smooth flow of contexts around the execution pipeline.

The MRQ merges memory requests from the instruction fetch unit (IFU) and the data fetch unit (DFU), and passes them on to the memory request unit (MRU). It is large enough to queue all the processes, and has static storage elements since memory requests may be queued for a number of cycles due to network conflicts. At reset the processes are initialised by the MRQ. Each MRQ entry is set up to contain, for each process, a memory request to fetch the first instruction from the reset location.

The RWQ merges processes coming from the DFU and the memory acknowledge unit (MAU), and passes them to the register write unit (RWU). Because processes are clocked out of the RWQ every cycle, dynamic storage elements can be used, and the queue length need only be $p/2 + 1$, where p is the number of executing processes [111].

3.1.3 Memory interface

Memory requests are issued by the MRU, and returning memory acknowledgements are received by the MAU. The format of the memory request packets is described in Section 3.2.2. Since not all the dynamic context of a process is required for a memory access some of it is stored in the context store unit (CSU), while the memory access is in progress. The CSU is a register file with 1 read-port and 2 write-ports, and contains an entry for each process (what is stored in the entry is described below). The CSU is written to by the IFU and DFU before a memory request, and read from by the MAU when a memory acknowledge returns.

The MRU uses two signals, *validout* and *readyin*, to control the transfer of memory request packets with the network. *Validout* is a boolean output signal and indicates to the connected network switch that a packet is ready to be transferred. *Readyin* is a boolean input signal which indicates that the connected switch is ready to receive a packet. If there is a memory request waiting in the MRU, and *readyin* has been asserted by the network, then the MRU asserts *validout* and transfers the packet. The MAU also has equivalent *validin* and *readyout* signals, but can receive an acknowledge packet every cycle, and so its *readyout* signal is permanently asserted.

3.1.4 Instruction execution

The new instruction cycle of each process starts at the IFU. The IFU reads the processes *pc* from the PCU, increments the *pc* by 1, and writes the new value back to the PCU. Concurrently the dynamic context not required for the memory access, which at this point consists only of the condition and *ifetch* flags, is written to the CSU. The *ifetch* flag is set before it is written to the CSU to indicate that an instruction fetch is in progress. On the following cycle the process is passed on to the MRQ in the form of a memory request. The *status* field is set to indicate a memory load, the new *pc* value is used as the *daddress*, the *pid* forms part of the *saddress*, and the *data* field is zeroed.

The process is then queued in the MRQ, for one cycle minimum, and passed to the MRU on the following cycle. The MRU then handshakes with the network as described above and issues a memory request packet. Because of network latency a number of cycles may elapse before the memory acknowledge packet returns to the MAU. The MAU then extracts the *pid* and *data* from the acknowledge packet and uses the *pid* to index into the CSU. The stored context is read from the the CSU, and since the *ifetch* bit is set, the process is passed to the IDU on the following cycle.

The IDU decodes the new instruction and sets all the relevant register enables

and control functions for the execution units. The process is then passed to the RRU which uses the *pid*, *read1address* and *read2address* values to index into the RFU and read the given operands. *Opa* is read from read-port 1, and *opb* from read-port 2. In the following cycle the BSU executes the shift function indicated by the *bsufunc* field on *opa*. If the *opbsel* flag is set then the *immediate* value is used as the shift value rather than *opb*. The process is then passed to the ALU which executes the function indicated by the *alufunc* field, *opbsel* again determining the source of one of the operands. After the *result* has been calculated the flags are updated, and the process is passed to the DFU on the next cycle.

If *dfufunc* indicates that a memory access should occur, then the DFU clears *isifetch*, and some of the process context is written to the CSU. In this case the context written includes *isifetch*, the register write-port signals, the flags, and the *result*. The *status* field of the memory request is set to indicate either load, store or fetch-and-add. The *daddress* field takes the value of *result*. The *data* field is zeroed for a load, set to the value of *opb* for a store, or is set to either *immediate* or *opb* (depending on *opbsel*) for a fetch-and-add. The memory request is then passed to the MRQ and progresses as described above. When the acknowledge returns to the MAU the stored context is read from the CSU, and the process is passed to the RWQ (since *isifetch* is cleared) along with the returning *data* value. If *dfufunc* indicates that no memory access should occur then the process is passed directly from the DFU to the RWQ on the following cycle.

The RWQ receives the process from either the DFU or the MAU, and the process is queued for a minimum of one cycle. The process is then passed to the RWU which uses *pid*, *write1port* and *write2port* to index into the RFU. The value of *result* is written to write-port 1, and *data* or *opb* to write-port 2. This completes the instruction cycle, and the process is passed to the IFU once again.

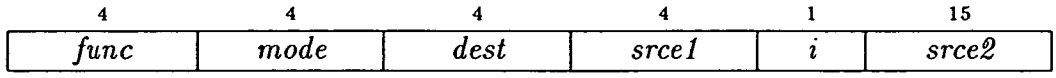


Figure 3.3: Instruction format

3.1.5 Instruction format

The Epp1 instructions have a single 32-bit format similar to the RISC I [112]. The instruction has six fields as shown in Figure 3.3. The *func* (function) field specifies the general instruction type, for example: branch, compare or arithmetic. The *mode* field specifies the particular instruction type, for example: branch always or branch greater-than. The *dest* (destination) field specifies the destination register address to which the result of the instruction is written. The *srce1* (source 1) field specifies a source register address. The *i* (immediate) bit specifies whether the *srce2* (source 2) field contains either a 15-bit signed immediate constant ($i = 1$), or a source register address in the lower 4 bits ($i = 0$).

The 4-bit register addresses in *dest, srce1* and *srce2* specify one of *r0–r15*. If *pc* (*r14*) is used then the PCU is accessed rather then the RFU. If *sr* (*r15*) is used then the flags are accessed.

3.1.6 Instruction set

The assembly language mnemonics of the Epp1 instruction set are shown in Table 3.2, where r_{dest} is a 4-bit destination register address, r_{srce1} and r_{srce2} are 4-bit source register addresses, and i_{15} is a 15-bit signed constant. The "quick" form of the instructions indicate that *srce2* is a constant rather than a register address. The instruction set was chosen to cover the usual range of instructions and no attempt was made to optimise the number of instructions (for instance *not* can be implemented by *exor*-ing with 1 [112], and *ble* can be implemented by using *bgt* with the operands swapped [19]).

The branch instructions conditionally change the value of the *pc* by an immediate offset. The branch condition is evaluated in the IDU using the flags. If the

instruction	description
add $r_{dest}, r_{srce1}, r_{srce2}$	add
addq $r_{dest}, r_{srce1}, i_{15}$	add quick
sub $r_{dest}, r_{srce1}, r_{srce2}$	subtract
subq $r_{dest}, r_{srce1}, i_{15}$	subtract quick
mul $r_{dest}, r_{srce1}, r_{srce2}$	multiply
mulq $r_{dest}, r_{srce1}, i_{15}$	multiply quick
div $r_{dest}, r_{srce1}, r_{srce2}$	divide
divq $r_{dest}, r_{srce1}, i_{15}$	divide quick
rem $r_{dest}, r_{srce1}, r_{srce2}$	remainder
remq $r_{dest}, r_{srce1}, i_{15}$	remainder quick
and $r_{dest}, r_{srce1}, r_{srce2}$	logical and
or $r_{dest}, r_{srce1}, r_{srce2}$	logical or
exor $r_{dest}, r_{srce1}, r_{srce2}$	logical exclusive-or
not r_{dest}, r_{srce1}	logical not
rol $r_{dest}, r_{srce1}, r_{srce2}$	rotate left
rolq r_{dest}, r_{srce1}, i_5	rotate left quick
ror $r_{dest}, r_{srce1}, r_{srce2}$	rotate right
rorq r_{dest}, r_{srce1}, i_5	rotate right quick
cmp r_{srce1}, r_{srce2}	compare
cmpq r_{srce1}, i_{15}	compare quick
bra i_{15}	branch always
bcs i_{15}	branch carry set
bvs i_{15}	branch overflow set
beq i_{15}	branch equal
bne i_{15}	branch not equal
blt i_{15}	branch less than
ble i_{15}	branch less than or equal
bgt i_{15}	branch greater than
bge i_{15}	branch greater than or equal
ldo $r_{dest}, r_{srce1}, i_{15}$	load offset
sto $r_{dest}, r_{srce1}, i_{15}$	store offset
ldi $r_{dest}, r_{srce1}, i_{15}$	load increment
sti $r_{dest}, r_{srce1}, i_{15}$	store increment
ldq r_{dest}, i_{15}	load quick
stq r_{dest}, i_{15}	store quick
f&a $r_{dest}, r_{srce1}, r_{srce2}$	fetch-and-add
f&aq $r_{dest}, r_{srce1}, i_{15}$	fetch-and-add quick
noop	no operation

Table 3.2: Epp1 assembly code mnemonics

branch condition is true then the IDU sets *alufunc* to *add*, and the immediate offset is added to the *pc*. The result is then written back to the PCU and the new instruction is fetched at the start of the next instruction cycle. If the condition is false then *alufunc* is set to *pass* and the *pc* is not written back. The immediate offset must be adjusted to account for the *pc* increment by the IFU.

Branch on condition instructions should be preceded by a compare instruction. The *cmp* and *cmpq* are implemented as a subtraction ($r_{srce1} - r_{srce2}$), the result of which is discarded. The ALU sets the flags accordingly and they are used by the subsequent branch instruction.

The shift instructions are executed by the BSU, with a shift magnitude between 0 and 32 bits (the bottom 5 bits of the shift operand). The value in r_{srce1} is rotated by the value of r_{srce2}, and stored in r_{dest}. The last bit rotated is available in the *carry* flag after the rotate.

The arithmetic and logic instructions are executed by the ALU. Only integer arithmetic is supported, and the ALU can perform both multiply, divide and remainder operations in a single cycle. This unrealistic assumption was made to simplify the design and in an implementation the ALU would itself be pipelined. For non-commutative operations such as subtract, divide and remainder r_{srce1} is assumed to be the left-hand operand (for example $r_{srce1} - r_{srce2}$, r_{srce1}/r_{srce2}, *etc*). The *carry*, *minus* and *zero* flags are set by all ALU operations, while the *overflow* flag is set only for addition and subtraction.

The load and store instructions have three addressing modes, each of which is implemented as a separate instruction. For a load the address register is r_{srce1}, and for a store it is r_{dest}. The offset is always a 15-bit immediate constant. The offset addressing mode allows the memory address supplied from a register to be modified by an immediate offset, and the address register is not changed. The increment addressing mode allows the memory address to be modified by an immediate offset, and the address register is updated with the new value (pre-increment). The quick addressing mode allows an immediate constant to be

loaded into a register or to be stored in a memory word.

The fetch-and-add instruction atomically adds a given value (which can be an immediate constant or from r_{srce2}) to a memory word addressed to by r_{srce1}, and returns the previous value of the memory word to r_{dest}.

The benchmark programs described in Chapter 5 are coded in the Epp1 assembly language mnemonics just described, and the assembly code for these programs is shown in Appendix A.

3.1.7 Single processor simulation

The number of processes required to achieve maximum pipeline utilisation in an instruction-interleaved processor depends on the pipeline length and the memory latency. To obtain steady-state performance bounds a single processor simulator was written using Pascal on a Sun workstation.

The simulated processor has the same pipeline configuration as the Epp1 processor described above, but can only execute three "meta-instructions" called: *icycle*, *acycle* and *dcycle*. These specify that an instruction fetch, an arithmetic operation, or a memory access are to be executed. No actual operations are carried out, the meta-instructions simply determine the path of a process through the processor. Various instruction mixes can be synthesised using the meta-instructions. The simulator allows the meta-instruction sequence, the interleaving level (number of executing processes), the number of memory pipeline stages and the memory cycle time to be varied. The total memory latency is given by the number of memory pipeline stages times the memory cycle time. The simulator collects a number of execution statistics including: the processor pipeline utilisation, the memory interface utilisation, the processor instruction time which is the number of cycles per instruction per processor, and the process instruction time which is the number of cycles per instruction per process. These are displayed after the simulator has been clocked for a given number of cycles.

Three synthetic workloads (meta-instruction sequences) called *best*, *average*

and *worst* are used in the simulations:

best = icycle, acycle.

average = icycle, acycle, icycle, dcycle.

worst = icycle, dcycle.

In the best case workload there is 1 memory access per instruction, in the average case 1.5, and in the worst case 2. As the simulator is clocked each of the processes repeatedly executes the given meta-instruction sequence (each process has its own meta-instruction counter). The number of instructions executed is equal to the number of *icycles* executed. In each simulation below the simulator was clocked until a total of 10000 instructions had been executed.

Varying the workload

The simulation results in Figures 3.4 to 3.7 show how the performance changes for the three workloads with the interleaving level increasing from 1 to 64 processes (in single process increments). The number of memory pipeline stages and memory cycle time were both 1 (therefore memory latency is 1 cycle).

For each workload the pipeline utilisation (Figure 3.4) increases as more processes are added, and then levels off at between 9 and 12 processes as the pipeline becomes full (or saturated). The maximum pipeline utilisation drops from 100% in the best case workload to 50% in the worst case. This is because returning memory requests are passed from the MAU to the IDU or the RWQ, and the pipeline utilisation is measured in the IDU. So whenever a data fetch returns the IDU remains idle and some utilisation is lost. The memory utilisation (Figure 3.5) increases to 100% as more processes are added, for all three workloads. The worst case workload reaches maximum utilisation faster than the best case workload. The processor instruction time (Figure 3.6) decreases as more processes are added, and then levels off at the saturation point. The best case workload executes at about 1 cycle per instruction. The process instruction time (Figure 3.7)

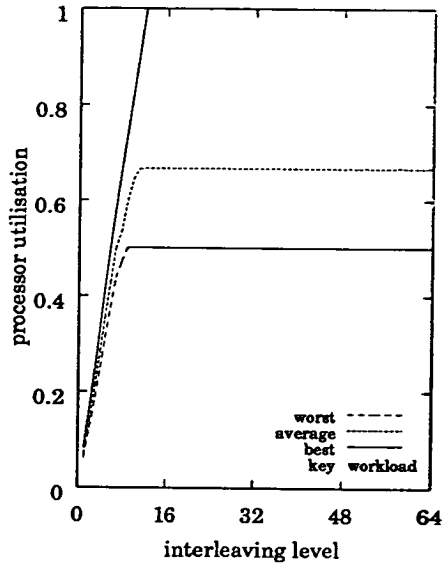


Figure 3.4: Pipeline utilisation with various workloads.

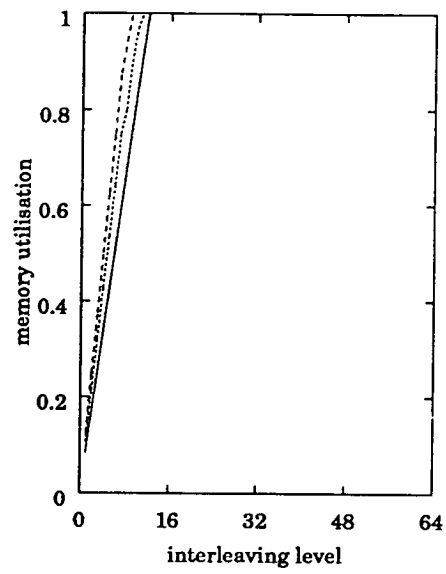


Figure 3.5: Memory utilisation with various workloads.

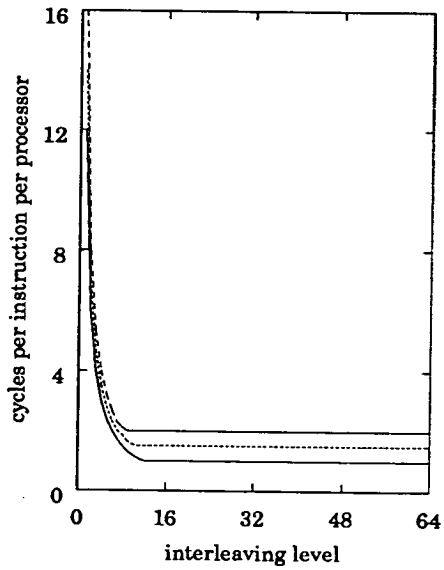


Figure 3.6: Processor instruction time with various workloads

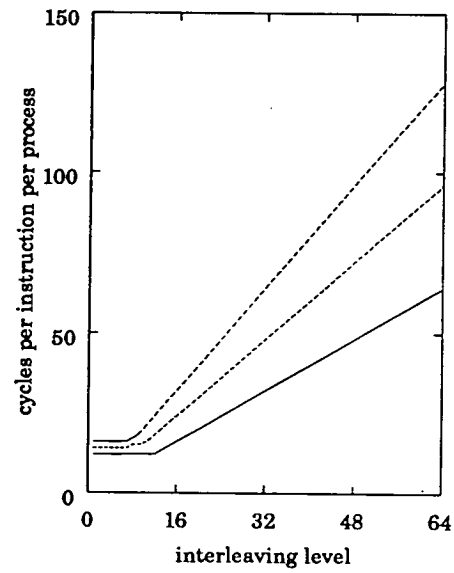


Figure 3.7: Process instruction time with various workloads

remains level until near the saturation point, and then increases linearly as more processes are added. Above the saturation point the processes start to compete for resources and spend more time queued in the MRQ and RWQ.

The workload of a real program would vary dynamically between the best case and the worst case shown. So for a system with unit memory latency the performance would be bounded by these results.

Varying the number of memory pipeline stages

The simulation results in Figures 3.8 to 3.11 show, using the average workload, how the performance changes with an increasing number of memory pipeline stages and interleaving level. The number of memory stages used are: 1, 2, 4, 8, and 16. The memory cycle time is 1 throughout, and so the memory latency is equal to the number of memory stages. This models the case in which the processor is connected to a pipelined network (such as a packet-switched multistage network), and the network cycle time equals the processor cycle time.

For all memory pipeline lengths the pipeline utilisation (Figure 3.8) increases as more processes are added, and then levels off at the maximum utilisation possible with this workload. As the number of memory pipeline stages is increased, more processes are needed to reach the saturation point (with 16 stages, 24 processes are needed to reach the maximum utilisation). The memory utilisation (Figure 3.9) increases to the maximum as more processes are added, but again more processes are required to reach the saturation point with greater numbers of stages. The processor instruction time (Figure 3.10) decreases as processes are added, and levels off at the saturation point. More processes are necessary to achieve the minimum instruction time as the number of stages increases. The process instruction time (Figure 3.11) remains level as processes are added, and then starts to increase near the saturation point. All the instruction times converge after the saturation point.

Increasing the number of memory pipeline stages increases the memory la-

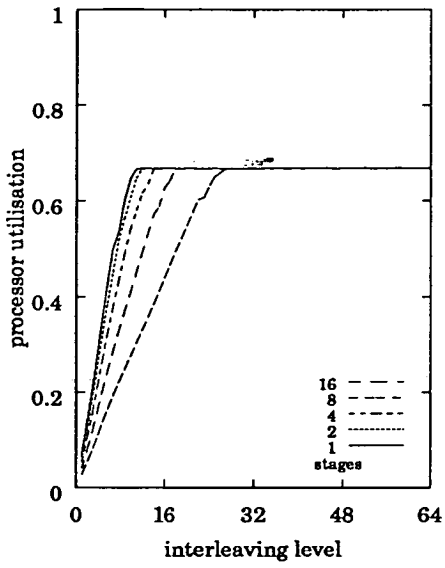


Figure 3.8: Pipeline utilisation with increasing memory pipeline stages.

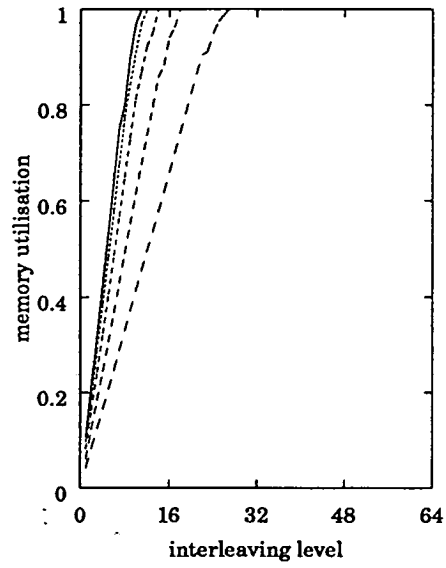


Figure 3.9: Memory utilisation with increasing memory pipeline stages.

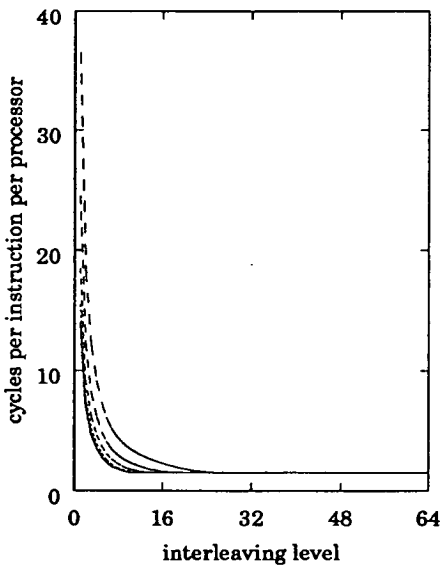


Figure 3.10: Cycles per instruction per processor with increasing memory pipeline stages.

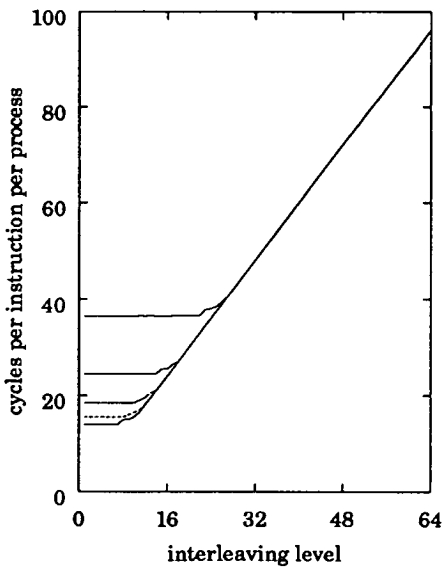


Figure 3.11: Cycles per instruction per process with increasing memory pipeline stages.

tency, but still allows pipelined memory access. The general effect is to move the saturation point to a larger number of processes. Even with a large number of stages the maximum utilisation can still be achieved using correspondingly greater numbers of processes.

Varying the memory cycle time

The simulation results in Figures 3.12 to 3.15 show, using the average workload, how the performance changes with increasing memory cycle time and interleaving level. The memory cycle times used are: 1, 2, 4 and 8 cycles. The number of memory pipeline stages is 1 throughout, and so the memory latency is equal to the memory cycle time. This models the case in which the processor is connected to a local memory which has a larger cycle time than the processor.

The pipeline utilisation (Figure 3.12) increases up to the saturation point, and then levels off. As the memory cycle time is increased, and memory accesses take longer, the maximum pipeline utilisation possible is reduced. The memory utilisation (Figure 3.13) quickly reaches the maximum as the interleaving level is increased. With longer memory cycle times the memory utilisation is increased as processes spend relatively more time in the memory. The processor instruction time (Figure 3.14) almost doubles with each doubling in memory cycle time, and few processes are required to reach the saturation point. The process instruction time (Figure 3.15) increases as processes are added, with the highest memory cycle time resulting in the highest rate of increase.

Increasing the number of processes cannot compensate for large memory cycle times in the same way as for the memory pipeline stages, since memory access cannot be pipelined. The memory becomes a bottleneck on the performance of the processor.

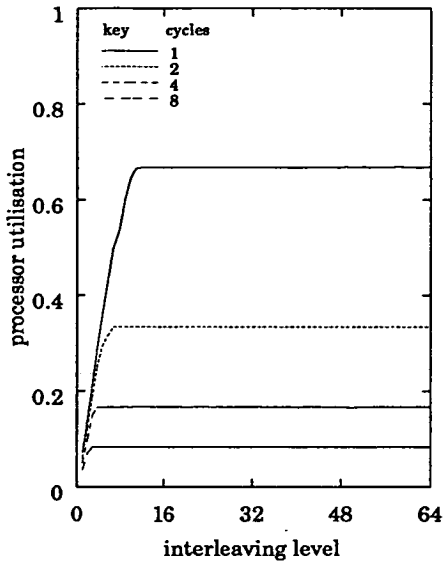


Figure 3.12: Pipeline utilisation with increasing memory cycle time.

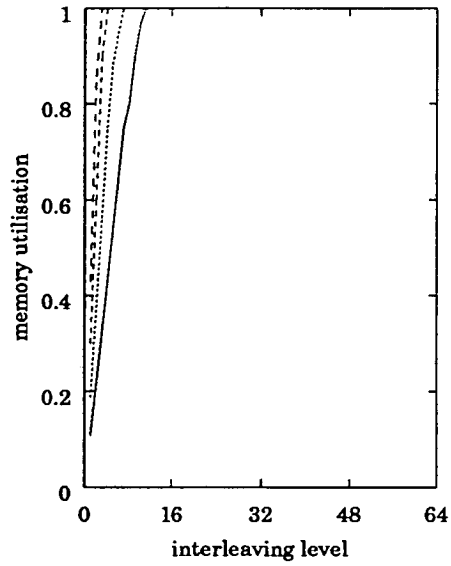


Figure 3.13: Memory utilisation with increasing memory cycle time.

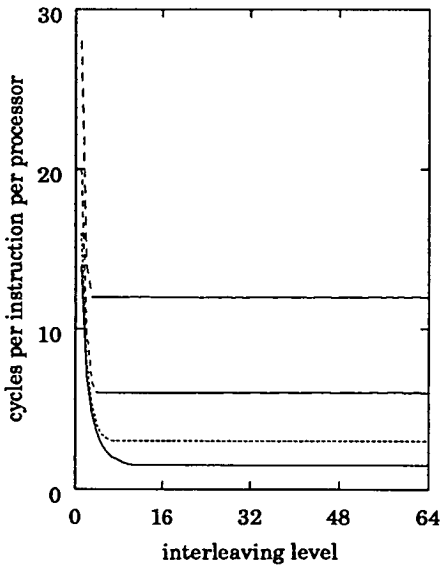


Figure 3.14: Cycles per instruction per processor with increasing memory cycle time.

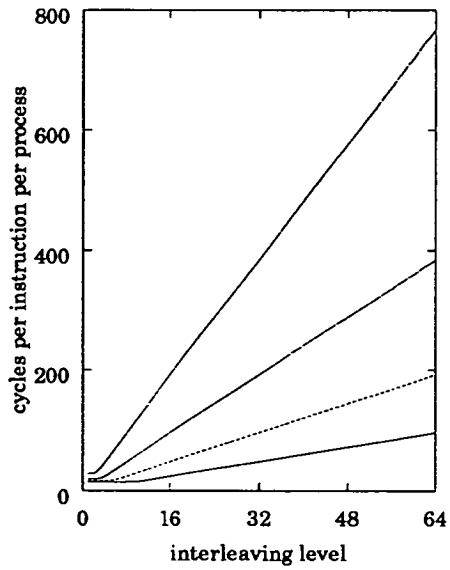


Figure 3.15: Cycles per instruction per process with increasing memory cycle time.

Summary

The simulation results show that the maximum processor utilisation possible is dependent on the workload and the memory cycle time. As the average number of memory accesses per instruction and the memory cycle time increase, the maximum utilisation possible is reduced. As the number of memory pipeline stages is increased, the interleaving level must also be increased to reach the maximum utilisation. These results are in agreement with similar studies [27, 26, 25] (Section 2.1.3). Lees [108] has performed simulations of a context-flow processor design with pipelined floating point units, in which the maximum queue lengths of the various merging queues in the design are measured.

This concludes the description of the Epp1 processor; the following section describes the network used to connect the processors and memories together.

3.2 The network

The Epp1 network is constructed from 2×2 combining switches connected in an indirect binary n-cube topology (Section 2.2.1 and Figure 3.1). The network is divided into a request half which routes request packets from the processors to the memories, and an acknowledge half which routes acknowledge packets from the memories to the processors. The request and acknowledge halves are physically merged to facilitate combining, so in a p input network there are $\frac{p}{2} \log_2 p$ switches.

The internal block structure of the switch is shown in Figure 3.16. The switch is also divided into a request and acknowledge halves. The request half consists of two input ports, two input latches, two request queues and two output ports. The request queues each have two inputs, and can accept two packets and output one packet in each cycle. If the queue is empty then the packet goes to the head of the queue, but is not output until the next cycle. Matching packets can be combined in the request queue. The acknowledge half consists two input ports, two input latches, two acknowledge queues and two output ports. The acknowledge queues

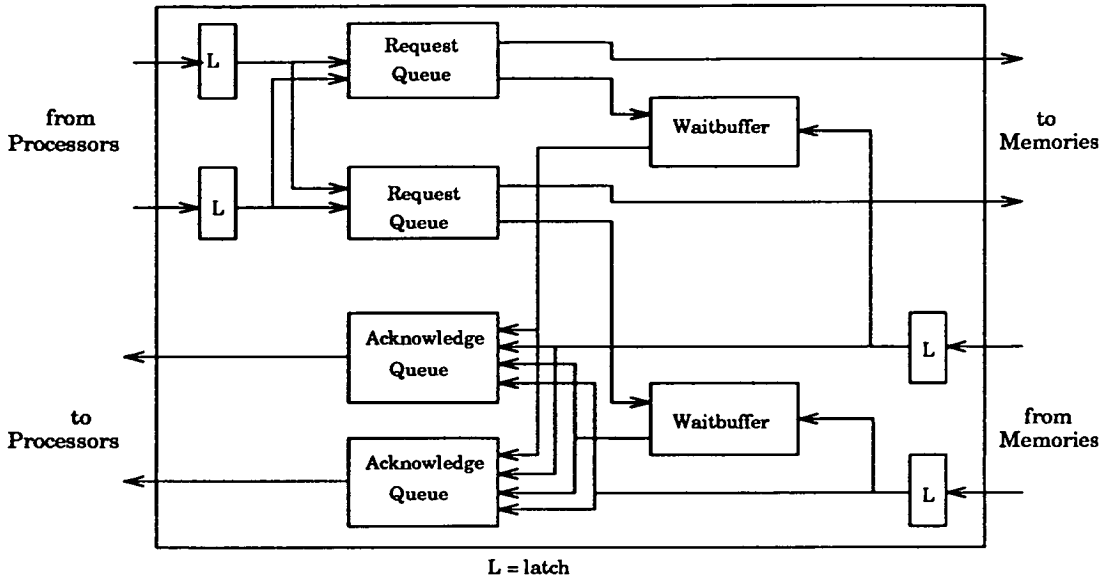


Figure 3.16: Internal structure of network switch

each have four inputs and can accept four packets (or more with higher combining levels), and output one packet each cycle. Each packet spends at least one cycle in the queue. The request and acknowledge halves of the switch are connected by the two wait-buffers. The wait-buffers are associative memories which are used to store one of a pair of combined packets, until the other one returns from memory and can be decombined. Each wait-buffer is written to by one of the request queues, and read by one of the acknowledge input latches. It is assumed that each request queue contains an adder unit used in combining fetch-and-add requests, and that each acknowledge input latch also contains an adder unit used in decombining fetch-and-add acknowledges (these adders are not shown in the diagram).

The switches are synchronously clocked from the same signal. In each clock cycle a switch can input a packet to each input port, and output a packet from each output port. A packet can be input and routed in one clock cycle, and output on the next clock cycle, although it may also be queued in the switch for a number of cycles.

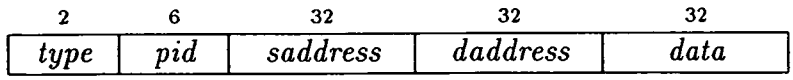


Figure 3.17: Request and acknowledge packet format

3.2.1 Ports

Each port of the switch consists of a pair of handshake signals and the packet transfer wires. The transfer of packets between the switches is controlled by *ready* and *valid* signals (the processor has the same memory interface described in Section 3.1.3).

An input port has a *readyout* output signal and a *validin* input signal. *Readyout* is asserted if the input latch is empty and the port can receive a new packet. If the connected switch has a packet to send then it will assert *validin* and the packet will be transferred.

An output port has a *readyin* input signal and a *validout* output signal. If the *readyin* signal is asserted by the connected switch and there is a packet in the output queue, then the *validout* signal is asserted and the packet is output. The whole packet is transferred in the same cycle.

3.2.2 Packet format

Request and acknowledge packets have the same format. Each packet consists of five fields as shown in Figure 3.17. The *type* field defines the memory access type, the *pid* field contains the process number of the requesting process, the *saddress* field contains the return address, the *daddress* field contains the destination address, and the *data* field contains a 32-bit datum. The *saddress* field in a request packet is initially empty, and the return address is generated as the packet traverses the network (as described below). The *daddress* field can be subdivided into a memory number, and an offset within that memory.

3.2.3 Routing

A packet is routed from the input latch to one of the queues depending on the routing bit, which is the least significant bit of the *daddress*. If the routing bit is 0 the packet is routed to the upper queue, if it is 1 to the lower queue. Before the packet is output to the next stage the *daddress* is shifted one bit to the right, discarding the current routing bit. These operations are the same for both request and acknowledge halves of the switch.

The return address is generated on the fly by the request half of the switch. When a packet is input to a port the *saddress* is shifted one bit to the left, and the port number (either 0 or 1) is shifted into the least significant bit. Assume a network with 8 inputs as shown in Figure 3.1, and that a request packet is transmitted from processor 5 destined for memory 7 (the bottom 3 bits of the *daddress* therefore have the value 111). Initially the *saddress* contains 000, after the first stage it contains 001, after the second stage 010, and after the third stage 101, which is the correct return address. This address is then used as the *daddress* in the acknowledge packet, again read from the least significant bit.

3.2.4 Combining

Combining occurs in the request queues. An incoming packet can be combined with one of the packets already in the request queue if the *type* and *daddress* fields of the two packets are identical. For simplicity combining is restricted to occurring between load and load packets, and fetch-and-add and fetch-and-add packets only. More general combining could also be implemented [72].

Figure 3.18 shows the combining of two packets originating from process A on processor A and process B on processor B, for both the load and fetch-and-add case. The packet in the request port input latch (from processor B) is first compared to all packets in the request queue, starting at the head of the queue. If the wait-buffer is not full, then the matching packet in the queue (from processor

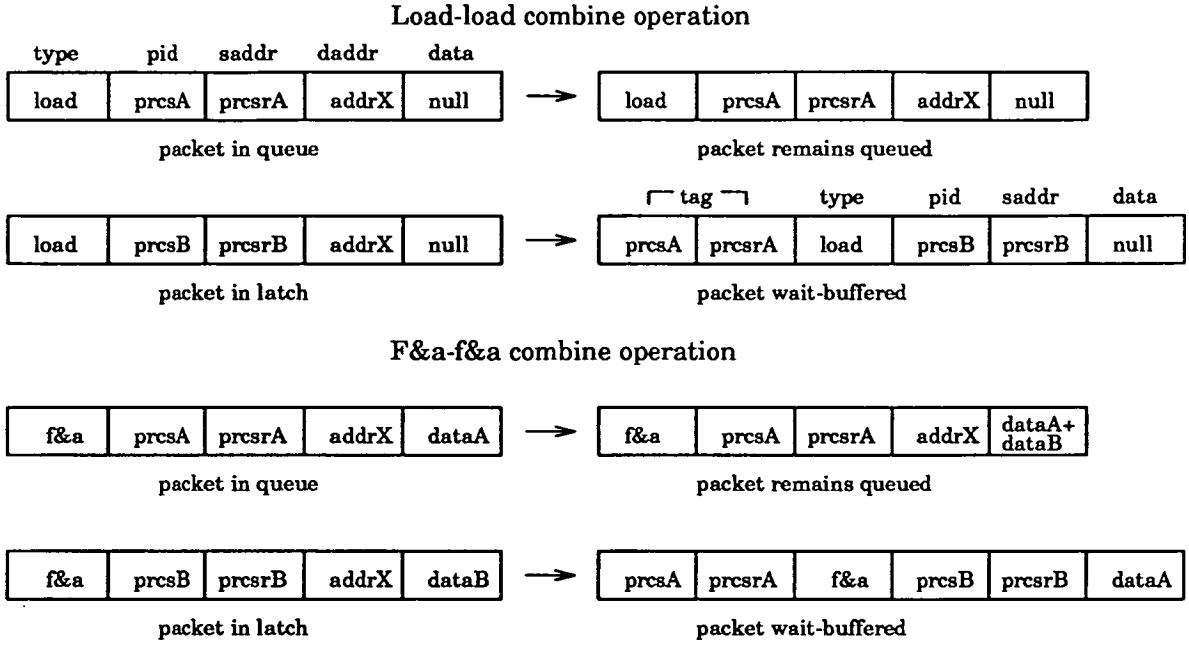


Figure 3.18: Combining operations

A) can be combined. The *pid* and *saddress* of the queued packet are used as a tag in the wait-buffer to identify the stored packet. The *saddress* of the incoming packet is copied to the wait-buffer, and the *daddress* discarded. The contents of the *data* field of the stored packet is dependent on the type of memory request. For load combining the *data* field is left empty. For fetch-and-add combining the *data* field of the queued packet is copied into that of the stored packet, and the queued packets *data* field is updated with the sum of itself and the *data* field of the incoming packet. The queued packet can then proceed to the next switch.

The request queues can allow more than 2-way or pairwise combining to occur (Section 2.2.5). If a queued packet is involved in a combine but is still in the queue in a subsequent cycle, then it may combine again. Each combine results in a new wait-buffer entry. The maximum combining level is 4-way.

3.2.5 Decombining

Decombining occurs in the acknowledge half of the switch. The *pid* and *daddress* field of the returning packet are compared to all the tags in the wait-buffer (an

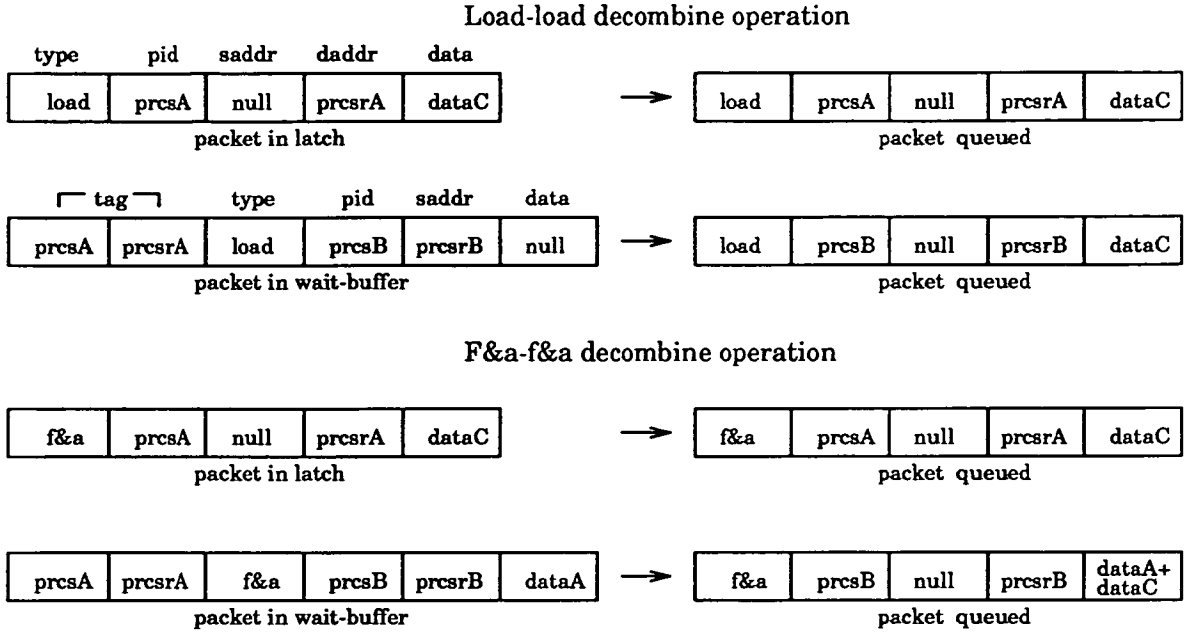


Figure 3.19: Decombining operations

associative match). If there is a match then the returning packet was previously combined with the packet in the wait-buffer.

Figure 3.19 shows the decombining operations for the packets combined in the previous diagram. For load decombining the returning packet's *data* field is copied to the decombined packet's *data* field, so the loaded value returns to both processes. For fetch-and-add decombining the returning packet's *data* field is added to the decombined packet's *data* field to generate the correct unique value. After decombining both packets are routed and queued. If the queue the decombined packet is to be routed to is full, then decombining is delayed until the next cycle.

If more than 2-way combining is allowed then all matching packets are decombined and queued in the same cycle (with higher combining levels this becomes rather idealised). If the acknowledge queues become full then the returning packet remains in the input latch, and decombining commences in the following cycle.

3.2.6 Isolated network simulation

The performance of multistage networks has been well researched using both analytical techniques and simulation [113, 47, 57, 114, 58, 59, 73, 74, 2] (Section 2.2). For purposes of comparison a simulator of an isolated indirect binary n-cube multistage network was written using C on a Sun workstation. The simulated network has the same switch design and topology as the request half of the Eppl network, except that combining is not implemented. The network is isolated in that there are no attached processors or memories, and the network loading is not related to network throughput (defined as an open system in queueing theory terminology). Packets are presented to the network inputs with a random destination address, unless a hot-spot is specified. The number of new packets entering the network is dependent on the given network loading. Packets output from the network are discarded.

The simulator allows the switch queue size and network loading to be varied. The loading is the percentage of submittable packets that were submitted to the network, for example if the network can accept 100 new packets in a cycle then only submitting 50 packets results in a 50% loading. The simulator also allows a single hot-spot to be specified; this is a memory address to which a higher percentage of packets are addressed than the other locations. The hot-spot percentage is the percentage of packets submitted that were addressed to the hot-spot, above the background traffic. The network performance is measured in terms of the network throughput and latency. The (normalised) throughput is the percentage of packets output by the network compared to the maximum throughput possible. The latency is the average number of cycles a packet takes to cross the network.

The simulation results below show the effect of queue size and a hot-spot on network performance, for networks with between 2 and 256 inputs. For each simulation the simulator was clocked for 10000 cycles to allow a steady state to be reached.

Varying the queue size

The simulation results in Figures 3.20 and 3.21 show the effect of increasing the queue size from 1 to 32 packets, with a 100% load. As the queue size is increased the throughput (Figure 3.20) increases for all network sizes, and approaches the maximum at a queue size of 32. The latency (Figure 3.21) increases unboundedly with queue size. This is because more packets can be accepted by the network as the queue size increases, resulting in longer waiting-times and thus higher latency. Increased throughput is desirable, but increased latency is not, so a small queue size of 4–8 packets appears to be optimal.

Varying load

The simulation results in Figures 3.22 and 3.23 show the effect of increasing the load from 50% to 100%, with a queue size of 4 throughout. As the load is increased the throughput (Figure 3.22) increases and then levels off, at which point the network is saturated and cannot accept more packets. The latency (Figure 3.23) increases with load, since there are more packets in the network resulting in higher contention.

Varying queue size with hot-spot

The simulation results in Figures 3.24 and 3.25 show the throughput and latency with varying queue size, but with 1% (above background traffic) of the 100% load directed at a hot-spot. Compared to Figure 3.20, the throughput (Figure 3.24) has been significantly reduced by the hot-spot for larger network sizes. For example a throughput of 95% drops to 30% for a queue size of 32 and 256 inputs. Increasing the queue size improves throughput only slightly for the larger network sizes. The latency (Figure 3.25) is also much higher with the hot-spot for all network sizes compared to Figure 3.21.

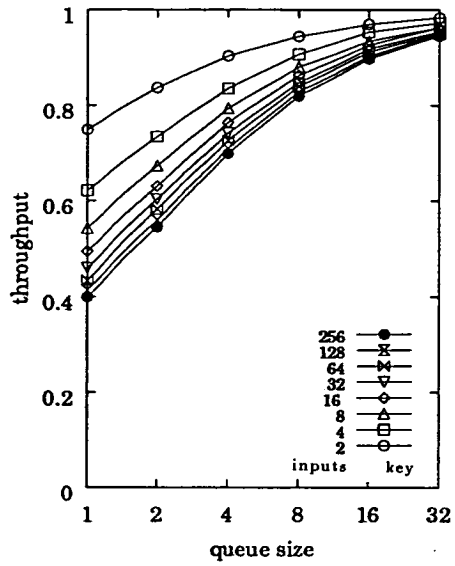


Figure 3.20: Network throughput with increasing queue size and 100% load.

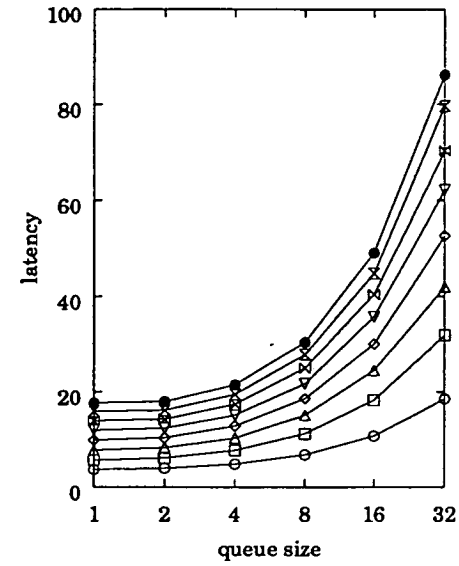


Figure 3.21: Network latency with increasing queue size and 100% load.

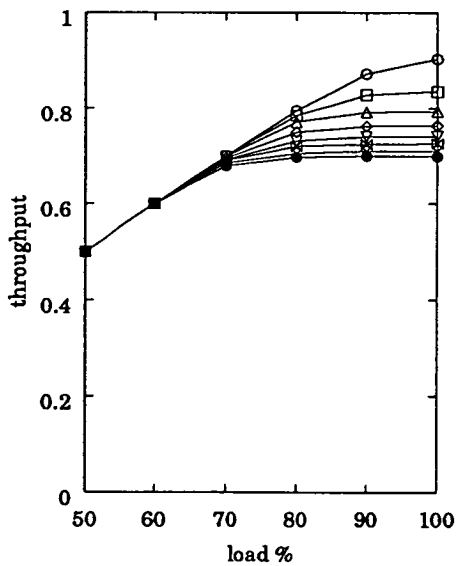


Figure 3.22: Network throughput with increasing load and queue size 4.

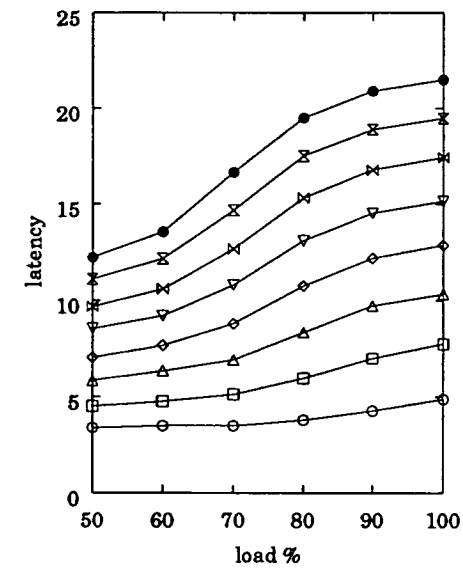


Figure 3.23: Network latency with increasing load and queue size 4.

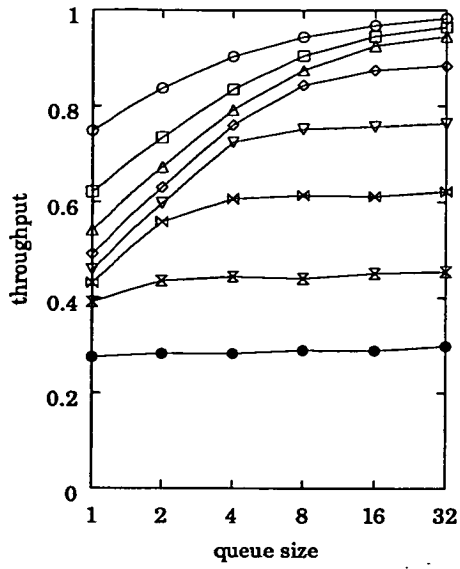


Figure 3.24: Network throughput with increasing queue size, 1% hot-spot and 100% load.

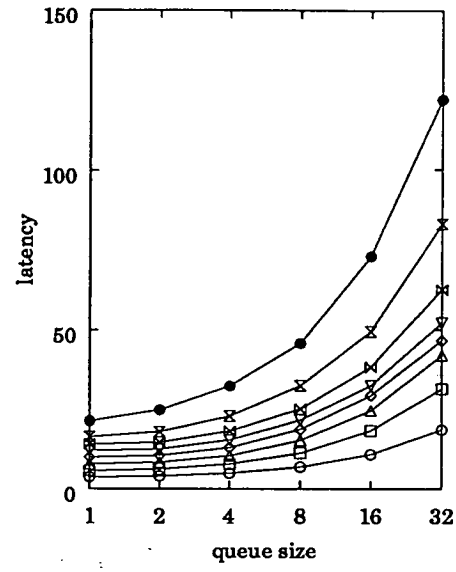


Figure 3.25: Network latency with increasing queue size, 1% hot-spot and 100% load.

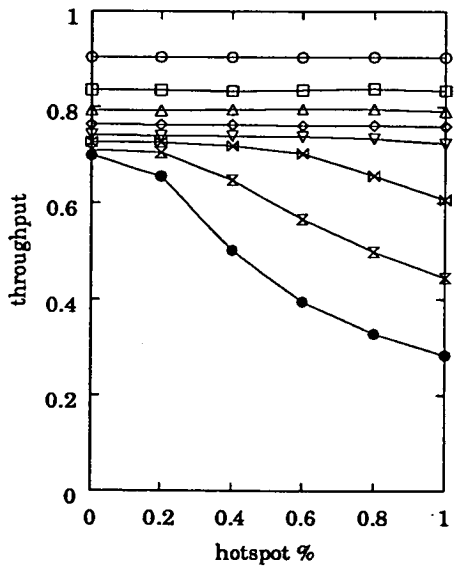


Figure 3.26: Network throughput with increasing hot-spot, queue size 4 and 100% load.

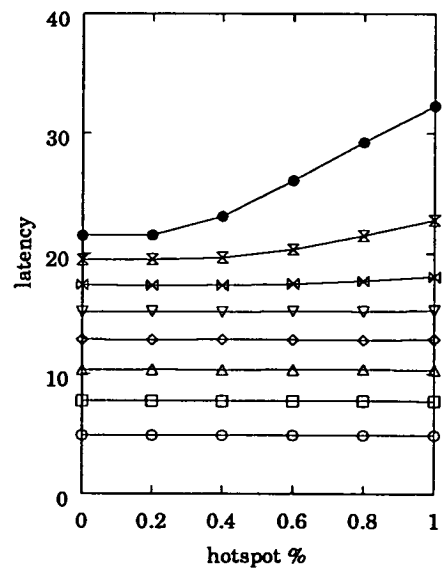


Figure 3.27: Network latency with increasing hot-spot, queue size 4 and 100% load.

Varying hot-spot percentage

The simulation results in Figures 3.26 and 3.27 show the effect of varying hot-spot percentages on throughput and latency with a 100% load and queue size 4. As the hot-spot percentage is increased from 0 to 1%, the throughput (Figure 3.26) decreases for networks larger than 16 inputs. The latency (Figure 3.27) only increases significantly for networks with more than 64 inputs. It is evident that even a very small hot-spot can cause a large reduction in network performance.

Summary

These results show similar characteristics to other network simulations, although the exact figures for throughput and latency depend on the switch design used. The actual performance of the network when used in the Eppl design will depend on the load resulting from the interleaving level, and the hot-spot percentage resulting from the memory access pattern.

This concludes the description of the Eppl network; the following section describes the operation of the Eppl memory.

3.3 The memory

The Eppl processors are connected to an equal number of shared memories. Logical memory addresses are low-order interleaved across the memories, so that consecutive physical addresses are in consecutive memory modules. In a p processor system, the lower $\log_2 p$ bits of the *daddress* are used to route the request packet to one of the memories. Then the remaining $32 - \log_2 p$ bits are used as an offset in that memory.

Each memory module has an input port, an array of memory, a memory controller and an output port as shown in Figure 3.28. The usual *ready* and *valid* signals are used to control the transfer of packets between the memory and

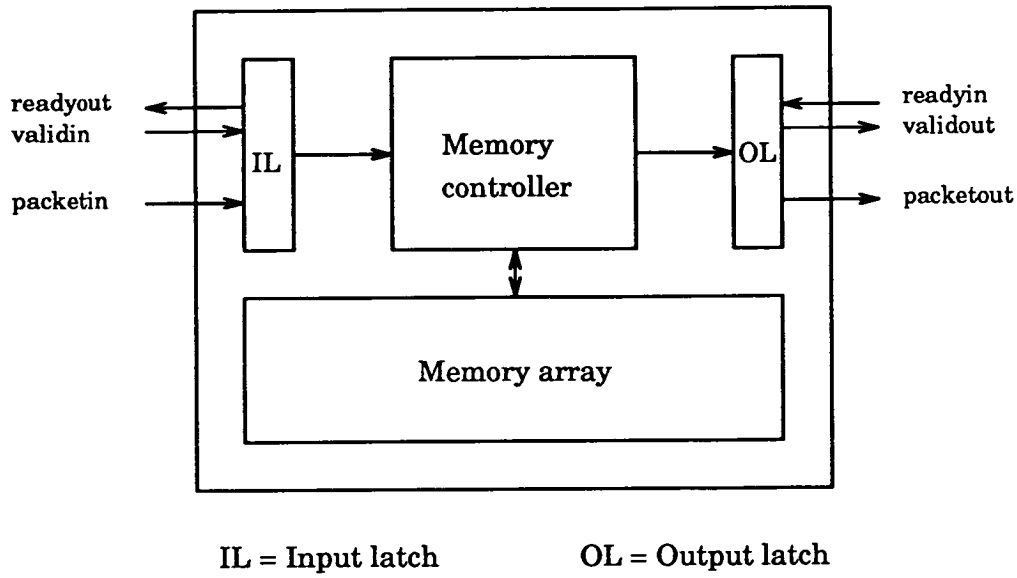


Figure 3.28: Block diagram of functional units in a memory module

the network. The memory controller performs the memory access, which can be either a load, store, or fetch-and-add. In the latter case a read-memory-write cycle is necessary, and the memory controller also has an integer adder which is used in calculating the result of the fetch-and-add. All memory accesses including fetch-and-add take one cycle.

3.3.1 Operation

At the start of a cycle, if the input latch is empty, the memory module asserts the *readyout* output signal on its input port. If the connected network switch has a packet, then the *validin* input signal will in turn be asserted by the switch, and the packet transferred. Concurrently, if there is a packet in the memories output latch, and the *readyin* input signal has been asserted, then the output port's *validout* signal is asserted and the packet is transferred to the network. If the packet in the output latch has been successfully transferred, then the packet in the input latch starts its memory access, and the result is written into the output latch. If the output latch is full, because the network switch cannot accept the packet, then the memory access is deferred until the output latch is empty.

When the memory module receives a request packet it copies the *type* and *pid* fields into the acknowledge packet, and transfers the *saddress* field, which contains the return address, to the *daddress* field of the acknowledge packet. The *saddress* field of the acknowledge packet is unused. If the memory request is a load the memory word at the offset given by the request packet's *daddress* field is copied into the acknowledge packet's *data* field. If it is a store then the request packet's *data* field is copied into the memory location. If it is a fetch-and-add the value of the memory location is copied to the acknowledge packet's *data* field, and the source packet's *data* field is added to the value in the memory location.

Assuming there is no congestion at the memory output, a request packet is clocked into the memory on the one cycle, and the acknowledge packet clocked out on the next. The memory thus has a maximum throughput of one packet per cycle.

3.4 Comments

This chapter has described the Eppl design and presented simulation results for a single processor and isolated network. The processor simulation provided performance bounds for given meta-instruction mixes and memory latencies, and the network simulation for given queue sizes, loads and hot-spot percentages. Without previous experience it is difficult to predict what the instruction mix and memory access pattern of parallel programs executing on the Eppl will be, since relevant statistics are not available. The frequency of load and f&a memory accesses to hot-spots is particularly important, since it will affect the network performance. Such information can only be provided by detailed simulation of the whole Eppl system, and such a simulator is described in the next chapter.

Chapter 4

Epp1 simulator

The Epp1 simulator is a distributed time-driven, instruction level simulation of the Epp1 multiprocessor design described in the previous chapter. To increase the simulation speed and the size of Epp1 system that can be simulated, the Epp1 simulator has been implemented in Occam on a transputer based, distributed memory multiprocessor. The largest Epp1 system that has been simulated is a 256 processor system using 128 transputers. The first section of this chapter briefly reviews Occam and the transputer. The following sections describe the machine on which the Epp1 simulator runs, how the simulator is parallelised, the user-interface of the simulator, and how the Epp1 components are simulated. The last section presents some performance results for the simulator.

4.1 Occam and transputers

Occam [115] is a strongly typed language based on CSP [116], and includes constructs to explicitly define parallelism and communication between processes. An Occam process can be anything from a single arithmetic statement to a large procedure, and processes communicate by passing messages through named channels. Only two processes may communicate through a channel, in one direction, using

blocking input and output message-passing commands.

Statements or procedures to be executed in parallel are included in the body of the PAR construct. Equivalently, statements to be sequentially executed must appear in the corresponding SEQ construct. A non-deterministic element is introduced by the ALT construct, the body of which contains a number of guarded input commands. When a communication occurs on one of the input channel guards, the statements subordinate to the input are executed. Each of the constructs may also be used in a replicated form. In the case of a replicated PAR the given number of copies of the body of the construct are executed in parallel.

In the transputer implementation of Occam [117] the level of parallelism must be statically defined at compile-time, and therefore the replication count in a replicated PAR must be constant. The placement of processes onto transputers is also static, and the top-level *configuration* part of the program specifies which processes are to be placed on which transputers using the PLACED PAR construct. Which Occam channels are to be associated with which transputer links is also defined at this level.

The transputer [118] is a single-chip microprocessor intended for parallel processing and embedded real-time applications. The highest performance T800 transputer contains 4 Kbytes of local on-chip RAM, a 64-bit floating point unit, a 32-bit microcoded processor, an external memory interface, and four 20 Mbits/sec serial communication links. The transputer design is optimised for executing Occam, and provides instructions and hardware support for Occam process communication and management. The Occam input and output instructions, and the PAR and ALT constructs compile almost directly into equivalent transputer assembly code instructions. The processor maintains a queue of executable processes, and context-switches the current process when it executes a communication operation or its time-slice has expired. The register set of each process is small so that context-switching is efficient.

The Occam channels are implemented using memory locations, but they can

also be mapped onto the transputer links. A number of transputers can be connected together using the links to form a distributed memory multiprocessor. Current transputers provide no hardware support for general, global communication between processes in such a multiprocessor. A process running on a transputer can only communicate directly with other processes on the same transputer, or with the links. Programs which require global routing of messages must use some sort of communications harness to route messages between processes on different transputers.

4.2 Simulation engine

The Eppl simulator runs on a large Meiko Computing Surface called the Edinburgh Concurrent Supercomputer (ECS) [119, 120]. This is an MIMD distributed memory multiprocessor constructed from transputers. The current ECS configuration, of which an overview is shown in Figure 4.1, contains over 400 T800 transputers with 4 Mbytes of external memory each. The ECS is a multi-user system, and provides a version of UNIX and a distributed file system, and can be programmed using Occam, and C or Fortran with added message-passing procedures.

The transputers in the ECS are divided up into about 20 single-user domains, each of which contains a fixed number of transputers. Each domain consists of a host transputer and a number of user transputers. The host transputer runs the system software and provides an interface to the i/o devices, while the user transputers are available for user programs. The domains range in size from 1 user transputer, through 5, 17, 32, 65, and 132 user transputers. Some of the domains also have an attached graphics board and colour monitor.

The user transputers in a domain can be connected in almost any topology using the transputer links. The Meiko hardware provides a number of link switching chips to which all the transputer links are connected. These switching chips

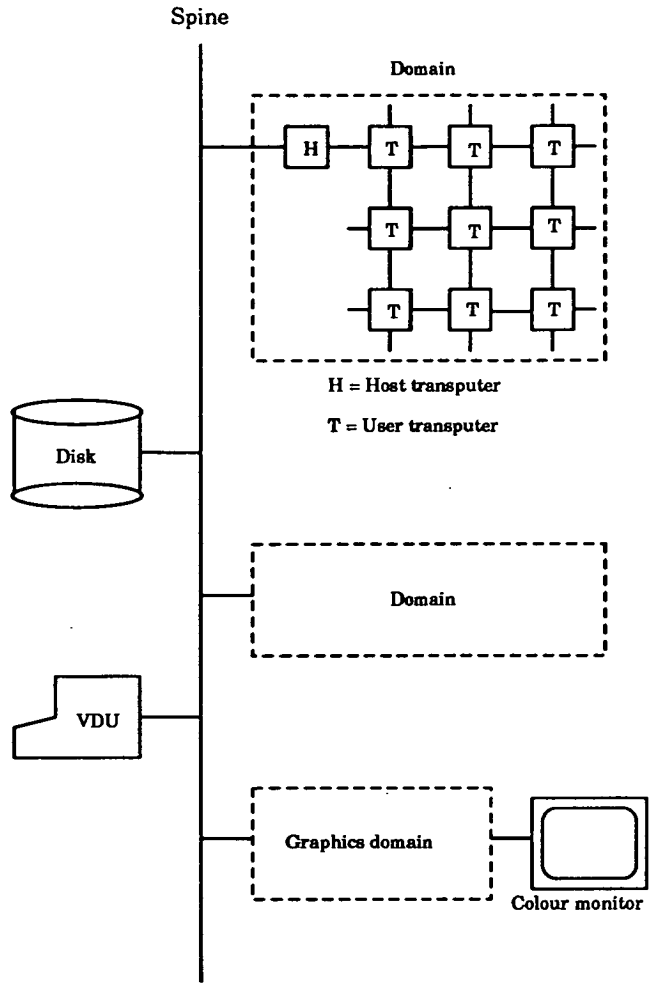


Figure 4.1: Overview of the domain organisation of the ECS.

can be electronically configured before a program is loaded, so that the user transputers are connected in the correct topology. This topology is static, and is not changed during a program run. The minimum connectivity necessary is a path through all the user transputers used, so that the program can be booted onto the transputers. All of the transputers in a domain are additionally connected by a memory mapped 8-bit bus called the supervisor bus. This bus has a low bandwidth and is used mainly for system purposes, such as configuring the switch chips. It can also be used for sending debugging messages from the user transputers in a domain to the user's terminal, via the domain's host transputer.

For most programs one of the user transputers is designated the master trans-

puter, and runs the control and user-interface part of the program. The master transputer is connected to the host transputer by one of its links (the host-link), so it can interface to the i/o devices. The other user transputers run the computation part of the program, and are referred to as slave transputers. In a typical task-farm application for example, the master sends work to be done to the slaves, which then return the result and wait for new work.

4.3 Simulator structure

The Eppl simulator is written entirely in Occam, and can be divided into three parts: the user-interface, the simulation modules and the communications harness. The processor, network switch and memory of the Eppl system described in the previous chapter are each simulated by an Occam procedure (or module). The simulator is parallelised by replicating and distributing the modules across a transputer domain, and the modules communicate by message-passing; the message packets being routed around the domain by the communications harness.

4.3.1 Domain topology

To execute the Eppl simulator, the transputers in the chosen domain are first configured in a de Bruijn graph or d-shuffle topology [121] using the transputer links. This graph has a logarithmic inter-node distance with fixed node degree, and has better performance for generalised communication than other topologies, such as a torus for example [122, 123]. The topology of an 8 slave domain is shown in Figure 4.2, in which the slave transputers are numbered 0–7. The de Bruijn graph has been slightly augmented to provide two end-around connections, one of which connects the master transputer. The host transputer, to which the master is connected, is not shown.

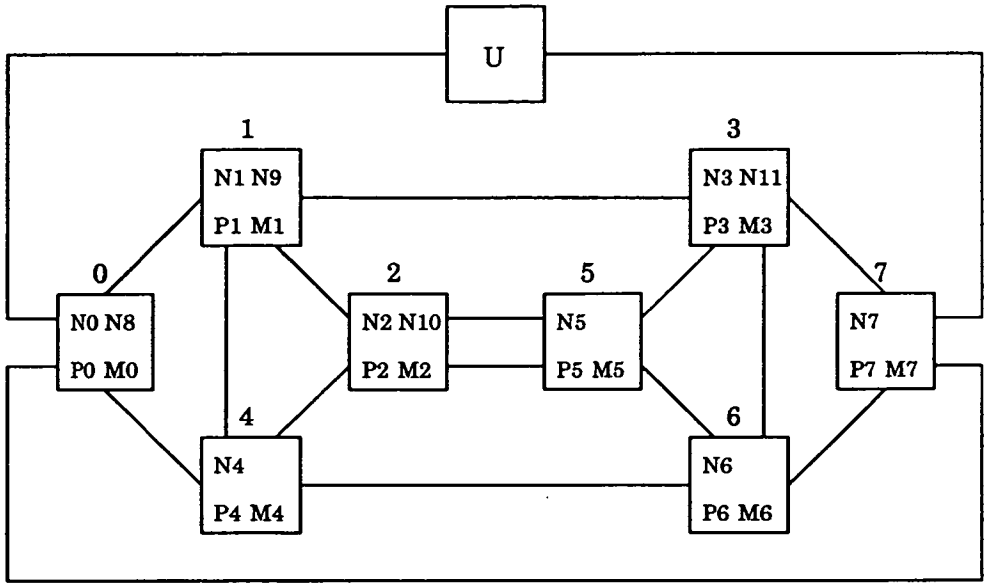


Figure 4.2: Topology and module distribution of an 8 processor Eppl system on an 8 transputer domain.

4.3.2 Module distribution

For the simulation of a p processor Eppl system a domain of maximum p slaves can be used. To balance the load of each slave the simulation modules are evenly distributed across the domain. If the slaves are numbered from 1 to p, then processor i is placed on slave i, switch j is placed on slave j mod p, and memory k on slave k. Table 4.1 shows how many modules there are per slave for increasing Eppl system sizes, using the largest domain possible in each case. Where the number of switches does not divide evenly over the number of slaves, half the slaves have one more switch than the other half (these latter slave transputers actually contain a dummy switch instead of a real switch, which swallows any packets it receives). In Figure 4.2 an 8 processor Eppl system (refer to Figure 3.1) is shown distributed across an 8 slave domain. The user-interface runs on the master transputer (and is shown as U). The processor (P_0–P_7), switch (N_0–N_{11}) and memory (M_0–M_7) modules run on the slave transputers.

Due to the static parallelism of Occam a separate version of the simulator must be compiled for each different system and domain size. The simulator has

Processors	Nodes	Memories	Total modules	Slaves	Modules per slave
1	0	1	2	1	2
2	1	2	5	2	2.5
4	4	4	12	4	3
8	12	8	28	8	3.5
16	32	16	64	16	4
32	80	32	144	32	4.5
64	192	64	320	64	5
128	448	128	704	128	5.5
256	1024	256	1536	128	12

Table 4.1: Number of modules with increasing simulation size

two parameters *number of processors* and *number of slaves* which are simply set to the relevant values before compiling. The size of the memory array in each memory module is also set before compilation, and is calculated so that the total amount of memory in the system is 128 Kbytes for any given system size (this is sufficiently large for all the benchmark programs described in the next chapter). A number of other simulator parameters can be varied at run time and are described below. The largest Eppl system that has been simulated is 256 processors using 128 slave transputers. Larger system sizes could be simulated, but the simulation becomes slow because of the high load on each slave.

4.3.3 Communications harness

A general transputer communications harness called Tiny [124], developed at Edinburgh University, is used to route packets around the domain. Tiny provides topology independent routing of variable sized packets between named processes in a domain. Each transputer in the domain has a number of harness processes running on it as well as the simulator modules. When the harness is started up it first explores the domain topology, and builds up routing tables containing the shortest paths between all the transputers. The processes using the harness must be given a unique domain-wide integer identifier, which is used to address

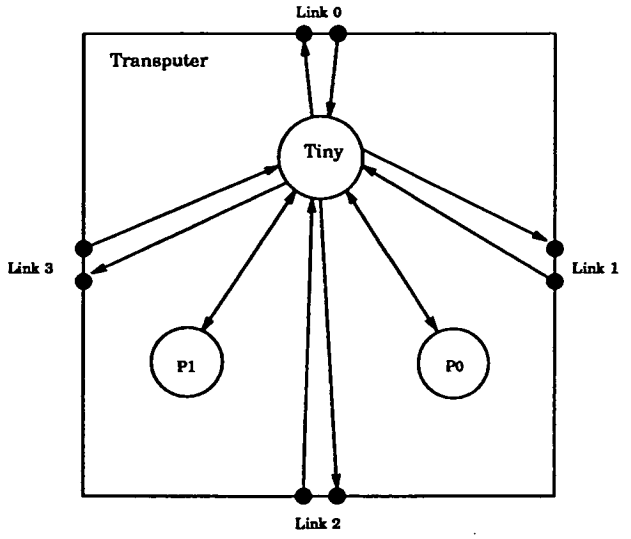


Figure 4.3: Connection of Tiny harness and user processes within a transputer.

packets. A number of packet read and write functions are provided with Tiny. They differ in the routing strategy used, which can be either sequential, adaptive or broadcast. In sequential routing all packets are routed by the same path, and the order of sending is maintained. In adaptive routing the packet is routed by the least congested path, and subsequent packets may arrive out of order. In broadcast routing the packet is broadcast to all other processes. This is done efficiently since the harness builds a broadcast tree from the source transputer to all the destination processors. The packet write functions are non-blocking, and packets arriving at a destination transputer are buffered until the receiving process reads them.

Each simulator module has a number of Occam channels that make up its interface. These channels are connected to the harness as shown in Figure 4.3, for the two modules *P0* and *P1*. The harness receives messages from a module's output channels and routes them to the given input channel of the destination module. Each module knows its own logical module number, which is used along with the output channel number as arguments to a mapping function. This returns the logical module number of the connected module and the input channel number, using the permutation functions described in Chapter 2. The destination

module number is then converted to a Tiny process number, which is used to address the packet. Currently all packets are 2 words long, and contain the input channel number and a data word. Longer messages must be sent sequentially (this is a limitation of the simulator not the harness).

4.4 The user-interface

The user-interface of the Eppl simulator runs on the master transputer, inputs commands from the user or a batch file, and controls the simulated Eppl system in the manner of a host computer. Commands are sent to the modules using control packets, which are output from the user-interface's *control-out* channel and routed to the *control-in* channel of the modules. Acknowledgement packets from the modules' *control-out* channels are received on the user-interface's *control-in* channel. The user-interface also has an *io-in* channel which is used for receiving asynchronous messages from the modules, and various other channels to connect it to the user terminal and the filing system via the host transputer. These connections are shown in Figure 4.4, where the user-interface is shown connected to a processor, switch and memory module.

The main commands of the user-interface are shown in Table 4.2, and are described below. There are a number of other commands which are used for debugging purposes, such as *setdebug* which instructs specified modules to print out debugging messages over the supervisor bus, and *trace* which is used to trace the path of specified Eppl processes, or to trace access to specified Eppl memory locations.

The user commands are executed by the user-interface using module control packets. These are either sent to specific modules or broadcast to all modules, depending on the command. Some of the commands simply send the corresponding control packet, while others use combinations of control packets in different ways. The main control packet types are shown in Table 4.3, and are also de-

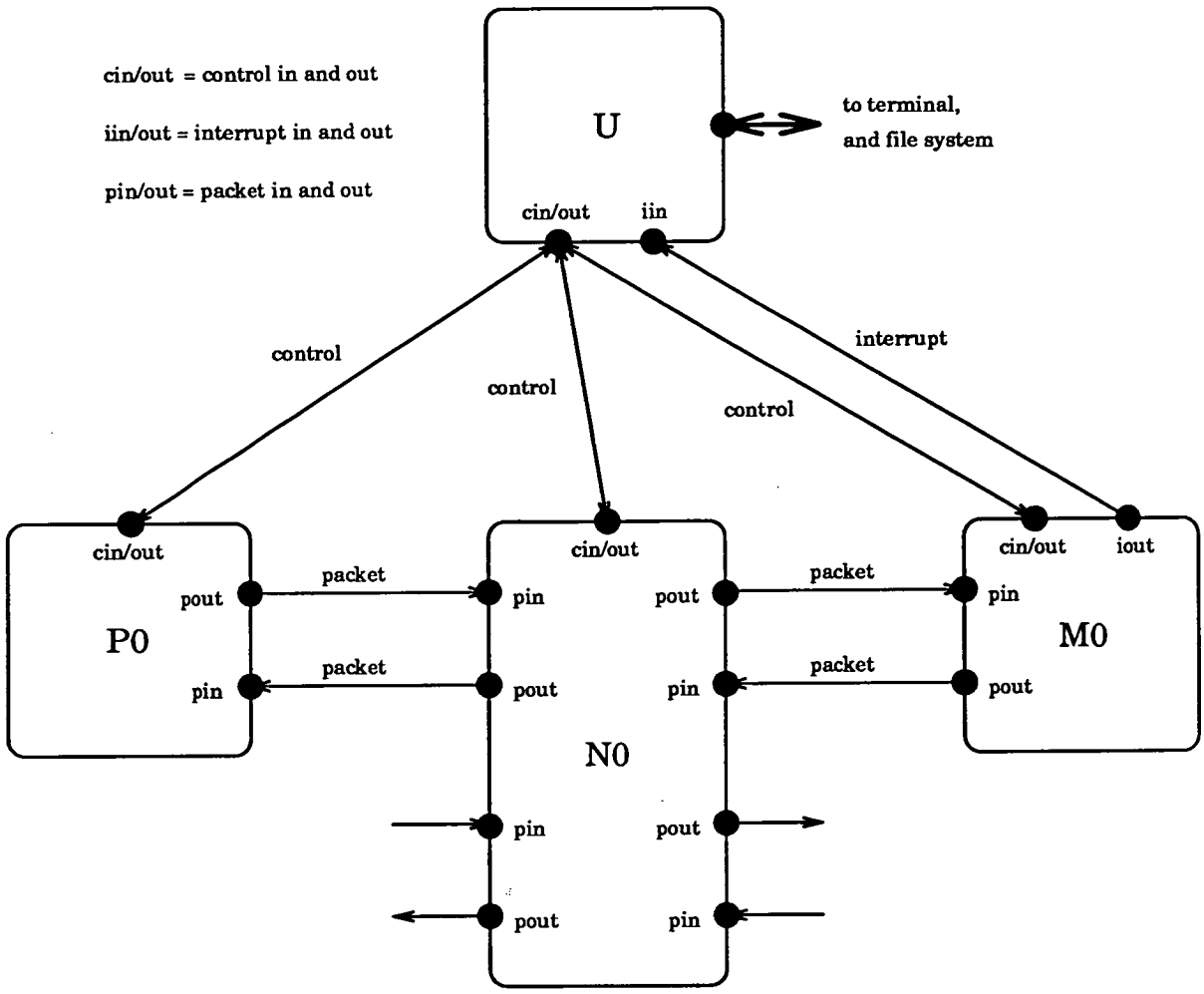


Figure 4.4: Channels connecting the user-interface and three modules.

scribed below.

4.4.1 System reset and initialisation

A number of parameters of the simulated Eppl system can be varied at run-time. Currently these are: the interleaving level which is the number of processes executing per processor, the combining level used in the switches, the size of the queues and wait-buffers in the switches, and the relative clock speed of the three module types. The default values of the system parameters are changed using the *Setup* command.

When the simulator is started up or the *Reset* command is executed, the

Command	Description
Clock	single-step 1 cycle
Display	show the internal state
Memory	load a program, dump, or poke memory
Nclock	clock for n cycles
Reset	reset system
Run	run in either timed or clocked mode
Setup	set default parameters
Toggle	auto-display
Quit	terminate the simulator

Table 4.2: User commands

modules are reset using the default values of the system parameters, and the memory is cleared (zeroed). The modules are reset by sending a *reset* control packet and the relevant parameters to each module, as described in Section 4.5.

4.4.2 Loading a program

A program is loaded into the memories using the *Memory (load)* command. The program is read from a file in ascii-hex format, produced by the Eppl assembler, and consists of a number of code blocks. Each block contains the starting address of the block, the size of the code in words, and the machine code itself. The code usually starts at the reset location (memory address 16), from which the first instruction is read by the processors after a reset. To load a memory word the given address is decomposed into a memory number and an offset, and that memory module is sent a *load* control packet with the offset and code word.

4.4.3 Program execution

After the program is loaded the simulator modules are clocked to begin execution. The *Clock* and *Nclock* commands are used to clock the modules explicitly for either a single cycle or for a given number of cycles respectively, and are used mostly in the development and debugging of programs.

The *Run* command is used to clock the modules implicitly (and continuously)

Control packet	Description
clock	clock for specified number of cycles
interrogate	return specified state information
load	load word into specified memory location
poll	return current clock count
reset	reset and initialise state
run	clock continuously until interrupted
stop	interrupt run
quit	terminate process

Table 4.3: Module control packets

for an unspecified number of cycles, and has two forms: timed or clocked. The timed form allows the clocking to be paused every given (real) time interval, for any internal state or utilisation statistics of the simulated system to be displayed, after which clocking is resumed. The clocked form allows the clocking to be paused every given number of clock cycles. The *Run* command finishes when an interrupt is received from either the modules or the user, and is used for executing complete programs since it is generally not possible to predict when the program will finish. A mechanism is provided which allows the program itself to send an interrupt to the user-interface indicating that the program has finished. Clocking and interrupts from a modules viewpoint are described further in Section 4.5.

4.4.4 Displaying state and statistics

The *Display* command is used to show the internal state or utilisation statistics of the modules on the user terminal. The internal state of the modules is accessed by sending an *interrogate* control packet and parameters which specify the particular piece of information. The state and statistics available in each module are described in the next section. The *Toggle* command is used to enable automatic execution of the *Display* command after a clock cycle, or during a run.

The execution of the Eppl system can also be visualised on a colour monitor (if a graphics domain is used). The display represents each of the modules as a box, and the colouring of the box is used to indicate some given value of utilisation.

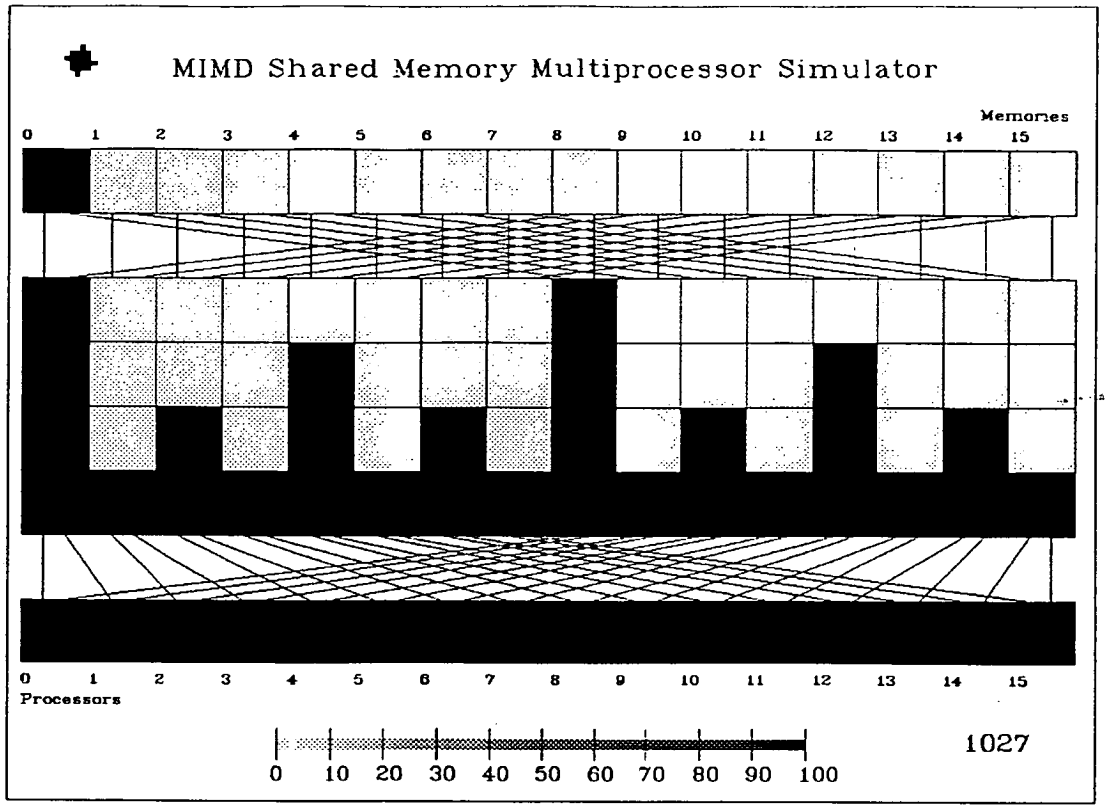


Figure 4.5: Visualisation of a hot-spot in a 16 processor system

The display can be updated each cycle, allowing the memory access pattern of a program to be seen, for example. Effects such as tree saturation (Section 2.2.4) can be observed in this way, as shown in Figure 4.5. This shows a 16 processor simulation in which an artificial hot-spot has been created at memory module 0. The requests and acknowledge halves of the network are displayed separately, and the dark shading of the modules indicates the presence of a packet (the scale at the bottom is not meaningful in this display mode). The current clock count is shown in the lower right hand corner.

4.5 The modules

The three module types simulate the Eppl processor, switch and memory. The Occam code for each module is serial, apart from packet input-output, which reduces the number of wasted cycles due to frequent context-switching on the

transputers. The body of each module consists of two parts: a control part which executes the commands from the user-interface, and a simulation part which simulates the corresponding Eppl component.

The modules communicate with the user-interface using the *control-in/out* channel pair as shown in Figure 4.4, and execute a continuous loop waiting on the *control-in* channel for a control packet from the user-interface. When a control packet is received the command is executed, and possibly some result packets are sent back to the user-interface. Each module also has a number of ports for inter-module communication, equivalent to those described in Section 3.2.1. An input port has channels called *ready-out*, *valid-in* and *packet-in*, and an output-port has channels called *ready-in*, *valid-out* and *packet-out*. The *ready-in/out* and *valid-in/out* channels (which are not shown Figure 4.4) transmit and receive single word packets containing a boolean value, and the *packet-in/out* channels transmit and receive four-word memory access packets (although each word is transmitted sequentially).

Each module has a number of parameters which are initialised on reset, and some state and utilisation statistics which can be accessed by the user-interface. On reset the processor module is sent the interleaving level and the processor clock factor. The state of the processor that can be accessed includes: the contents of the pipeline stages, the contents of the register file, and the context of each process. The statistics recorded include: the number of instructions executed, the number of memory requests issued and the average latency of memory requests.

The switch module is sent four parameters on reset: the combining level, the queue size, the wait-buffer size and the switch clock factor. The request and acknowledge queues are both set to the same size. The accessible state is: the contents of all the latches, queues and wait-buffers in the switch. The statistics recorded include: the number of packets received and transmitted, the number, type and level of packets combined, and the number of request, acknowledge and wait-buffer conflicts.

The memory module is sent only the memory clock factor on reset, the memory size having been defined at compile-time. The accessible state is the contents of the latches and the memory locations. The only statistic recorded is the number of memory accesses.

The parts of the module code which simulate the Eppl components were developed before the rest of the simulator. To ensure that they worked correctly simple tests were performed, such as verifying the correct execution of all the instructions in the processor. When a complete system was assembled the execution of simple test programs was analysed, and at a later stage also the execution of the benchmark programs described in the next chapter.

4.5.1 Clocking

When a *clock* control packet with an argument is received the simulator part of the module is invoked to execute the given number of cycles, after which an acknowledgement is returned to the user-interface. The modules are not clocked by repeatedly broadcasting single *clock* packets, which would be inefficient as shown in the performance section below. With a *run* control packet, the modules start clocking themselves as above, but only finish when a *stop* control packet with an argument is received. The *control-in* channel is therefore polled every cycle to see if a packet has arrived.

During clocking the modules are synchronised by the exchange of packets between ports, and therefore clock at the maximum rate possible. Because of the loosely-coupled synchronisation between the modules, each module can be one cycle out of step with a connected module (the simulation is loose time-driven). In a p processor simulation it is possible for a processor to be up to $(\log p)+1$ cycles out of step with a memory (the module that is furthest from the processors), and *vice versa*. The modules therefore need to be synchronised after a run, so that they have all reached the same clock cycle. To do this the user-interface first sends a *poll* control packet to all the modules (this does not interrupt the run),

which then return their local clock count to the user-interface. The maximum value returned is added to the maximum separation in cycles possible (as defined above), and sent to the modules as an argument to the *stop* control packet. The modules then stop when they reach this combined clock count, and return an acknowledgement to the user-interface. The *poll* control packet is also used to see how many cycles have been executed during a run.

4.5.2 Interrupts

A run may be paused periodically using the *poll/stop* mechanism, or may be stopped by user keyboard input. The program being executed by the simulator also has the capability to stop the run, by sending an interrupt to the user-interface. The first sixteen logical memory addresses (#00 to #0F) of the Eppl memory space are designated as special memory mapped i/o locations. By writing to the designated interrupt location (#03), the program causes the memory module to which that address is mapped to send an interrupt packet to the user-interface, through its *io-out* channel. When the user-interface receives an interrupt packet during a run it stops the run as described above, otherwise the user is simply alerted that an interrupt has been received.

Interrupts are used to stop clocking after a program has finished. Generally it is the last executing process to complete its work that invokes the interrupt. The current clock period of the memory module is sent with the interrupt packet, and this value is used as the completion time of the program. Because the modules are clocked for a further number of cycles to synchronise them, the statistics displayed after the run are not exactly the same as those at the moment the program finished.

4.5.3 Clock factor

Usually the modules are set to clock at the same rate, but using the module clock factor parameter they can be given differing clock rates. The module clock factor dictates how many external clock cycles equal one internal clock cycle. For instance if the switches and memories are to clock twice as fast as the processors, then this is achieved by making the processors half as fast as the other modules by setting the processor clock factor to two. The processors will then only execute an internal cycle every two external clock cycles. The connected modules still expect to receive *ready* and *valid* signals (packets) every cycle, therefore a dummy clock routine is executed which outputs *valid* false and *ready* false to the connected modules.

4.6 Simulator performance

Generally a p transputer domain is used when simulating a p processor Eppl system up to 128 processors, although it is possible to use a smaller domain. With a small domain size the number of modules executing on each slave is larger, but as the domain size is increased the average distance between the slaves increases, resulting in higher message transmission times. This section presents results for the relative speed-up of the simulator with increasing domain size, and looks at the simulator as a parallel application in its own right.

Table 4.4 shows the cycle time, relative speed-up and efficiency of simulating a 64 processor system with increasing domain sizes of 4 to 64 slaves. Due to a limitation of the harness the simulation could not be carried out for 1 and 2 slaves. The cycle time is the time in seconds to execute one clock cycle, and is averaged over the execution of 1000 cycles. The speed-up is calculated relative to the 4-slave cycle time, and the efficiency is the ratio of actual speed-up to the linear speed-up (shown in brackets). During clocking the processes are simply executing no-operation instructions. A comparison of the cycle times of

Slaves	Cycle time(s)	Speedup (ideal)	Efficiency
4	0.31	1.0 (1)	1.0
8	0.17	1.8 (2)	0.89
16	0.09	3.3 (4)	0.81
32	0.05	5.9 (8)	0.74
64	0.03	10.1 (16)	0.63

Table 4.4: Performance of simulator, simulating a 64 processor system, with increasing domain size

the benchmark programs under similar conditions did not show any significant difference to executing no-operation instructions, although the memory access pattern of the program can affect the simulation speed as discussed below. Using 64 slaves the simulator can execute about 33 simulated cycles per second, which is about 10^6 times slower than one might expect a hardware implementation to perform. As the domain size increases, the efficiency progressively falls to 63% for 64 slaves, which indicates that the simulator becomes communication bound using the larger domain sizes.

As mentioned above the modules are not clocked by repeatedly broadcasting a *clock* control packet for each clock cycle. The overhead introduced by the repeated broadcasts and acknowledgements would certainly slow the simulator down considerably. To compare the speed of broadcasting repeated *clock* control packets with no argument (*i.e.*, clock for one cycle only) to using a single *clock* control packet with an argument, a 64 processor system was clocked for 1000 cycles on a 64 slave domain using both methods. The repeated broadcast results in an average clocking time of 0.0987 seconds per cycle, while the more efficient method results in a clocking time of 0.0304 seconds per cycle. The repeated broadcast method corresponds to the barrier synchronisation between each clock cycle used in a shared memory time-driven simulation of a multistage network [100]. In the Eppl simulator the modules are synchronised by the exchange of packets between ports, and can therefore clock at the maximum rate possible. Global synchronisation of the modules is only required when the system state is to be

accessed.

Because the pattern of packet traffic in the domain is dependent on the program being executed by the simulator, the performance can vary with different programs. For example if the program exhibits hot-spot behaviour, in which a large number of memory requests are directed at one memory location, then the number of packets routed to the memory module containing that memory location increases. A hot-spot in the program can therefore not only increase the execution time of the program (measured in clock cycles), but also the execution speed of the simulator (measured in cycles per second). A similar observation was reported in the shared memory simulation cited above.

4.7 Comments

A more recent version of the Tiny communications harness [124] can monitor the transputer link utilisation, allowing potential bottlenecks to be discovered. As the speed-up results show, the simulator is communication bound and its performance could be improved by reducing the number and size of packets transmitted between the modules, and by optimising the placement of the modules, which would reduce the average distance each packet has to travel. Due to time constraints these improvements were not attempted, and work was concentrated on evaluating the performance of the Eppl design executing three assembly code benchmark programs which are described in the following chapter.

Chapter 5

Benchmark programs

Because the Eppl simulator models the Eppl design at the instruction level it can be used to execute realistic parallel programs. This allows the development and evaluation of parallel algorithms for this kind of shared memory architecture. In the simulation experiments described in the next chapter three parallel benchmark programs are used as system loads, under the assumption that they generate memory access patterns similar to those of real parallel applications. Since no high-level language compiler for the Eppl instruction set was available all programming of the Eppl has been done using the Eppl assembly code defined in Section 3.1.6.

This chapter describes the algorithms and implementation of the three benchmark programs. The first section explains the choice of algorithm, the next section describes the process control mechanisms used in the programs, and the following sections describe the algorithm and implementation of each of the programs. The assembly code of each program is shown in Appendix A.

5.1 Choice of algorithms

The three algorithms used for the benchmark programs are: parallel matrix multiply, bitonic merge sort and Moore shortest path. These algorithms were chosen because they were readily available in the literature [125, 126, 127], and they satisfied the constraints for execution on the Eppl simulator. Most of these constraints were of a practical nature, for example:

- the algorithms should be short and easy to code in assembly language.
- they should exhibit plenty of obvious parallelism, enough to keep all the processes busy.
- they should have reasonable time and space complexity, so that the programs do not take too long to run and do not need too much data space. On the other hand they should be long enough to be interesting.
- they should not require floating point arithmetic, as the simulated Eppl processors only support integer arithmetic.
- they should not require dynamic process creation or memory allocation, since there is no operating system to provide these functions.

The implementations of the algorithms each exhibit a different mode of execution, and use different parallel constructs and data structures as described in the next section. Therefore each program should present a qualitatively different load to the system when executed. These algorithms have been frequently used in the evaluation of other multiprocessor designs, for example a variant of the Moore algorithm used here has been implemented on the HEP [128], and a paracomputer simulator [76]. The term "benchmark" is used loosely here though, since no direct comparisons with other machines are made here.

5.2 Process control

The benchmark programs use a single-program-multiple-data (SPMD) execution model [129], in which all the processes execute the same shared code, but are each given a different set of data to work on. All the processes are spawned at the beginning of the program, and continue to execute until the program is finished. Each process performs a repeated cycle of fetching some work, executing that work, returning the results, and then fetching more work. When all the work has been completed the processes terminate, and the program is finished. The SPMD model is simple to implement in a shared memory multiprocessor, and the algorithms used here fit the model well. Process control in the benchmark programs consists of spawning, synchronising and terminating processes. Since there is no Eppl operating system all process control must be coded into the programs, and the fetch-and-add synchronisation algorithms described in Section 2.3 form the basis of process control for the benchmark programs as described below.

Spawning involves assigning work to a process. In the matrix multiply and bitonic merge sort programs the test-modify-retest functions (Section 2.3.3) are used to initially give each process a unique integer identifier which the process uses to index into the data arrays. The Moore shortest path program uses a parallel queue (Section 2.3.4) as a central data structure, and processes fetch work from this queue. Since all the processes are self-scheduled [6], the load on each processor is automatically balanced.

Apart from the implicit synchronisation which occurs during process spawning and termination, explicit synchronisation used to control access to shared variables occurs only in the Moore shortest path program. The blocking `p()` and `v()` semaphore functions (Section 2.3.1) are used to lock entries in an array of shared variables which are updated in parallel.

Termination in all three programs is detected using a termination count which counts how much work has been completed. When a process finishes its portion of work it increments the termination count, and then loops to the start of the

program to see if there is more work to be allocated. This will be the case in programs where the program parallelism (the amount of work to be done) is greater than the machine parallelism (the number of processes executing). When the termination count has reached the given maximum value then all the work has been completed, and the program is finished. All the processes then loop forever doing nothing. The end of the program is identified by the last process sending an interrupt to the user-interface (Section 4.5.2). In the pseudo-code descriptions of the programs below, the function `done()` is assumed to send the interrupt.

Because the Eppl system provides many executing processes, context-switching of processes in the usual operating system sense is assumed to be unnecessary and is not used. This greatly simplifies the amount of process control necessary. In the assembly code implementations of the benchmark programs the process context size is kept very small, and all private variables are stored in registers. All function calls are also expanded in-line by hand, so that no manipulation of stacks or swapping of register sets is necessary. These optimisations have been made to minimise the execution time of the benchmark programs.

To test the correctness of the benchmark programs the results of executing the programs on the Eppl simulator, with various data sets, were compared with the results of executing serial versions of the same algorithms written in C and executing on a workstation.

5.3 Matrix multiply

The matrix multiply program calculates the product of an $l \times m$ integer matrix A with an $m \times n$ matrix B to get an $l \times n$ result matrix C. Each entry of the result matrix is given by:

$$c_{i,j} = \sum_{k=0}^{m-1} a_{i,k} b_{k,j} \text{ where } i = 0, l-1 \text{ and } j = 0, n-1.$$

```
int i,j,k;
int a[l][m],b[m][n],c[l][n];

forall(i=0;i<l;i++)
   forall(j=0;j<n;j++){
      c[i][j]=0;
      for(k=0;k<m;k++)
         c[i][j]=c[i][j]+a[i][k]*b[k][j];}
```

Figure 5.1: Matrix multiply algorithm

5.3.1 Parallel algorithm

The parallel matrix multiply algorithm is shown in pseudo-C form in Figure 5.1. The `forall()` statement is a parallel equivalent of the normal `for()` statement and spawns the given number of processes, assigning each process a unique value of the index. In this algorithm each entry of the result array `c[][]` is evaluated in parallel. There are three loops, the outer two loops are parallelised and spawn $l \times n$ processes, and each process executes the inner loop sequentially summing the m products. It would be possible to parallelise the inner loop as well, but this would require an indivisible array update or some form of mutual exclusion. It would be possible to use fetch-and-add to do the summing directly [79], but this is not a general solution (there is no arithmetic error detection in the combining hardware of the network switches). The matrix multiply algorithm has a parallelism of $l \times n$, and time complexity $\Theta(m)$ (assuming constant spawning time).

5.3.2 Implementation

The matrix multiply program is implemented as shown in Figure 5.2, using normal C syntax, and is intended to be equivalent to the actual assembly code shown in Appendix A. On reset each of the processes begins executing the function `mmult()`. The variables declared above the function are globally accessible, and the variables declared inside the function are private to the process. The constants

`l`, `n` and `m` contain the size of the arrays, the spawning count `p` and termination count `d` are initialised as shown, and the data arrays `a[] []` and `b[] []` are assumed to be initialised with the data.

The body of the function is an infinite `while()` loop. Within the loop the process first calls `tdr()`, the blocking test-decrement-retest function defined in Section 2.3.3, to get a unique identifier `lp` from which it calculates the array indices `i` and `j`. The process then executes `m` iterations of the sequential `for()` loop, multiplying entries from arrays `a[] []` and `b[] []` and summing the result in temporary variable `t`. The result of the summation is then written to the result array entry `c[i] [j]`, and the termination count `d` is decremented using the fetch-and-add function `f&a()`. If the result is zero the `done()` function is called to send an interrupt indicating the end of the program. The process then loops back to the beginning, and if the parallelism of the result array is larger than the number of active processes then the process is given a new entry to evaluate, otherwise the process blocks forever in function `tdr()`. This concludes the description of the matrix multiply program.

5.4 Bitonic merge sort

The bitonic merge sort is a parallel sorting algorithm based on the recursive application of the bitonic merge [130]. The sequence of numbers

$$A = \{a_1, a_2, \ldots, a_n\}$$

is bitonic if either:

1. there exists an integer i, where $1 \leq i \leq n$, such that:

$$a_1 \leq a_2 \leq \cdots \leq a_i \geq a_{i+1} \geq \cdots \geq a_n$$

2. the sequence A can be cyclically shifted to satisfy the above condition.

```
int p=1*n-1, d=1*n-1;
int a[1][m], b[m][n], c[1][n];

void mmult(){
   int lp, i, j, k, t;

   while(1){
      lp=tdr(&p,-1);
      i=lp/n;
      j=lp%n;
      t=0;
      for(k=0;k<m;k++)
         t=t+a[i][k]*b[k][j];
      c[i][j]=t;
      if(f&a(&d,-1)==0)
         done();}}
```

Figure 5.2: Matrix multiply program

A bitonic sequence of length n can be sorted in $\log_2(n)$ stages using a bitonic merge. An example of sorting a bitonic sequence of 8 integers in 3 stages is shown in Figure 5.3. Each stage of the bitonic merge consists of a shuffle permutation (Section 2.2.1), followed by a compare-exchange operation. In this case the compare-exchange sorts each pair of numbers into increasing order (as indicated by the $>$ in each box), so the final sequence is sorted in increasing order from top to bottom.

To sort an unordered sequence of integers A of length n (assuming n is even), the list A is first divided into $\frac{n}{2}$ unsorted bitonic subsequences of length 2:

$$A = \{\{a_1, a_2\}, \ldots, \{a_{n-1}, a_n\}\}$$

Each of these subsequences is labelled $s_{i=1,\frac{n}{2}}$:

$$A = \{s_1, \ldots, s_{\frac{n}{2}-1}, s_{\frac{n}{2}}\}$$

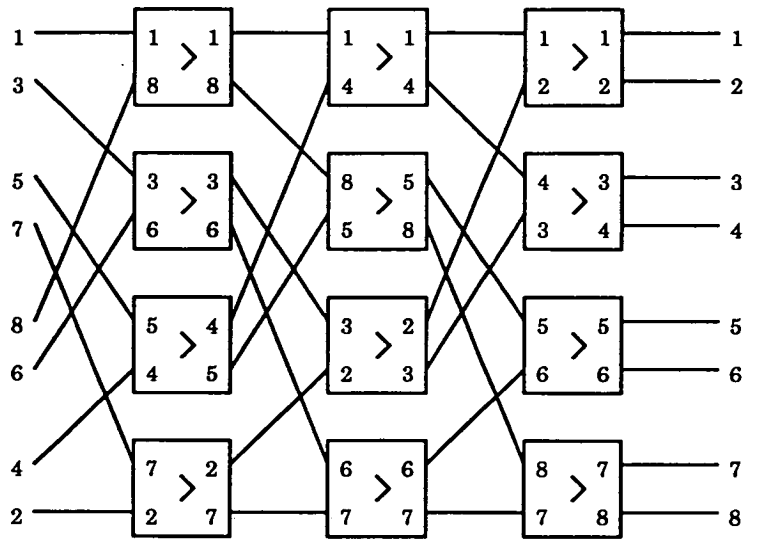


Figure 5.3: Example bitonic merge of a bitonic sequence of 8 integers

The subsequences $s_{odd(i)}$ are then sorted into ascending order and the subsequences $s_{even(i)}$ into descending order, using bitonic merge. The result is $\frac{n}{4}$ bitonic subsequences of length 4. By repeatedly applying bitonic merge to larger and larger subsequences the whole sequence A is eventually sorted. To sort a sequence of length n requires $\log_2 n$ iterations of bitonic merge.

5.4.1 Parallel algorithm

A parallel algorithm for bitonic merge sort is shown in Figure 5.4. The n integers to be sorted are stored in array `a[]` (again assume n is even). The first sequential `for()` loop executes $\log_2 n$ iterations, each of which is a bitonic merge. The second sequential `for()` loop executes the i stages of the bitonic merge. Since the length of each subsequence to be merged is 2^i, then $\log_2 2^i (\equiv i)$ stages of merge are required to sort the subsequence.

In each stage of the merge n parallel processes are spawned by the first `forall()`, and execute a shuffle permutation. The function `shuffle()` takes an index to the array (`k`), it shuffles the index (the shuffle being dependent on the iteration of the merge), and moves the entry from its original index to the new index. Note that since the `shuffle()` functions are spawned in parallel

```
int i, j, k;
int a[n];

for(i=1;i<=log2(n);i++)
   for(j=1;j<=i;j++){
      forall(k=1;k<=n;k++)
         shuffle(a,k,i);
      forall(k=1;k<n;k+=2)
         compare_exchange(a,k,i);}
```

Figure 5.4: Bitonic merge sort algorithm

there may be conflicts for the array entries, therefore some form of conflict resolution is assumed. The second `forall()` then spawns $\frac{n}{2}$ processes which execute a compare-exchange operation. The `compare_exchange()` function is given an index to the array (`k`), and compares and perhaps exchanges entry `a[k]` and `a[k+1]`. The direction of the exchange (greater than or less than) is dependent on the iteration of the sort.

To sort a sequence of $n = 2^k$ numbers a total of $\frac{k}{2}(k+1)$ stages of bitonic merge are needed, and the algorithm has a time complexity of $\Theta(\log^2 n)$ (assuming constant spawning time) [125]. A parallel quicksort algorithm for a shared memory multiprocessor with a combining network [131] also has a time complexity of $O(\log^2 n)$. Because the number of processes in each stage is not constant the parallel quicksort algorithm would not perform as well as the bitonic merge sort when executed on the Eppl system.

5.4.2 Implementation

The bitonic program shown in Figure 5.5 implements the algorithm described above, except that in each stage of the merge only $\frac{n}{2}$ processes are spawned. Each process must therefore shuffle two array entries sequentially, and then do a compare-exchange. This arrangement appears more efficient than first spawning n shuffles and then $\frac{n}{2}$ compare-exchanges, during which the other $\frac{n}{2}$ processes must

```
int i=0, j=0, k=0, d=0;
int a[n], b[n];

void bitonic(){
   int lk, ld, p, q, s;
   int *from, *to, *t;

   from=a;
   to=b;
   while(1){
      lk=tir(&k,1,n/2);
      p=ips(lk*2,i);
      q=ips((lk*2)+1,i);
      s=(lk/(1<<i))%2;
      if(((s==0)&&(from[p]>from[q]))||((s==1)&&(from[p]<from[q]))){
         to[lk*2]=from[q];
         to[(lk*2)+1]=from[p];}
      else{
        to[lk*2]=from[p];
        to[(lk*2)+1]=from[q];}
      ld=f&a(&d,1);
      if((ld+1)==(n/2)){
         j++;
         if(j>i){
            j=0;
            i++;
            if(i==log2(n))
               done();}
         else{
            t=from;
            from=to;
            to=t;
            f&a(&d,-(n/2));
            f&a(&k,-(n/2));}}}}
```

Figure 5.5: Bitonic merge sort program

idle (assuming there are n processes active). Also the problem of contention for the array entries during the shuffle is solved by having two arrays `a[]` and `b[]`. Initially the integers to be sorted are assumed to be in array `a[]`, and array `b[]` is empty. The global variable `i` is the iteration count, `j` is the stage count, and `k` and `d` are the spawning and termination counters respectively. All these variables are initialised to zero.

On reset each process starts executing function `bitonic()`. The local variables of this function, which are private to each process, are mainly local count values and indices, with the exception of `from`, `to` and `t` which are pointers to the base of an array. At the start of the function `from` is set to point at array `a[]`, and `to` at array `b[]`.

The body of the `while()` loop starts with a call to `tir()`, a blocking test-increment-retest function (Section 2.3.3), which returns an index `lk`. Using this index and the inverse (perfect) shuffle function `ips()`, two indices `p` and `q` are generated. These indices then index into the array `from[]` (which is actually array `a[]` at this point). The correct direction for the subsequent compare-exchange is evaluated in the sign variable `s`, using the current iteration count `i`. In the `if()` statement the entries of array `from[]` indexed by `p` and `q` are compared, and moved into the entries of array `to[]` (which is actually array `b[]` at this point) indexed by `lk*2` and `lk*2+1`. If the sign `s=0` and `from[p]` is greater than `from[q]`, then `from[p]` is moved into `to[lk*2+1]` and `from[q]` into `to[lk*2]`. The same occurs if `s=1` and `from[p]` is less than `from[q]`. Otherwise `from[p]` is moved into `to[lk*2]` and `from[q]` into `to[lk*2+1]`. Having two arrays means that no locking has to be performed, and the pointers to the arrays simply have to be exchanged at the end of the stage. After the compare-exchange the termination count `d` is incremented. If all the processes in that stage have finished, then the stage count `j` is incremented by the last process to finish. If all the stages in the iteration have been executed this process resets the stage count, and increments the iteration count `i`. If all the iterations have been executed the process then

calls done(), indicating the end of the program. If the current iteration has not been completed then the array pointers from and to are swapped, and the spawning and termination counters are reset to zero. All the processes then loop back to the start and try to get another index.

The test of the termination count d effectively forms a barrier, since the spawning count k is only reset when all the processes have passed through the barrier. A re-usable barrier of the sort described in Section 2.3.2 is not necessary here, since processes can only re-enter the loop once all the other processes have exited. This concludes the description of the bitonic merge sort program.

5.5 Moore shortest path

The Moore shortest path algorithm [132] solves the single-source shortest path problem, that is: to find the shortest paths from a single source node to all other nodes in a weighted directed graph. The serial algorithm is shown in Figure 5.6.

The array weight[u][v] contains a weight for all pairs of nodes u,v. If u and v are connected then weight[u][v] is a positive integer greater than 0. For nodes that are not connected weight[u][v]=0, and also weight[u][u]=0. The array distance[] contains the distance from the source node s to all other nodes, where initially distance[s]=0 and the distance to all other nodes is ∞. The nodes to be searched are stored in a FIFO queue. The function enqueue() appends the given node to the tail of the queue, and the function dequeue() removes a node from the head of the queue. The function queue_empty() returns non-zero if the queue is empty, and the function in_queue() returns non-zero if the given node is already in the queue.

The algorithm first enqueues the source node s. Then the while() loop is repeatedly executed until the queue is empty. In the body of the loop a node u is dequeued (initially this will be the source node), and the for() then loops though all the nodes. If node u is connected to node v, then newdistance is set

```
int s, newdistance, u, v;
int distance[n], weight[n][n];
struct fifoq queue;

enqueue(s);
while(~queue_empty()){
   dequeue(u);
   for(u=0;u<n;u++)
      if(weight[u][v]>0){
         newdistance=distance[u]+weight[u][v];
         if(newdistance<distance[v]){
            distance[v]=newdistance;
            if(~in_queue(v))
               enqueue(v);}}}
```

Figure 5.6: Moore shortest path serial algorithm

equal to `distance[u]+weight[u][v]`. If `newdistance` is less than `distance[v]`, then a shorter path to `s` from `v` has been found, and `distance[v]` is updated. If node `v` is not already in the queue then it is appended, and the next node connected to `u` is examined. When there are no further nodes in the queue the array `distance[]` will contain the shortest paths between `s` and all other nodes.

5.5.1 Parallel algorithm

The parallel Moore shortest path algorithm [125] achieves speed-up by allowing the queued nodes to examined concurrently. A number of processes are spawned which wait to dequeue a node from the queue, and then proceed as in the serial algorithm. The parallel algorithm is essentially the same as the serial algorithm, and is therefore not shown as a separate figure. Because the array `distance[]` and the queue are shared data structures some form of mutual exclusion is necessary to ensure correctness, for example semaphores could be used to lock both the array and the queue during update. Also, before the algorithm terminates one must make sure that no processes are active as well as that the queue is empty,

since it is possible that another process is still examining a node even though there are no queued nodes.

The Moore shortest path algorithm differs from the other two algorithms described in this chapter in that it is not so explicitly data parallel. In the matrix multiply and bitonic merge algorithms the number of processes executing is constant, but for the Moore algorithm the amount of parallelism available will depend in part on the size of the graph and its density, and the contention for the shared data structures. A comparison of two parallel implementations of single-source and all-pairs shortest path algorithms on the HEP shows that single-source shortest path performs more efficiently with sparse graphs [128].

5.5.2 Implementation

The implementation of the parallel Moore algorithm is shown in Figure 5.7. To reduce the size of the data describing the graph each node has been limited to having eight outgoing arcs. The weights of each arc are stored in the array `arc[][]`, which for each node has eight entries containing the connected node and the weight of the arc to that node. The parallel FIFO queue structure used to queue nodes is similar to the one described in Section 2.3.4, with a number of changes to make it more efficient. For example, the queue upper and lower bound counters have been replaced by the variable `count`, and since the queue size is made equal to the number of nodes queue overflow is not checked for. Also each queue entry no longer contains insert and delete semaphores, rather a special value (-1) is used to indicate that the entry is empty. Because the maximum number of processes is also equal to the number of nodes, there can be no race conditions between two processes trying to delete the same queue entry.

A number of global structures and variables are defined at the start of the program. The structure queue is assumed to be initialised with the source node `s` as the first entry (in the assembly code of the Moore program the source node is always node 0). The variable `ps` counts the number of active processes, and is

```
struct{
   int count, insert, delete;
   int entries[n];}queuestruct

struct{
   int node, weight;}arcstruct

struct queuestruct queue;
struct arcstruct arc[n][8];
int ps, ws, distance[n], iqsem[n], disem[n];

void moore(){
   int u, v, *qn, i, newdistance;

   while(1){
      tdr(queue->count,1);
      qn=&(queue.entries[f&a(queue->delete,1)%n]);
      while((u=*qn)==(-1));
      *qn=(-1);
      f&a(&iqsem[u],-1);
      f&a(&ps,1);
      for(i=0;i<8;i++){
         v=arc[u][i].node;
         newdistance=distance[u]+arc[u][i].weight;
         p(&disem[v]);
         if(newdistance<distance[v]){
            distance[v]=newdistance;
            v(&disem[v]);
            if(f&a(&iqsem[v],1)==0){
               f&a(&ws,1);
               qn=&(queue.entries[f&a(queue->insert),1)%n]);
               while(*qn!=(-1));
               *qn=v;
               f&a(queue->count,1);}
            else
               f&a(&iqsem[v],-1);}
         else
            v(&disem[v]);}
      if(f&a(&ws,-1)==1)
         done();}}
```

Figure 5.7: Moore program

initialised to 0. The variable ws is used to detect termination, and is initialised to 1 (since the source node is already queued at the start of the program). The array iqsem[] indicates whether a node is in the queue (playing the part of the in_queue() function). If node u has been queued then iqsem[u]=1, or 0 otherwise. The array distance[] contains the distances from the source node, and is initialised so that distance[s]=0 and all other entries have a very large value. The array disem[] contains the semaphores used to lock entries in the distance[] array. If disem[u]=1 then distance[u] is locked, and the process must wait to access the array.

On reset the processes start executing function moore(). Each process has a number of private variables including u and v which are node numbers, and qn which is a pointer to a queue entry. The process first calls tdr(), which decrements the queue count. If the queue is empty (count=0), then tdr() will block. The queue delete pointer is then incremented, and qn set to point at the head queue entry. The process then checks that the queue entry has a valid node in it, reads it into u, and marks the queue entry empty. Since node u has now been dequeued iqsem[u] is decremented, and the process count ps is incremented. The process then loops though each of the 8 outgoing arcs from node u. For each connected node v the process calculates the newdistance, and then locks distance[v] using the p() semaphore function (Section 2.3.1). Assuming that the new path length is shorter than the current path length, the process updates distance[v] and unlocks the entry using the v() semaphore function. The node v must then be enqueued, so iqsem[v] is first checked to make sure it is not already in the queue. If not, the termination count ws is incremented, and qn is set to point to the tail queue entry. The process then waits until the queue entry is empty. Node v is then written to the queue entry, and count incremented. If after all the arcs have been examined ws=1, then the program is finished and done() is called.

The process count ps is not used directly in the algorithm, but allows the

number of active processes to be traced by examining the value of this variable during execution. The termination count `ws` is incremented when a new node is enqueued, and decremented when a process finishes examining a node. Its value is therefore similar to `count`, but it is not possible to use `count` directly to detect termination because of race conditions which can occur. This concludes the description of the Moore shortest path program.

5.6 Comments

For the simulation experiments described in the next chapter a number of versions of each benchmark program were assembled with different data sizes. The data for the programs is randomly generated. Each assembled program is referred to by a combination of mnemonic and number: for example mm256, bit4096, and moo32. The mnemonic refers to the program type, and the number indicates the parallelism of the program. For matrix multiply and bitonic merge sort the program parallelism is directly related to the data size: for example the mm256 program has a result array of 256 entries and a parallelism of 256, the bit4096 program has a sequence of 8192 entries to sort and a parallelism of 4096. For the Moore shortest path program the number refers to the number of nodes in the graph: for instance the moo32 program has a graph with 32 nodes. This defines the maximum parallelism possible, but the exact parallelism is data dependent.

Chapter 6

Performance evaluation

This chapter describes the simulation experiments carried out using the Eppl simulator. The majority of the experiments involve observing the effect of changing the Eppl system parameters on the performance of the Eppl design, as measured by the execution time of the programs and other execution statistics collected by the simulator. The first section investigates the speed-up of the programs executing on increasing system sizes. The following sections investigate the effect on performance of increasing the switch queue and wait-buffer size, combining level, processor interleaving level, and the module clock factor. Most of the simulation results are shown as graphs of program execution time (in clock cycles), and the other execution statistics are shown as tables in Appendix B. The final section presents dynamic execution profiles for each of the benchmark programs.

For most of the simulations the data size of the program being executed is chosen to be a saturation load. This means the program parallelism is the same as the system parallelism, and all processes have some work to do. For example with a 16 processor system and an interleaving level of 16, giving a total of 256 processes, a 256 entry result array is used for the matrix multiply program (mm256). In all simulations the given switch queue size applies to both request and acknowledge queues, and the given wait-buffer size applies to each wait-buffer.

6.1 Program speed-up

The speed-up simulation experiments investigate how the execution time of each benchmark program (with constant data size) changes as the system size is increased. The speed-up of a program is dependent on both the scalability of the system on which the program is being executed, and the scalability of the program itself. The scalability of the Eppl design is predominantly dependent on the interconnection network: the memory access latency grows logarithmically with network size and is further increased by contention. The Eppl processor architecture tries to overcome the increased latency by instruction interleaving, and the switch architecture tries to reduce the contention by combining. The scalability of the program depends on how the program parallelism increases with the problem size, and the amount of process control overhead.

6.1.1 Simulation details

For the speed-up simulations the mm4096, bit4096 and moo4096 programs were executed on increasing system sizes of 1 to 256 processors. The default values of the system parameters were: an interleaving level of 16, 2-way combining, a queue size of 4 packets, a wait-buffer size of 16 packets, and equal module clock factors.

The graphs of execution time, speed-up, efficiency and memory latency for the programs are shown in Figures 6.1 to 6.4. The speed-up and efficiency are additionally shown numerically in Table 6.1. The detailed simulation results are shown in Tables B.2 to B.4 of Appendix B.

6.1.2 Description of results

For all three programs the execution time (Figure 6.1) is reduced as system size increases. Bitonic has the longest execution time, and matrix multiply and Moore have (coincidentally) similar execution times. The speed-up (Figure 6.2) is calcu-

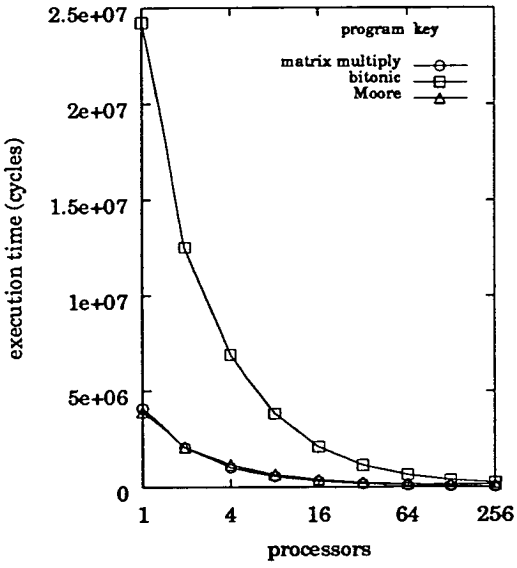


Figure 6.1: Execution time for all three programs with increasing system size.

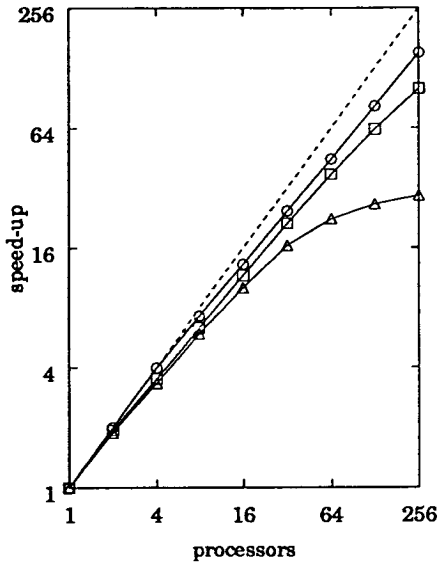


Figure 6.2: Speed-up for all three programs with increasing system size.

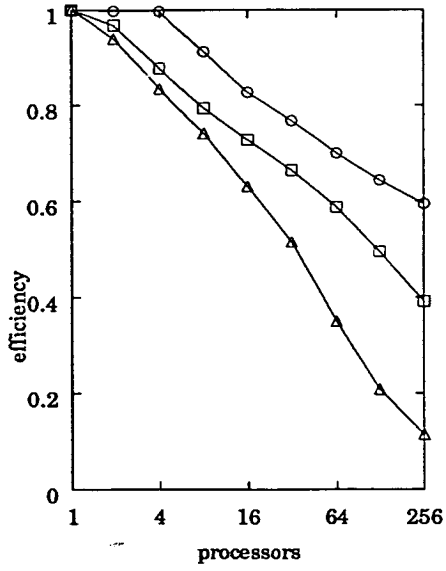


Figure 6.3: Efficiency for all three programs with increasing system size.

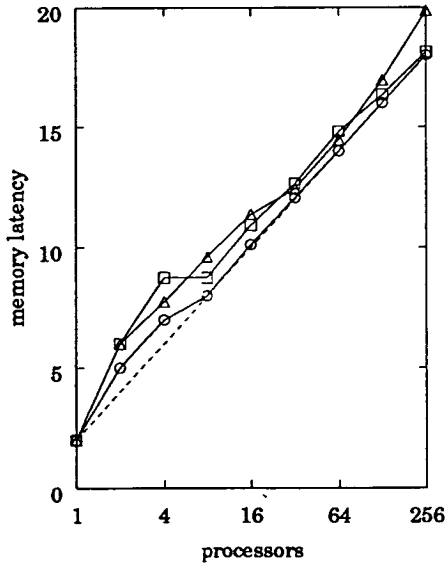


Figure 6.4: Memory latency for all three programs with increasing system size.

program	matrix multiply		bitonic		Moore	
processors	speed-up	efficiency	speed-up	efficiency	speed-up	efficiency
1	1.00	1.00	1.00	1.00	1.00	1.00
2	2.00	1.00	1.94	0.97	1.88	0.94
4	3.99	1.00	3.51	0.88	3.34	0.84
8	7.30	0.91	6.35	0.79	5.93	0.74
16	13.23	0.83	11.64	0.73	10.08	0.63
32	24.55	0.77	21.25	0.66	16.49	0.52
64	44.80	0.70	37.58	0.59	22.39	0.35
128	82.35	0.64	63.40	0.50	26.60	0.21
256	152.23	0.59	100.43	0.39	29.35	0.11

Table 6.1: Speed-up and efficiency of matrix multiply (mm4096), bitonic (bit4096) and Moore (moo4096) programs with increasing system size.

lated relative to the single processor execution time of each program, the dashed line on the graph showing the ideal linear speed-up. Matrix multiply has the best speed-up, and Moore the worst — already reaching the knee of the curve at 32 processors. How much the programs fall below the ideal speed-up is shown more clearly by the graph of efficiency (Figure 6.3). The efficiency is derived by dividing the actual speed-up by the expected linear speed-up, and the resulting fraction shows how well the system is being utilised with each increase in system size. The numerical results in Table 6.1 show that the efficiency for matrix multiply is 59% compared to 11% for Moore with 256 processors. The memory latency (Figure 6.4) for matrix multiply is almost at the minimum possible for each system size (shown by the dashed line), and for the other two programs it is close to the minimum.

The detailed results (Tables B.2 to B.4) show that the processor utilisation decreases as the system size increases for all programs, mainly due to the increasing memory latency. With 64 or more processors the memory latency is higher than the interleaving level. Also, the number of instructions executed to complete each program should ideally be almost the same as the system size is increased. This is the case for matrix multiply, but for bitonic and Moore the number of instructions executed increases monotonically with system size. This indicates

that the processes are spending more time busy-waiting.

6.1.3 Summary

The speed-up experiments show the matrix multiply program is executing near the design's limit, since the execution efficiency of the program is similar to the processor efficiency. The other programs have worse performance caused by contention and lack of program parallelism. Whether increasing the interleaving level can increase processor utilisation, and improve execution time, is investigated in Section 6.4.

6.2 Queue and wait-buffer size

The queue and wait-buffer size simulation experiments investigate the effect of varying the switch queue and wait-buffer size on system performance. Previous research on non-combining networks [56, 59] and the network simulations in Section 3.2.6 show that the network throughput improves as queue size is increased, but that the network latency also rises. When a hot-spot is present increasing the queue size is less effective in improving throughput. Previous research on combining networks [73] shows that the number of combinations may be increased by larger request queues. Also the effectiveness of combining will be reduced if possible combinations cannot occur because the wait-buffers are full.

6.2.1 Simulation details

For the queue and wait-buffer simulations the saturation load programs were executed on system sizes of 2 to 128 processors (256 for matrix multiply), with varying queue size of 2 to 32 packets and varying wait-buffer size of 2 to 32 packets. The default values of the other system parameters were: an interleaving level of 16, 2-way combining, and equal module clock factors.

The graphs of execution time for the programs are shown in Figures 6.5 to 6.19,

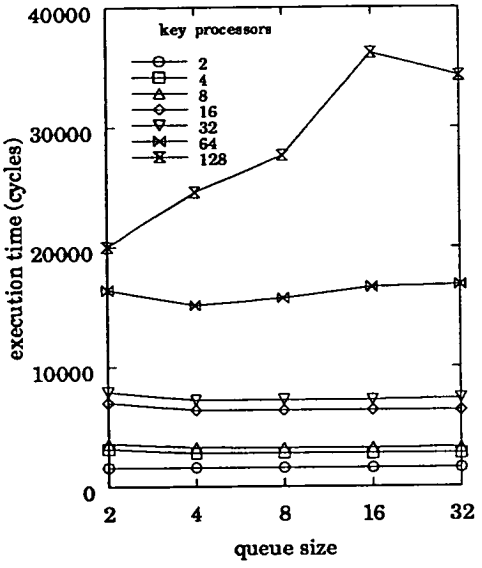


Figure 6.5: Execution time of matrix multiply with varying queue size and wait-buffer size 2.

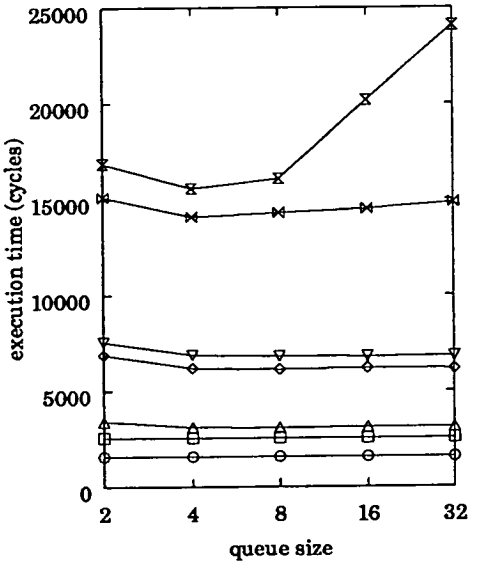


Figure 6.6: Execution time of matrix multiply with varying queue size and wait-buffer size 4.

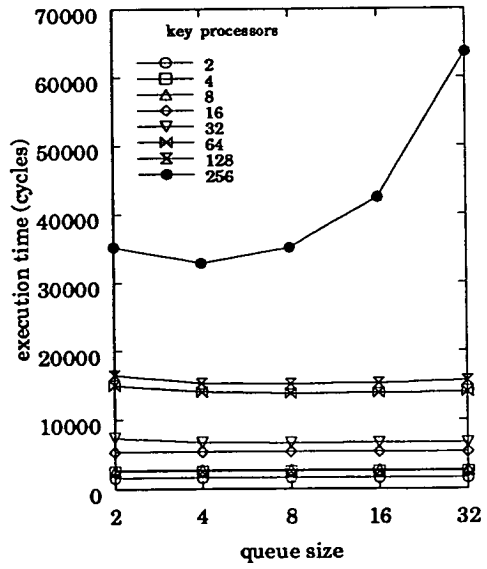


Figure 6.7: Execution time of matrix multiply with varying queue size and wait-buffer size 8.

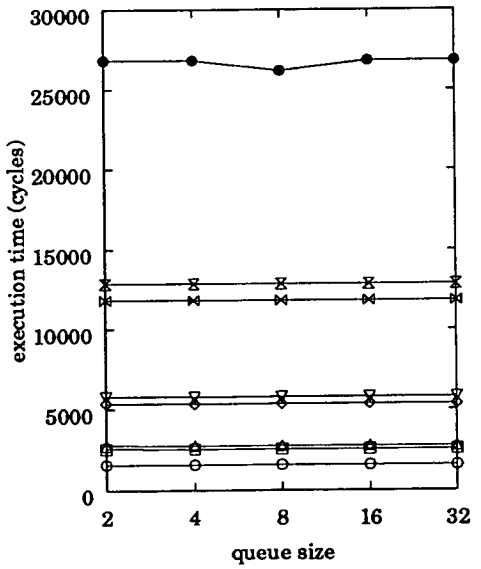


Figure 6.8: Execution time of matrix multiply with varying queue size and wait-buffer size 16.

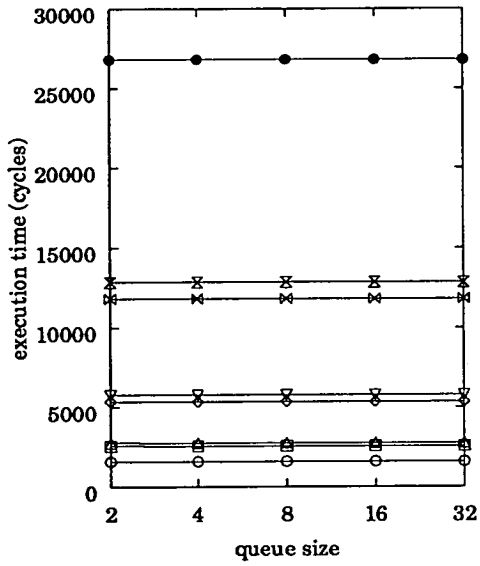


Figure 6.9: Execution time of matrix multiply with varying queue size and wait-buffer size 32.

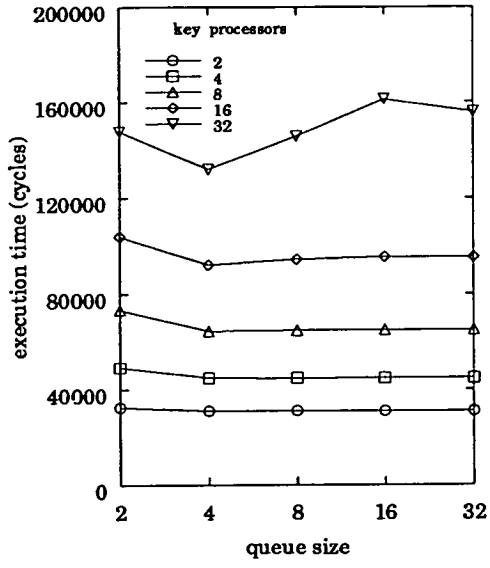


Figure 6.10: Execution time of bitonic with varying queue size and wait-buffer size 2.

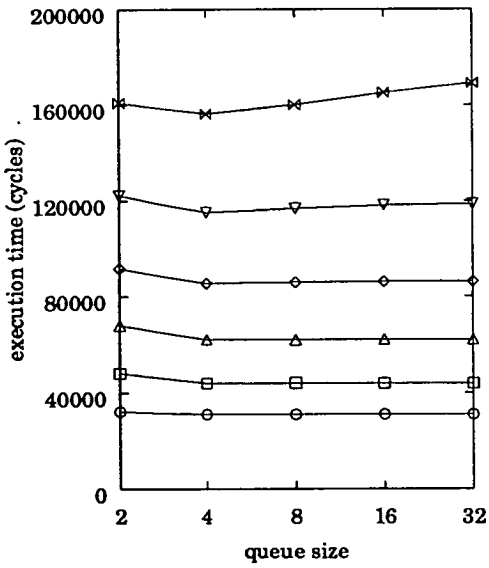


Figure 6.11: Execution time of bitonic with varying queue size and wait-buffer size 4.

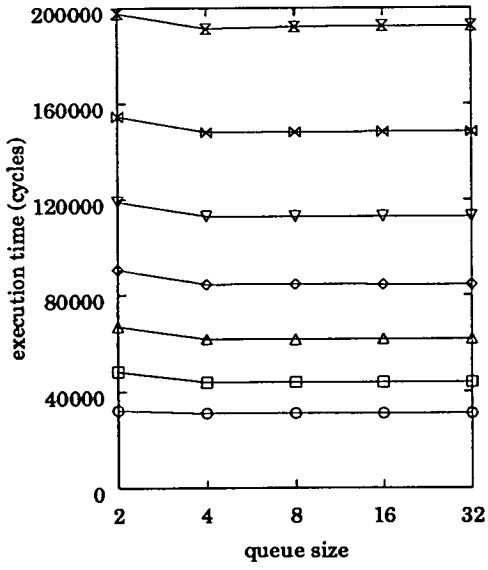


Figure 6.12: Execution time of bitonic with varying queue size and wait-buffer size 8.

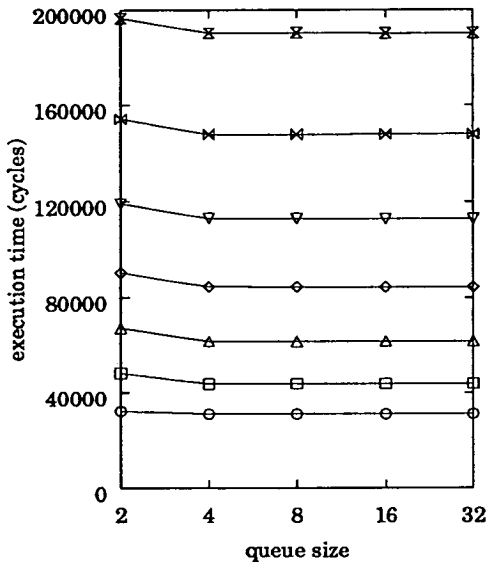


Figure 6.13: Execution time of bitonic with varying queue size and wait-buffer size 16.

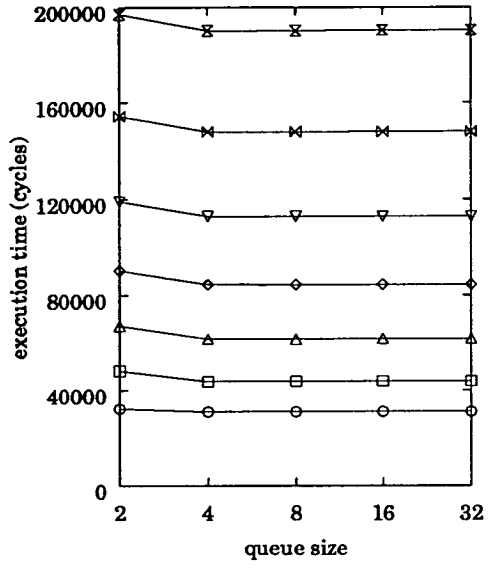


Figure 6.14: Execution time of bitonic with varying queue size and wait-buffer size 32.

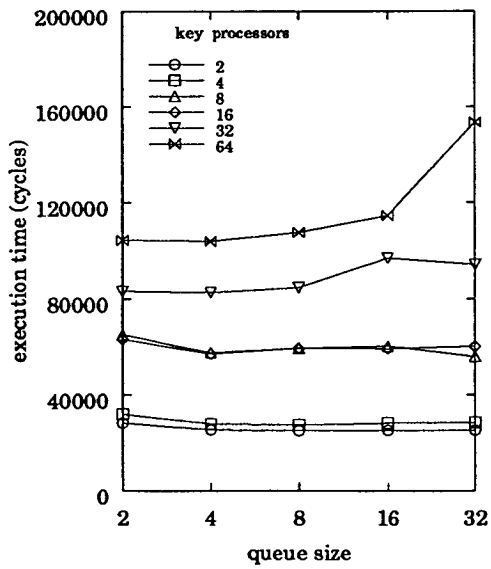


Figure 6.15: Execution time of Moore with varying queue size and wait-buffer size 2.

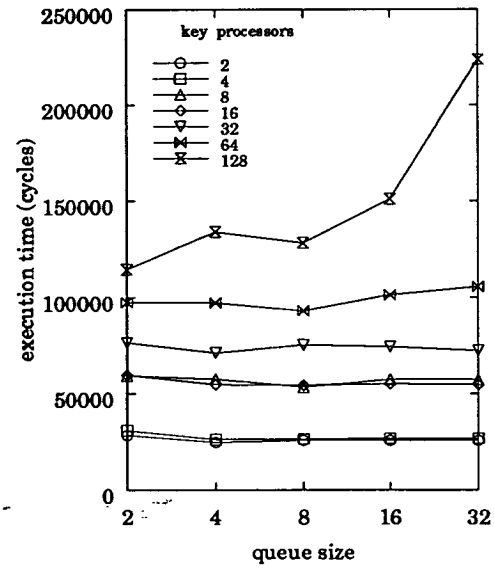


Figure 6.16: Execution time of Moore with varying queue size and wait-buffer size 4.

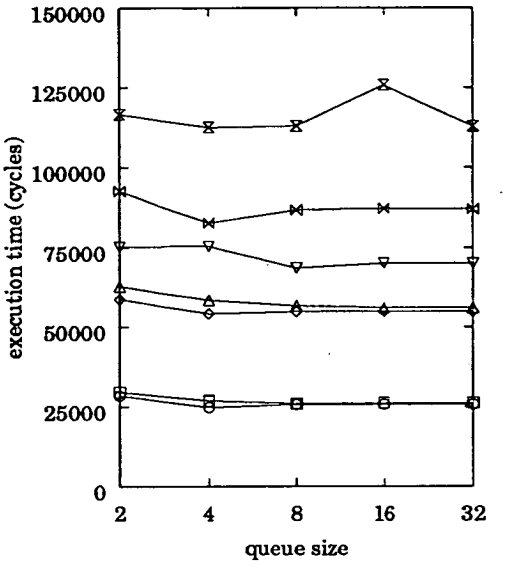


Figure 6.17: Execution time of Moore with varying queue size and wait-buffer size 8.

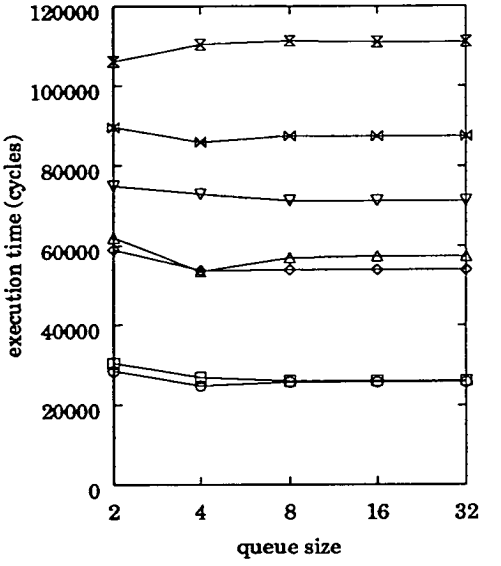


Figure 6.18: Execution time of Moore with varying queue size and wait-buffer size 16.

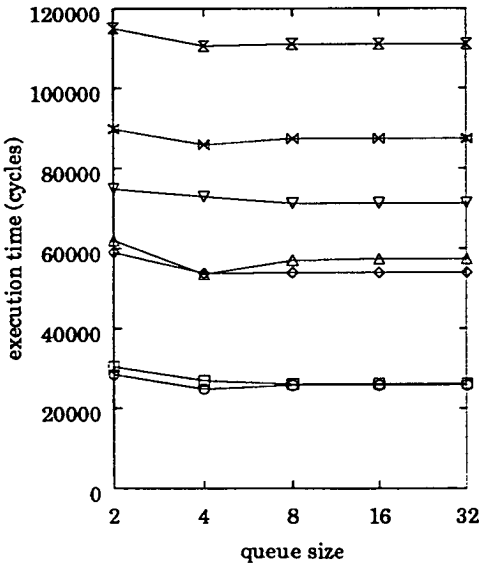


Figure 6.19: Execution time of Moore with varying queue size and wait-buffer size 32.

and are arranged so that each graph shows the execution time of one program with a constant wait-buffer size and varying queue size. The detailed simulation results are shown in Tables B.5 to B.25 of Appendix B.

6.2.2 Description of results

The matrix multiply program with a wait-buffer size of less than 16 packets (Figures 6.5 to 6.7), shows an initial drop in execution time between a queue size of 2 and 4 packets. On systems larger than 32 processors, there is an increase in execution time as the queue size increases above 4 packets. With a wait-buffer size of 16 or more (Figures 6.8 to 6.9), increasing the queue size has little effect on execution time. Increasing the wait-buffer size decreases the execution time for each system size, up to a point above which there is no further improvement. For matrix multiply on the 256 processor system this maximum wait-buffer size is 16 packets.

The bitonic program (Figures 6.10 to 6.14), shows a similar pattern to matrix multiply. The initial drop in execution time between queue size 2 and 4 occurs for all wait-buffer sizes. Above a wait-buffer size of 8 there is little change in execution time, though a wait-buffer size of 32 is necessary for there to be no wait-buffer full events with 128 processors (Table B.18).

The results for the Moore program (Figures 6.15 to 6.19), are less consistent than the other two programs. In some cases a queue size of 8 gives the minimum execution time, and in others a queue size of 4. Again, increasing queue size with small wait-buffers increases execution time. Increasing the wait-buffer size above 16 results in little change in execution time.

The detailed results for all the programs (Tables B.5 to B.25), show that as queue size is increased the number of request and acknowledge queue conflicts decreases, leading to a decrease in memory latency. The number of combinations also increases slightly with queue size. With small wait-buffers the number of queue conflicts is significantly higher, resulting in increased memory latency. If

the wait-buffer size is sufficiently large then there are fewer or no queue conflicts.

6.2.3 Summary

There is a general drop in execution time as queue size increases from 2 to 4, and then little further change assuming the wait-buffers are sufficiently large. No increase in latency with queue size, comparable to that in the isolated network simulations in Section 3.2.6, occurs. This is because the Eppl network is part of a closed system, and so the load on the network is related to its throughput (as there are only a finite number of possible memory requests which can be in progress at any one time). The maximum wait-buffer size necessary for no wait-buffer full events to occur increases with system size, and also varies with the program. Overall a queue size of 4 and a wait-buffer size of 16 is sufficient to give close to the best performance for all the simulated system sizes and programs.

6.3 Combining level

The combining level experiments investigate the effect of increasing the combining level (including no combining) on system performance. Previous research [58, 59] and the network simulations in Section 3.2.6 show that hot-spots degrade the performance of non-combining multistage networks. This is also the case in the Eppl system when combining is turned off: the execution times of the programs become so large with no combining, that only simulations with small numbers of processors could be completed within the time constraints on access to the ECS. Previous research [58] shows 2-way combining to be sufficient to improve performance for networks with up to 64 inputs, but for larger networks and higher loads 3-way combining must be used [73, 74]. From the results of the previous two sections it appears that the Eppl network does perform acceptably with 2-way combining, as indicated by the average memory latency being close to the minimum. Therefore the contention for any hot-spots must be sufficiently low

or transient for 2-way combining to remain effective. Increasing the combining level could improve performance though, by allowing more combinations to occur. With higher combining levels the acknowledge queues must be large enough to accept the multiple decombinations, and the maximum wait-buffer size necessary may be reduced even though the number of combinations increases [74].

6.3.1 Simulation details

For the combining level simulations the saturation load programs were executed on system sizes of 2 to 256 processors, with no combining and 2-way to 4-way combining. The default values of the other system parameters were: an interleaving level of 16, a queue size of 4 packets, a wait-buffer size of 64 packets and equal module clock factors.

The graphs of execution time for the programs are shown in Figures 6.20 to 6.22. Additionally the number of combinations occurring for the 3-way and 4-way combining levels are shown in Table 6.2. The detailed simulation results are shown in Tables B.26 to B.28 in Appendix B.

6.3.2 Description of results

For the matrix multiply program (Figure 6.20), results with no combining were obtained for up to 32 processors (the last point was omitted from the graph and falls at 52434 cycles). It can be seen that from 8 processors the execution time becomes much higher with no combining compared to 2-way combining. As the combining level is increased above 2-way there no change in execution time. Most of the possible combinations are already occurring with 2-way combining, and there are only very few 3 or 4-way combinations (Table 6.2).

For the bitonic program (Figure 6.21), results with no combining were only obtained for up to 8 processors (above this the simulation did not finish even after 96 hours running time). Again the performance with no combining is im-

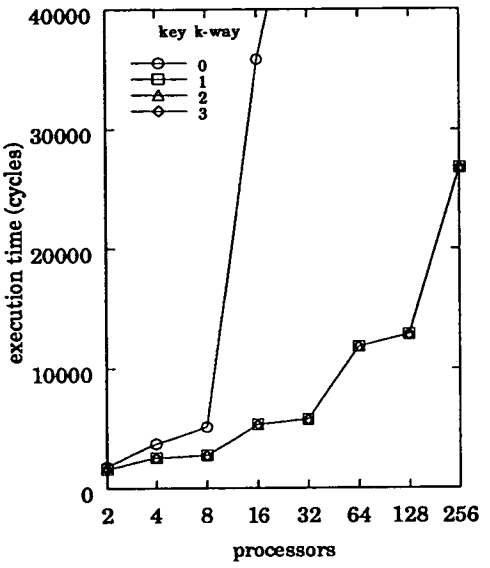


Figure 6.20: Execution time for matrix multiply with increasing combining level.

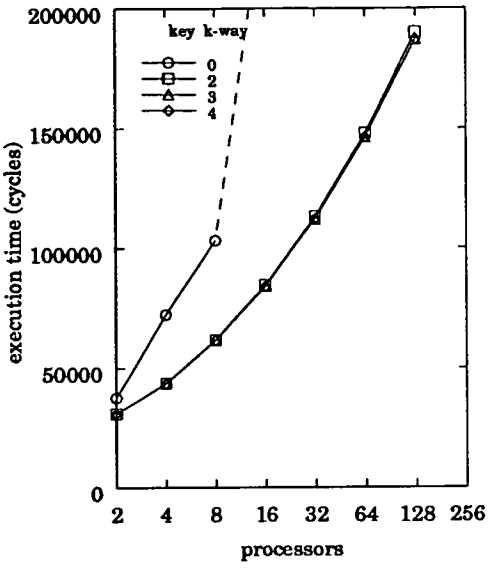


Figure 6.21: Execution time for bitonic with increasing combining level.

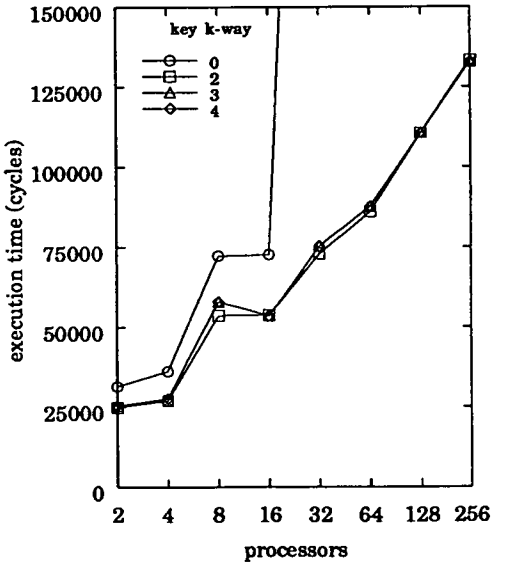


Figure 6.22: Execution time for Moore with increasing combining level.

program		matrix multiply		bitonic		Moore	
processors	k	3-way	4-way	3-way	4-way	3-way	4-way
2	3	0	0	1251	0	1227	0
2	4	0	0	1359	117	1167	177
4	3	3	0	4003	0	2986	0
4	4	3	1	3993	325	2869	294
8	3	1	0	9867	0	9419	0
8	4	1	0	9598	747	9312	848
16	3	0	0	23452	0	15884	0
16	4	0	0	22847	1949	15631	1623
32	3	3	0	54808	0	48713	0
32	4	2	1	52732	5108	41621	4968
64	3	0	0	123055	0	97673	0
64	4	0	0	116873	13135	96503	11956
128	3	0	0	271698	0	230059	0
128	4	0	0	255587	31839	220639	27824
256	3	0	0	—	—	560267	0
256	4	0	0	—	—	567841	77392

Table 6.2: 3 and 4-way combinations for matrix multiply, bitonic and Moore programs.

mediately worse than with 2-way combining. As the combining level is increased the execution time does decrease slightly for the larger system sizes. The detailed results (Table B.27) show that the number of acknowledge queue conflicts is reduced, even though the number of multiple decombinations increases with the combining level. The total number of combinations increases, although in some cases there is a drop between 3-way and 4-way combining.

For the Moore program (Figure 6.22), results with no combining were obtained for up to 32 processors (the final point was omitted from the graph and falls at 5038280 cycles). Increasing the combining level above 2-way actually reduces performance in most cases. The detailed results (Table B.28) show that the memory latency is reduced as combining level increases, and that the number of combinations increases.

Further simulations were carried out in which each only one decombine can occur every cycle (the detailed results are not presented here). This means that returning packets have to stay in the input latches of the acknowledge ports until

all matching packets in the wait-buffers are decombined. Increasing the combining level then results in a large decrease in performance, due to the blockage of the inputs of the return network by acknowledge packets waiting to decombine. This leads to request queue conflicts occurring in the switches connected to the hot memories. The ability to decombine multiple packets is necessary if greater than 2-way combining is to be used.

6.3.3 Summary

With no combining the performance of the network is drastically reduced even with only 8 processors, as a result of the increased number of request queue conflicts and memory accesses. Increasing the combining level above 2-way may improve the performance slightly depending on the program being executed. Overall, there is not enough improvement to warrant the extra implementation cost of increasing the combining level above 2-way.

6.4 Interleaving level

The interleaving level simulation experiments investigate the effect of increasing the interleaving level (number of processes per processor) on system performance. The single processor simulations in Section 3.1.7 show that, for a given memory latency and instruction mix, there is an optimal interleaving level which results in the maximum processor utilisation and throughput per process. Increasing the interleaving level above the optimal number reduces the throughput per process as the processes have to contend for the processor resources. In the Eppl system the size of the program being executed must be increased if the interleaving level is increased, otherwise the extra processes will not have any useful work to do and will simply busy-wait. The optimal program parallelism will also be determined by the system parallelism, and ideally the best performance should result when the program and system parallelism are equal.

6.4.1 Simulation details

For the interleaving level simulations the programs were executed on system sizes of 1 to 64 processors, with interleaving levels of 16 to 64 processes per processor (incremented in steps of 8 processes). The program size used was large enough to saturate the highest interleaving level, and was used for all the simulations with that system size. For instance with the 64 processor system the mm4096 program was used, which saturates all 4096 processes when the interleaving level is 64. Larger system sizes than 64 processors were not simulated because 4096 was largest program size assembled, and also due to time constraints. The default values of the other system parameters were: 2-way combining, a queue size of 4 packets, a wait-buffer size of 64 packets and equal module clock factors.

The execution times of the programs with increasing interleaving level are shown in Figures 6.23 to 6.25. The detailed simulation results are shown in Tables B.29 to B.34 of Appendix B.

6.4.2 Description of results

For the matrix multiply program (Figure 6.23), an interleaving level of 32 processes per processor results in the minimum execution time for all system sizes. With less than 16 processors the execution times with interleaving levels of 16, 32 and 64 are similar, but with 16 or more processors an interleaving level of 16 performs less well than an interleaving level of 32. In all cases an interleaving level of 56 results in the worst performance.

The results from the bitonic program (Figure 6.24) are similar to matrix multiply. For up to 16 processors the minimum execution time is with an interleaving level of 16, changing to 32 as the system size increases above 16 processors. An interleaving level of 64 performs worse than an interleaving level of 16, and the maximum execution time again occurs with an interleaving level of 56.

For the Moore program (Figure 6.25), an interleaving level of 16 initially

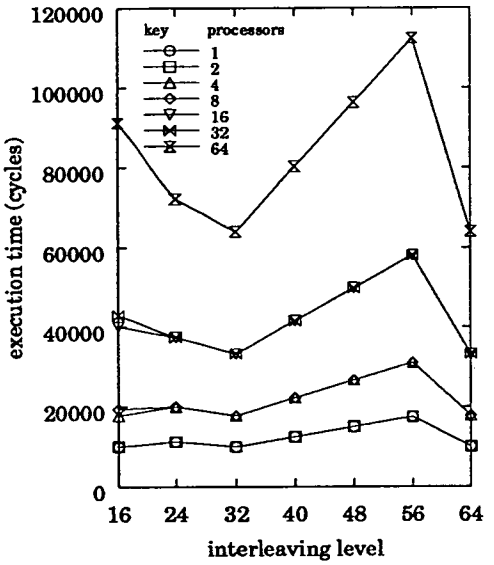


Figure 6.23: Execution time for matrix multiply with increasing interleaving level.

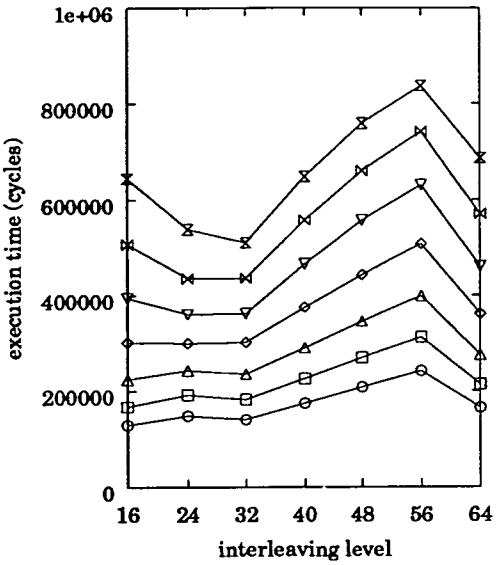


Figure 6.24: Execution time for bitonic with increasing interleaving level.

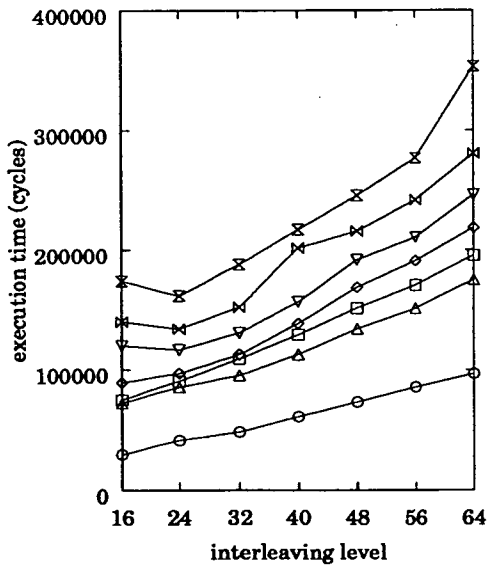


Figure 6.25: Execution time for Moore with increasing interleaving level.

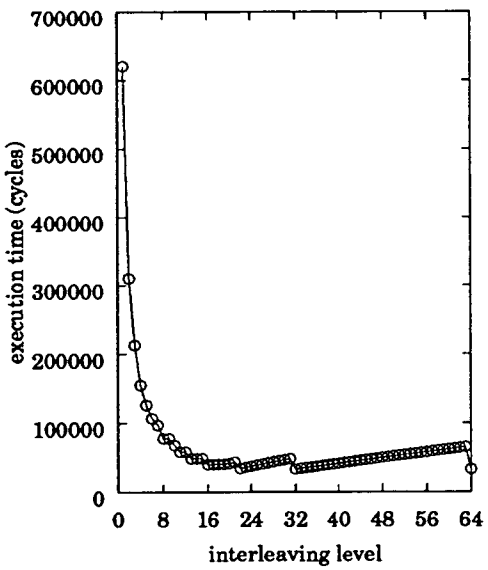


Figure 6.26: Execution time for matrix multiply on 16 processors with increasing interleaving level.

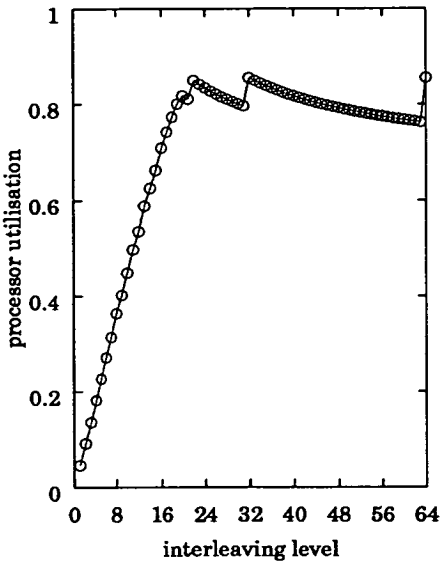


Figure 6.27: Processor utilisation for matrix multiply on 16 processors with increasing interleaving level.

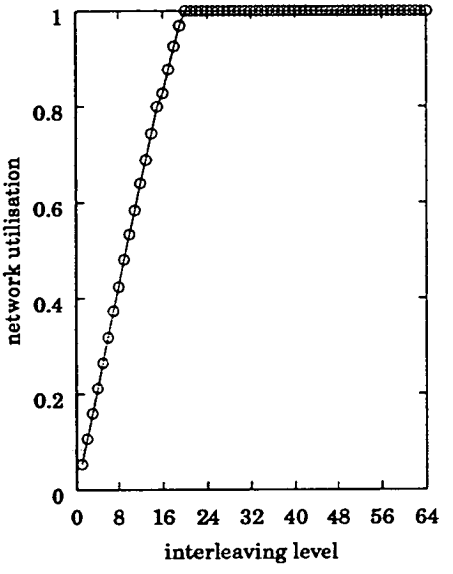


Figure 6.28: Network utilisation for matrix multiply on 16 processors with increasing interleaving level.

gives the best performance, changing to 24 with 32 or more processors. There is a monotonic increase in execution time as further processes are added. The detailed results (Tables B.33 to B.34) show an almost monotonic increase in memory latency with interleaving level, and a monotonic increase in request and acknowledge queue conflicts. The number of f&a combinations also increases with interleaving level.

A further simulation was carried out in which the matrix multiply mm1024 program was executed on a 16 processor system, with the interleaving level being varied from 1 to 64 processes in steps of one. The default values of the other system parameters were as above. The execution time, processor and network utilisation are shown in Figures 6.26 to 6.28.

The minimum execution time (Figure 6.26) occurs at 32 processes, although the curve has almost leveled off by 16 processes after which there are three "ramps", with drops in execution time at 22, 32 and 64 processes. The graphs of processor (Figure 6.27) and network utilisation (Figure 6.28) show a similar

behaviour to the single processor simulations in Section 3.1.7. The processor utilisation reaches a maximum at 22 processes, after which there are similar ramps, and the network utilisation reaches the maximum at 20 processes.

6.4.3 Summary

The best execution times for matrix multiply and bitonic occur with interleaving levels of 16, 32 or 64 depending on system size, since the program parallelism is a power of 2 for both these programs. The lowest execution time for these two programs occurs with an interleaving level of 32, even though each process has to do twice as much work as with an interleaving level of 64. Increasing the interleaving level above 32 results in increased contention, which is shown by the increase in memory latency for bitonic, and cannot be compensated for by adding more processes. For the Moore program it is clear that the program parallelism is not large enough to support an increased number of processes. With large interleaving levels the majority of processes have no work to do and busy-wait, as shown by the increased number of f&a combinations.

6.5 Relative module speed

The relative module speed simulation experiments investigate how the speed of the network and memory (or memory subsystem) affects the system performance. The speed of the memory subsystem is varied from 4 times slower to 4 times faster than the processors, using the module clock factor system parameter (Section 4.5.3). Increasing the speed of the memory subsystem above that of the processors should reduce the memory latency and decrease the program execution time.

6.5.1 Simulation details

For the relative module speed simulations the saturation load programs were executed on system sizes from 2 to 256 processors, with varying module clock factors. To make the processors execute slower than the memory subsystem the processor module clock factor is increased, and *vice versa* for the memory subsystem. The default values of the other system parameters were: an interleaving level 16, 2-way combining, a queue size of 4 packets and a wait-buffer size of 64 packets.

The execution time and resulting speed-up for the programs are shown in Figures 6.29 to 6.34. The memory subsystem speed ratio shows the relative speed of the memory subsystem to the processors. The detailed simulation results are shown in Tables B.35 to B.40 of Appendix B.

6.5.2 Description of results

For all programs and system sizes (Figures 6.29 to 6.31), the execution time decreases as the speed of the memory subsystem is increased from 4 times slower than the processors to equal, but above this significant further decreases in execution time are seen only with 32 or more processors. The speed-up for each system size (Figures 6.32 to 6.34), is calculated relative to the execution time with equal module speed. When the speed ratio is below 1 the speed-up is similar to the speed ratio, but increases with system size for all programs. Slowing down the memory subsystem therefore slows down the larger systems comparatively less (an exception appears to be Moore on 256 processors). When the speed ratio is above 1 the speed-up is less than the speed ratio, but increases with system size. Increasing the memory subsystem speed therefore increases the overall speed of the larger systems comparatively more.

The detailed results (Tables B.35 to B.40) show that the processor utilisation increases with memory subsystem speed, but reaches the maximum utilisation at some point depending on the system size. The network utilisation decreases

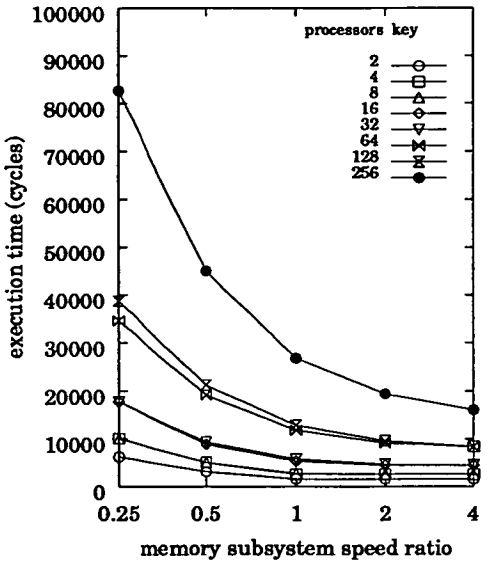


Figure 6.29: Execution time with increasing memory subsystem speed for matrix multiply.

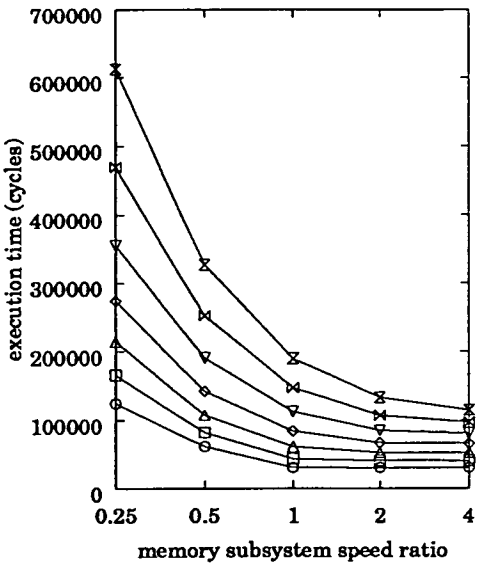


Figure 6.30: Execution time with increasing memory subsystem speed for bitonic.

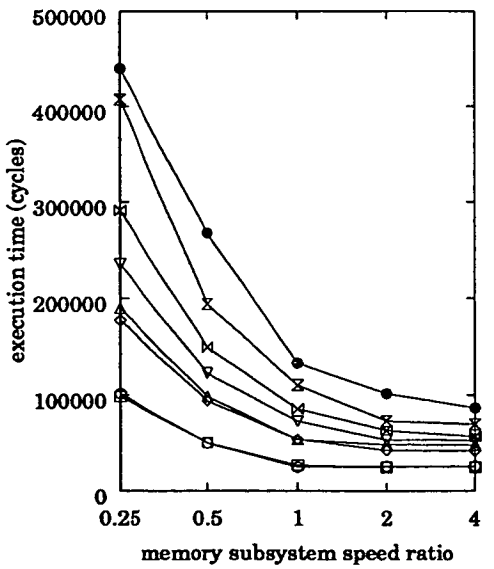


Figure 6.31: Execution time with increasing memory subsystem speed for Moore.

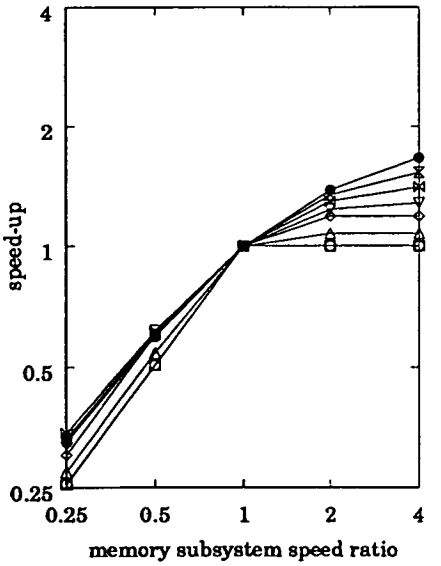


Figure 6.32: Speed-up with increasing memory subsystem speed for matrix multiply.

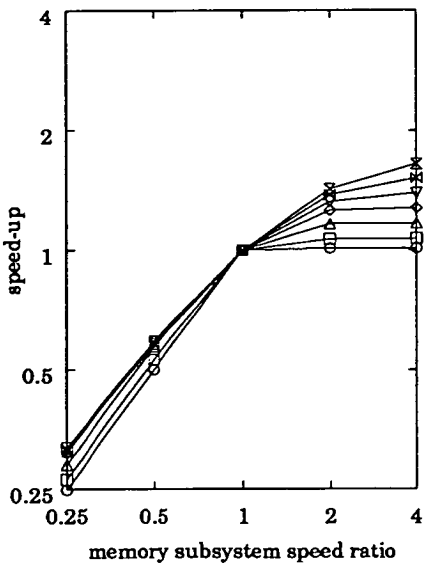


Figure 6.33: Speed-up with increasing memory subsystem speed for bitonic.

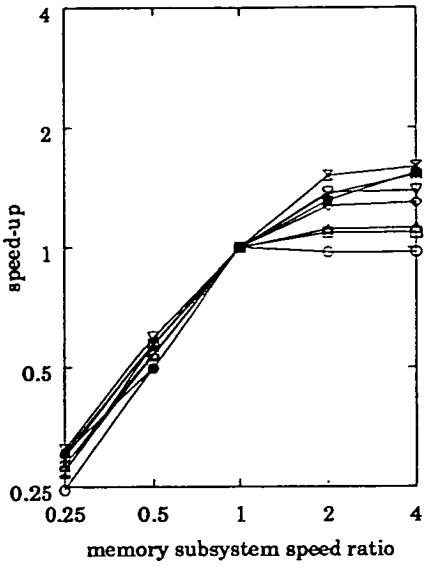


Figure 6.34: Speed-up with increasing memory subsystem speed for Moore.

steadily, since the load on the network is reduced, converging to 25% utilisation with a speed ratio of 4 for all system sizes. Comparing the increases in memory latency with the speed ratio shows that although the changes in memory latency are lower than the speed ratio, they are greater than the values for speed-up. Therefore the network performance does increase with speed ratio, above what the execution times and speed-up suggest. As the speed ratio is increased the number of combinations decreases, since there is less congestion and fewer chances to combine, resulting in an increase in the number of memory accesses.

6.5.3 Summary

The results show that increasing the memory subsystem speed above the processor speed does not necessarily result in an equal increase in overall system performance. With smaller systems the memory latency quickly drops to the minimum as the speed ratio is increased, and the processor throughput becomes the performance bottleneck, as shown by the processor utilisation. Larger sys-

tems show better response to increasing the memory subsystem speed, but the return in performance is not sufficient to justify the cost of increasing the memory subsystem speed. Reducing the memory subsystem speed below the processor speed results in an equal performance drop.

6.6 Program execution profiles

The execution profile simulation experiments investigate how the system performance varies during the execution of the programs. In the above simulations the execution statistics were displayed at the end of each simulation. These statistics represent either the total number of occurrences or the averaged value of some variable during the simulation, and provide little information about the dynamic behaviour of the program. By displaying the execution statistics at regular intervals during the program an execution profile (or trace) can be created. In addition to the usual execution statistics some of the program variables are displayed as well, in particular the process scheduling and termination counters, which allow the number of active processes to be derived.

6.6.1 Simulation details

For the execution profile simulations the mm1024, bit1024 and moo1024 programs were executed on a 64 processor system, and the execution statistics were displayed every 10 clock cycles during the simulation. The wait-buffer size used was 8 packets for the mm1024 and bit1024 programs, and 16 for the moo1024 program (so that some queue conflicts would occur). The default values of the other system parameters were: an interleaving level of 16, 2-way combining, a queue size of 4 packets, and equal module clock factors.

The execution profiles for the programs are shown in Figures 6.35 to 6.63. The eventual totals of the execution statistics can be found in the relevant parts of the queue and wait-buffer simulation result Tables in Appendix B.

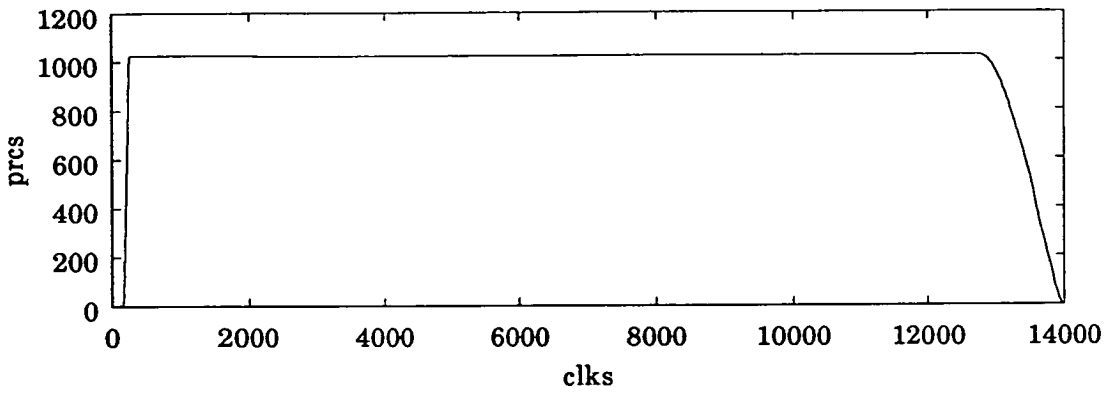


Figure 6.35: Execution profile for matrix multiply showing the number of active processes.

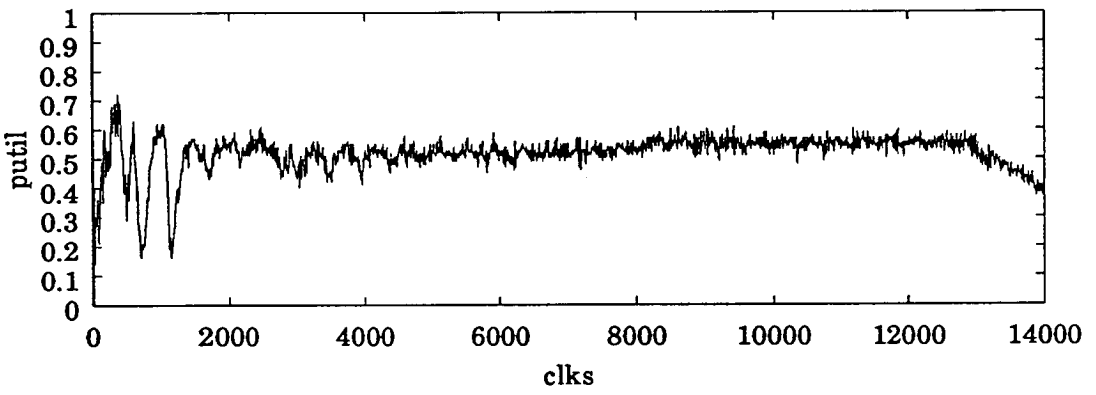


Figure 6.36: Execution profile for matrix multiply showing the processor utilisation.

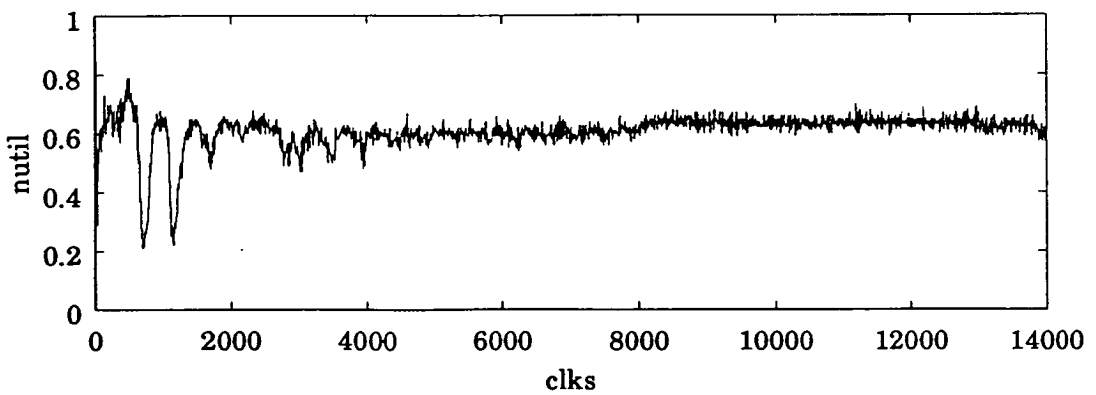


Figure 6.37: Execution profile for matrix multiply showing the network utilisation.

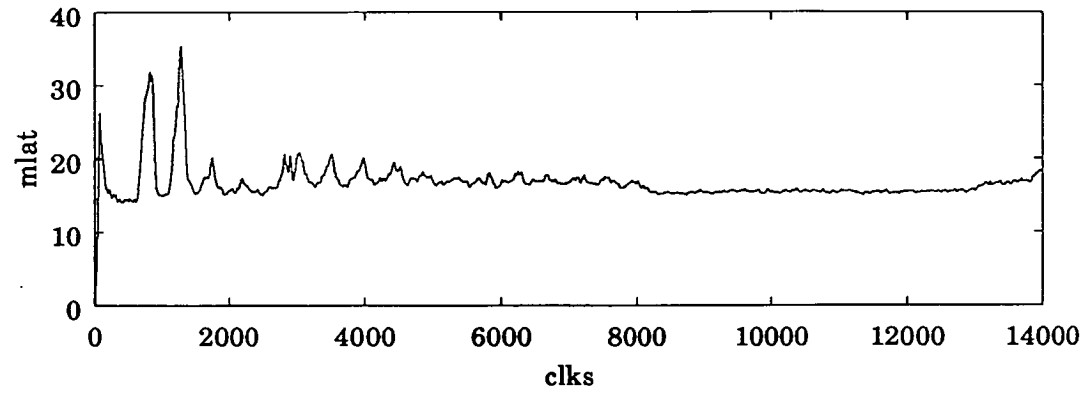


Figure 6.38: Execution profile for matrix multiply showing the average memory latency.

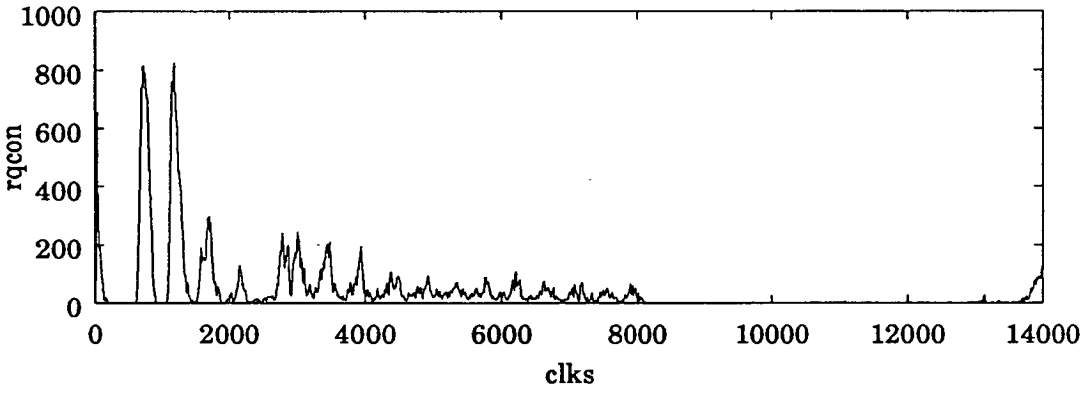


Figure 6.39: Execution profile for matrix multiply showing the number of request queue conflicts.

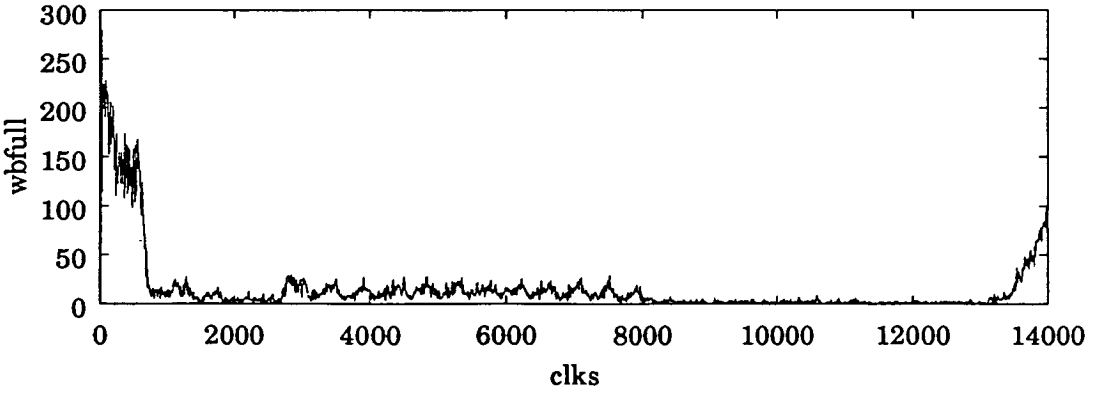


Figure 6.40: Execution profile for matrix multiply showing the number of wait-buffer full events.

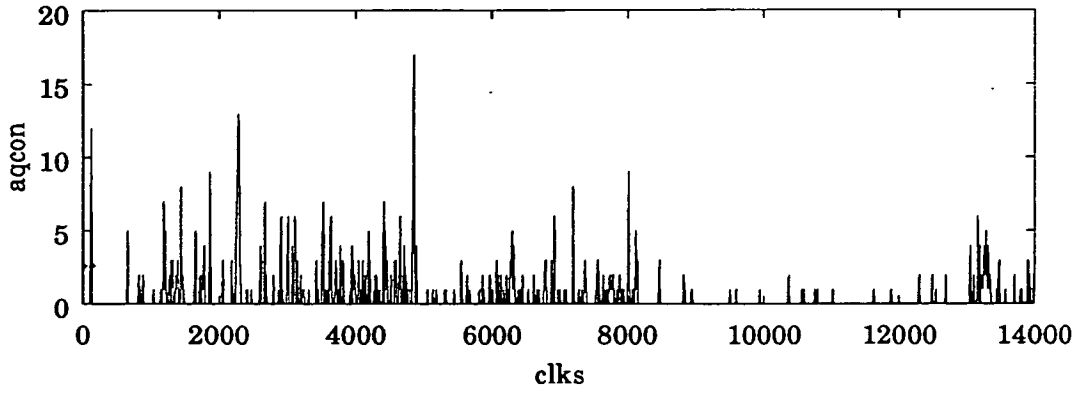


Figure 6.41: Execution profile for matrix multiply showing the number of acknowledge queue conflicts.

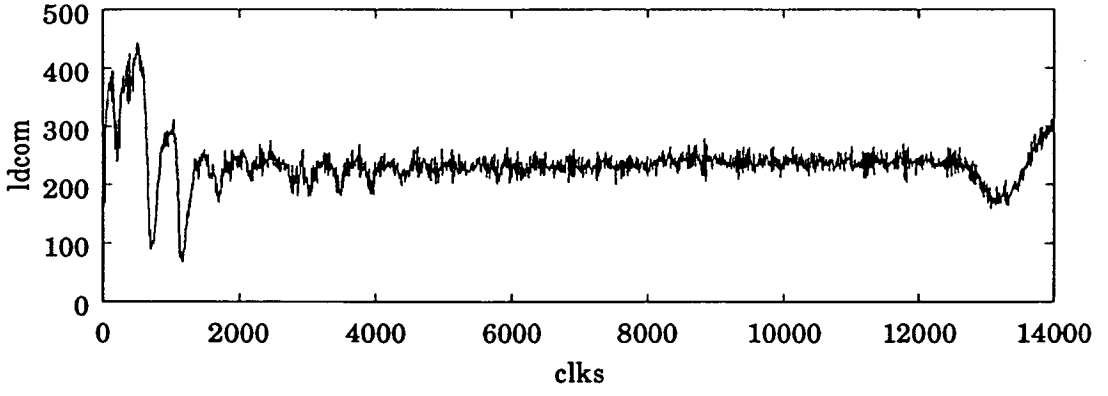


Figure 6.42: Execution profile for matrix multiply showing the number of load combinations.

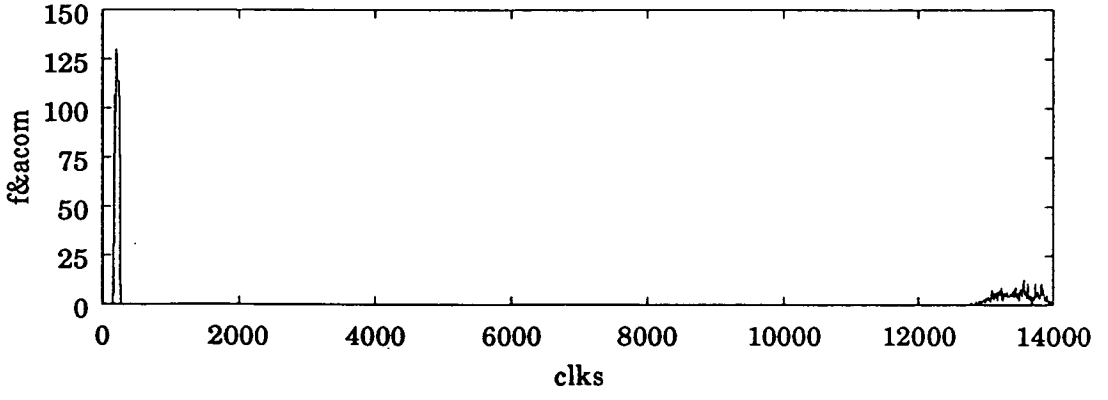


Figure 6.43: Execution profile for matrix multiply showing the number of f&a combinations.

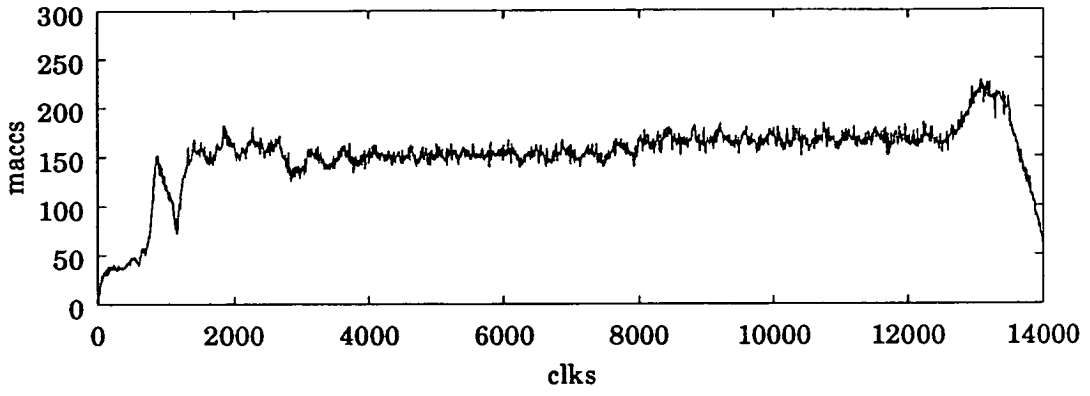


Figure 6.44: Execution profile for matrix multiply showing the number of memory accesses.

6.6.2 Description of matrix multiply profile

At the start of the program each process fetches the index of a result array element at 200 cycles, and the number of active processes (Figure 6.35) increases to the maximum of 1024 in about 100 cycles. There is a resulting peak in the number of load and f&a combinations (Figures 6.42 and 6.43), and wait-buffer full events (Figure 6.40) as the semaphore is accessed. The base addresses of the two operand arrays are then loaded into registers, resulting in another peak in the number of load combinations at 400 cycles.

Each process then starts executing the 32 iterations of the product-sum loop. In each iteration two operand array elements are fetched, multiplied and added to the sum. The accesses to the operand arrays cause contention because they are all to different memory locations, and there is no combining. This is shown by the rise in request queue conflicts (Figure 6.39) and memory latency (Figure 6.38), and results in a reduction of processor and network utilisation (Figures 6.36 and 6.37) at 800 and 1200 cycles. The first iteration shows more features than subsequent iterations, probably because the processes become more "smeared out".

When the iterations are finished the processes write to the result array, shown by the increase in memory accesses (Figure 6.44) at 12500 cycles. Each process

then increments the termination counter, resulting in an increase in f&a combinations starting at 13000 cycles. There are about 1000 cycles between the first and last process to finish. The program finishes at 14031 cycles (when the interrupt is received by the user-interface).

6.6.3 Description of bitonic merge sort profile

The bit1024 program sorts an array of 2048 integers in 66 merge iterations, but only the first 3 iterations are shown in each profile since the subsequent 63 iterations are essentially identical. The interrupt indicating the end of the program is received at 148008 cycles. Only the first iteration is described below, but the description applies equally to all iterations.

Each process first fetches an index which points to the result array entries, with the number of active processes (Figure 6.45) reaching the maximum 1024 in about 100 cycles. Access to the index causes the two peaks in f&a combinations (Figure 6.53) at 100 and 200 cycles, and a high number of wait-buffer full events. Just after the start of the iteration at 400 cycles there is a peak in request queue conflicts (Figure 6.49), resulting in a peak in memory latency (Figure 6.48) and decreases in processor and network utilisation (Figure 6.46 and 6.47).

The processes then calculate the positions of the two operand array entries by performing two inverse perfect shuffle operations. These are mainly arithmetic operations, so the processor and network utilisation is high, and there are many (instruction) load combinations (Figure 6.52). After the shuffles have been calculated the entries are fetched from the operand array, compared and written to the result array, with corresponding peaks in memory accesses (Figure 6.54) at 1100 and 1600 cycles.

The processes then finish the iteration by incrementing the termination counter at 1500 cycles, with about 200 cycles between the first and last process to finish. During the next 900 cycles before the next iteration starts the last process to finish the iteration initialises the next iteration, while the other 1023 processes

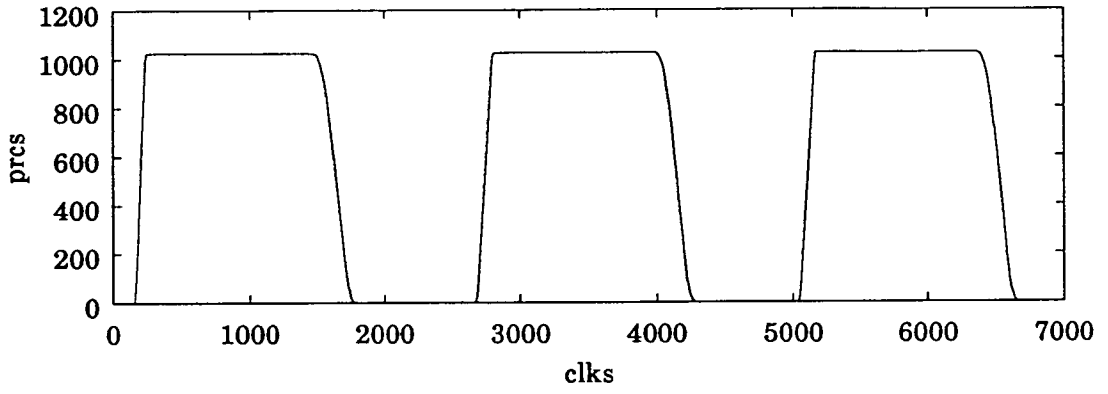


Figure 6.45: Execution profile for bitonic showing the number of active processes.

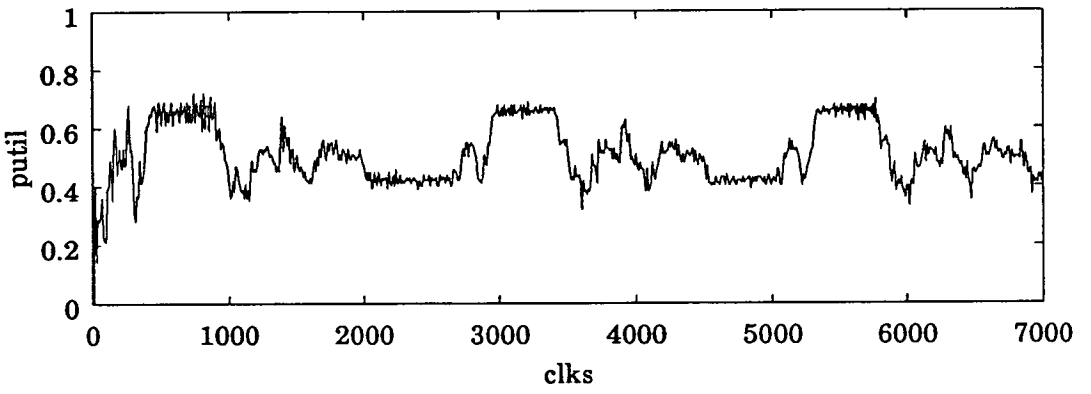


Figure 6.46: Execution profile for bitonic showing the processor utilisation.

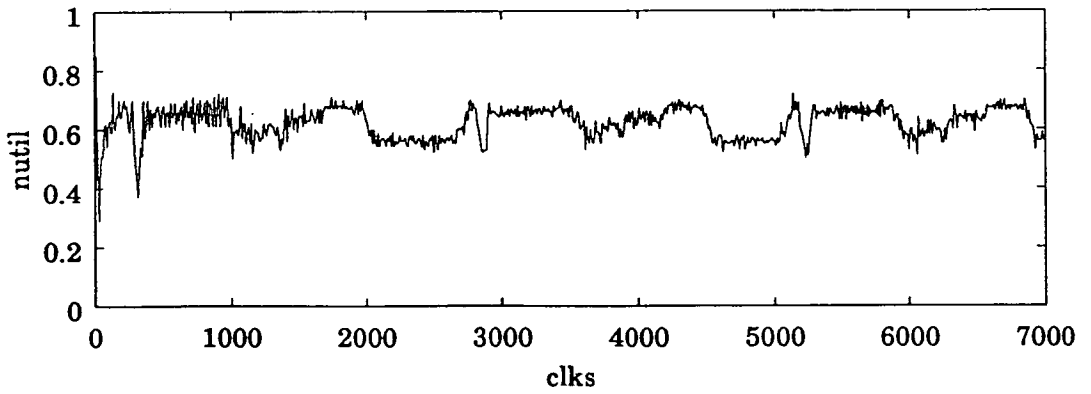


Figure 6.47: Execution profile for bitonic showing the network utilisation.

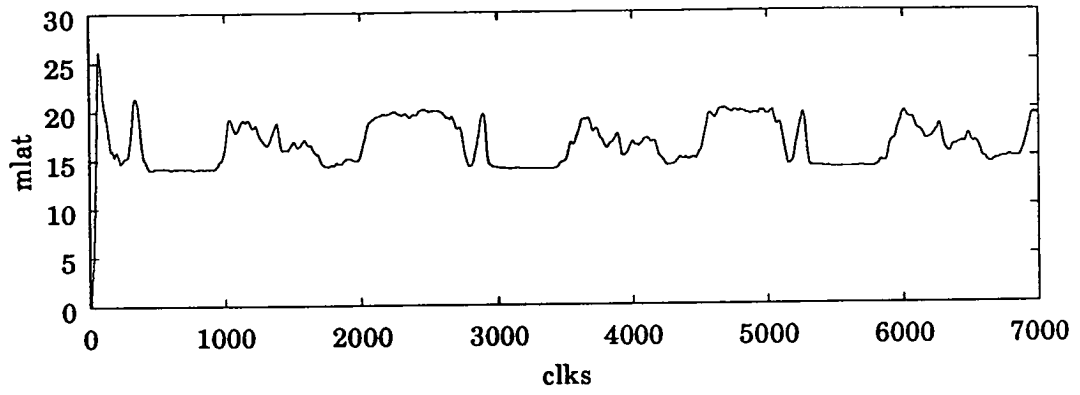


Figure 6.48: Execution profile for bitonic showing the average memory latency.

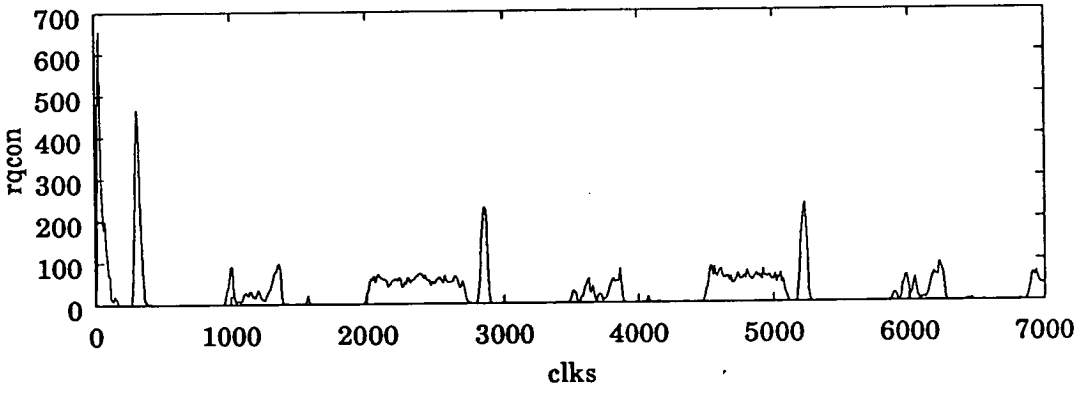


Figure 6.49: Execution profile for bitonic showing the number of request queue conflicts.

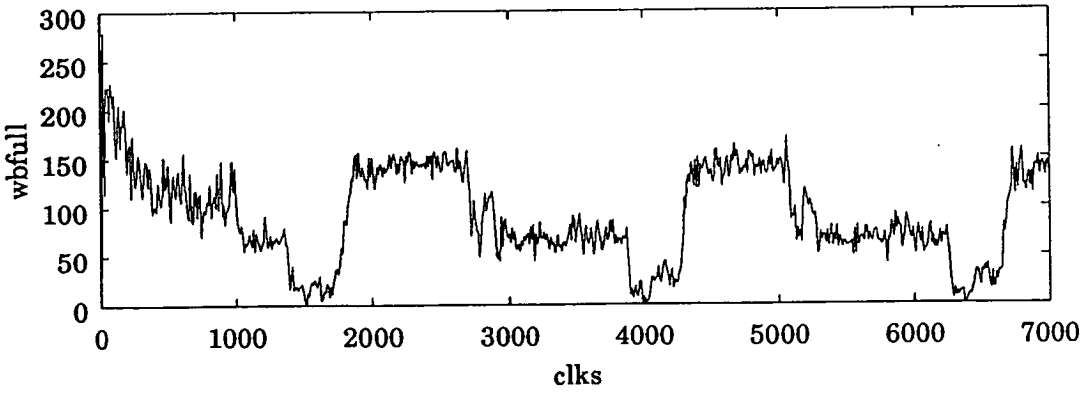


Figure 6.50: Execution profile for bitonic showing the number of wait-buffer full events.

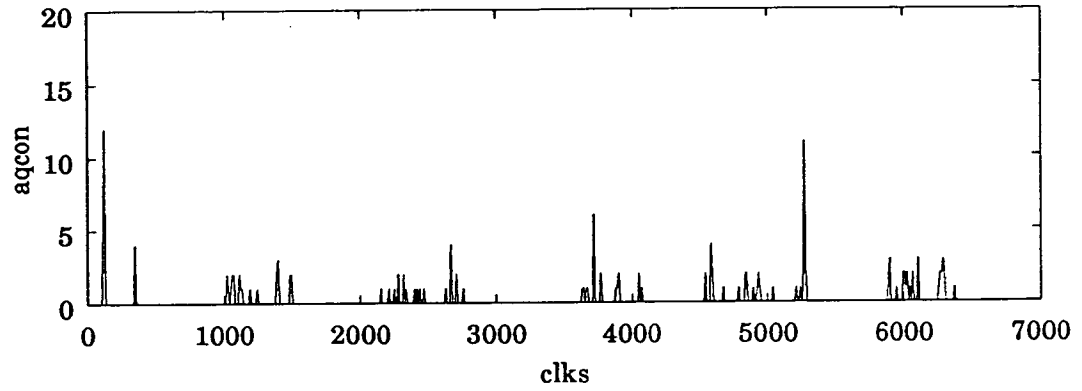


Figure 6.51: Execution profile for bitonic showing the number of acknowledge queue conflicts.

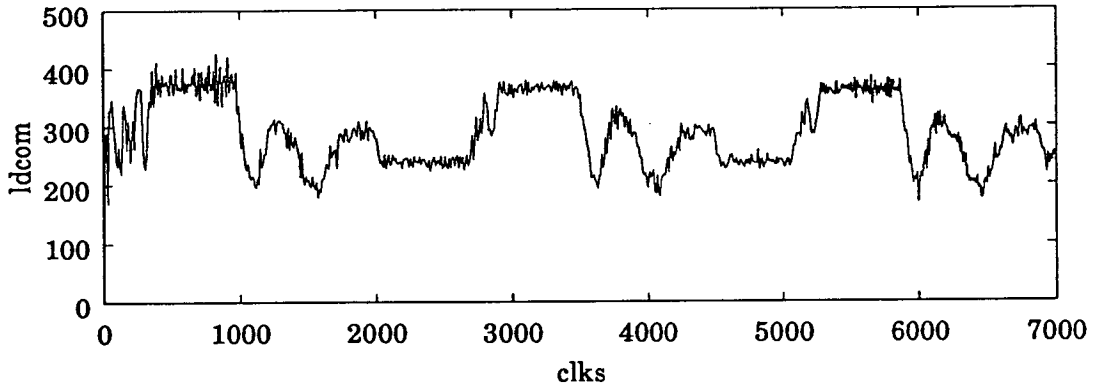


Figure 6.52: Execution profile for bitonic showing the number of load combinations.

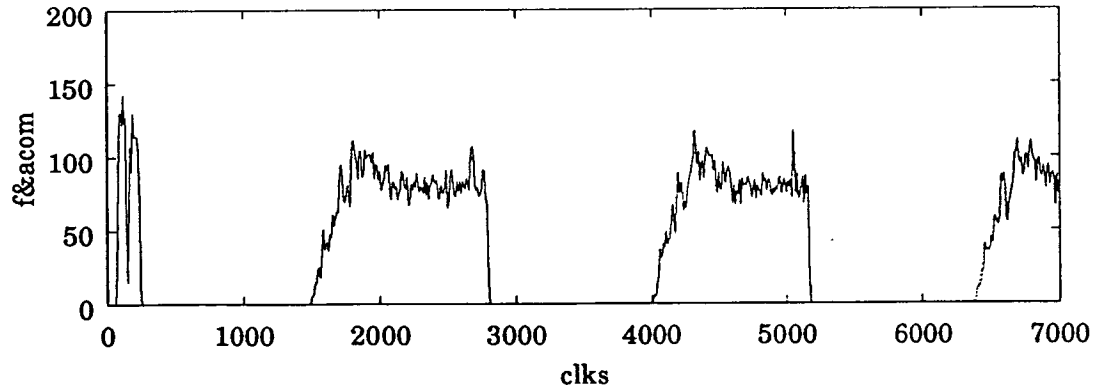


Figure 6.53: Execution profile for bitonic showing the number of f&a combinations.

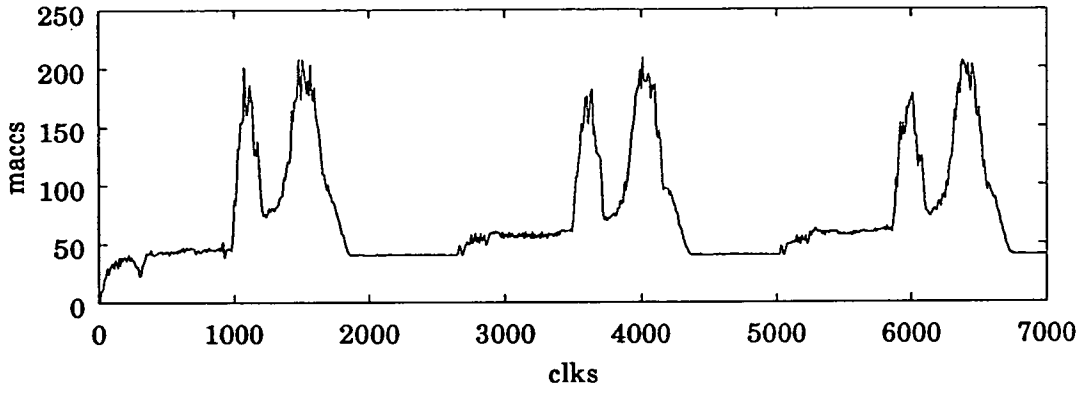


Figure 6.54: Execution profile for bitonic showing the number of memory accesses.

busy-wait. This results in the large peak in f&a combinations from 1500 cycles, and increased memory latency.

6.6.4 Description of Moore shortest path profile

The mool024 program has an irregular profile compared to the other two programs, and the profile is also much longer (as only a part of the bitonic profile was shown). It is therefore more difficult to match changes in the profile to what the processes are executing. There are no wait-buffer full events so this profile is not included.

Initially there is only one process executing, which then places other nodes to be examined into the queue. The other processes busy-wait until a node becomes available on the queue. When a process gets a node it searches all the connected nodes, updates the distance if there is a shorter path, and queues that node.

The profile of active processes (Figure 6.55) shows a slow increase up to the maximum 1024 processes in about 20000 cycles. The number of processes then drops and begins to tail-off at about 40000 cycles, taking another 40000 cycles to actually finish. The interrupt is received by the user-interface at 85871 cycles. The process profile shows that the processes spend their time busy-waiting. The processor and network utilisations (Figures 6.56) and 6.57) show large initial

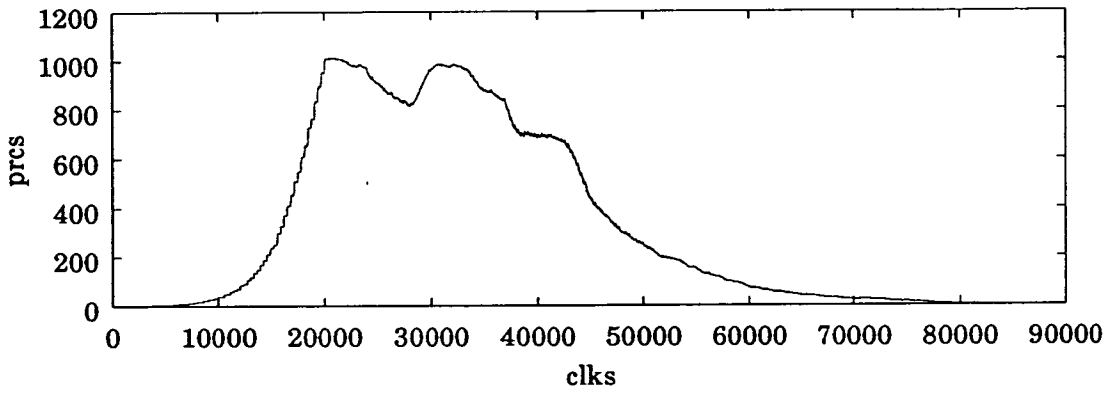


Figure 6.55: Execution profile for Moore showing the number of active processes.

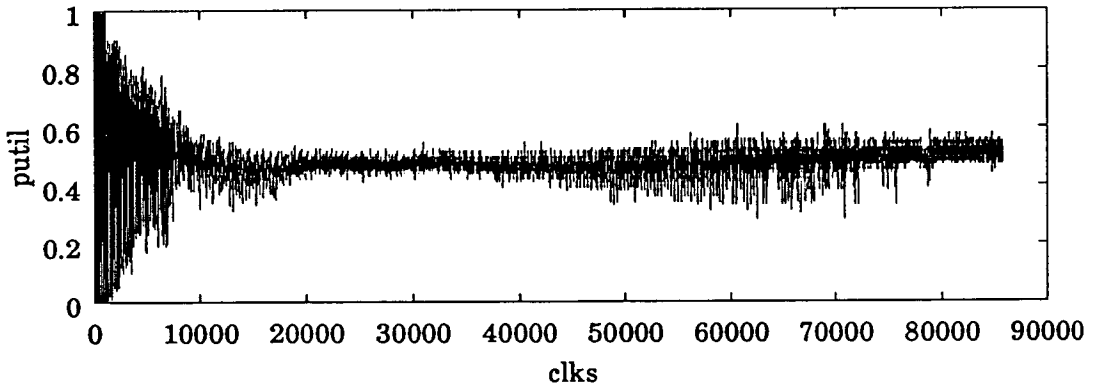


Figure 6.56: Execution profile for Moore showing the processor utilisation.

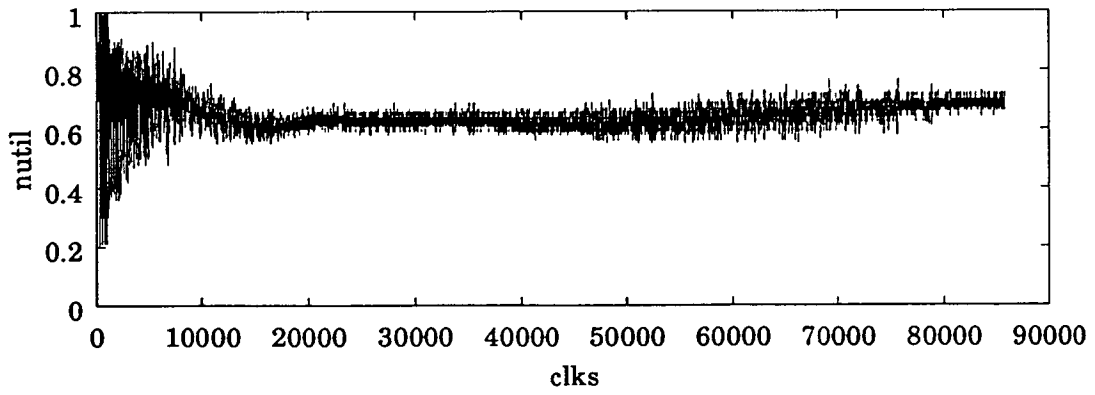


Figure 6.57: Execution profile for Moore showing the network utilisation.

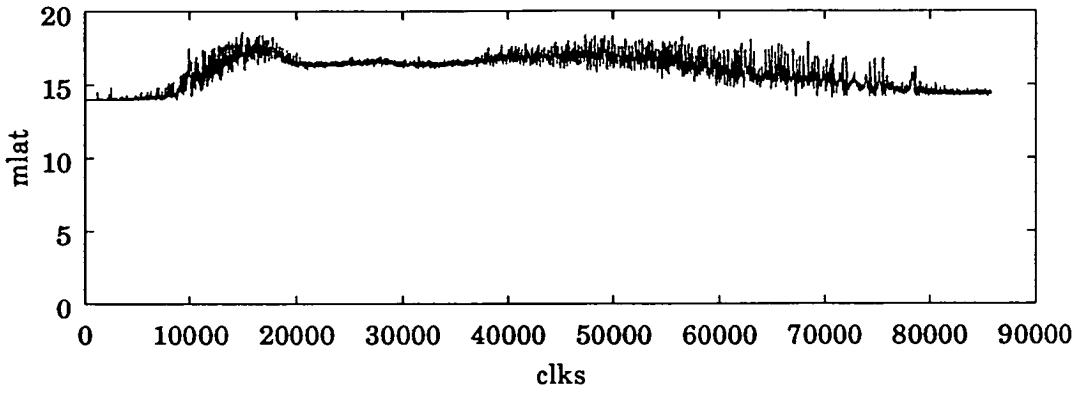


Figure 6.58: Execution profile for Moore showing the average memory latency.

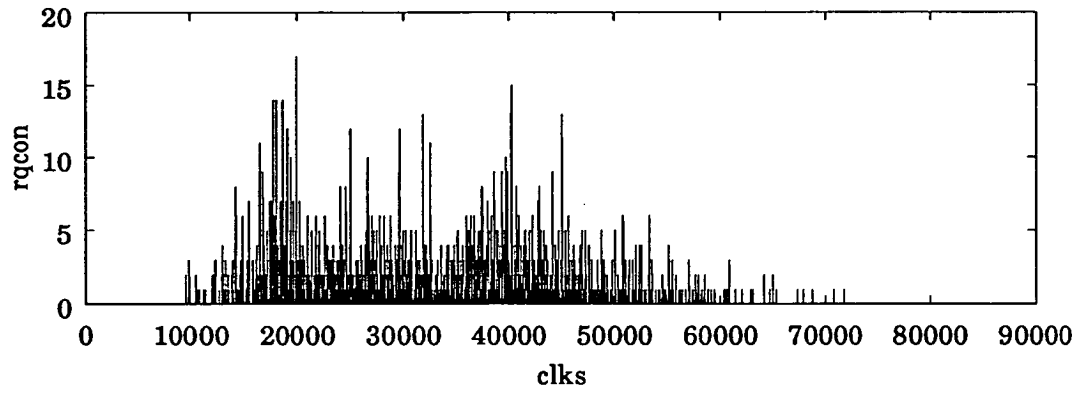


Figure 6.59: Execution profile for Moore showing the number of request queue conflicts.

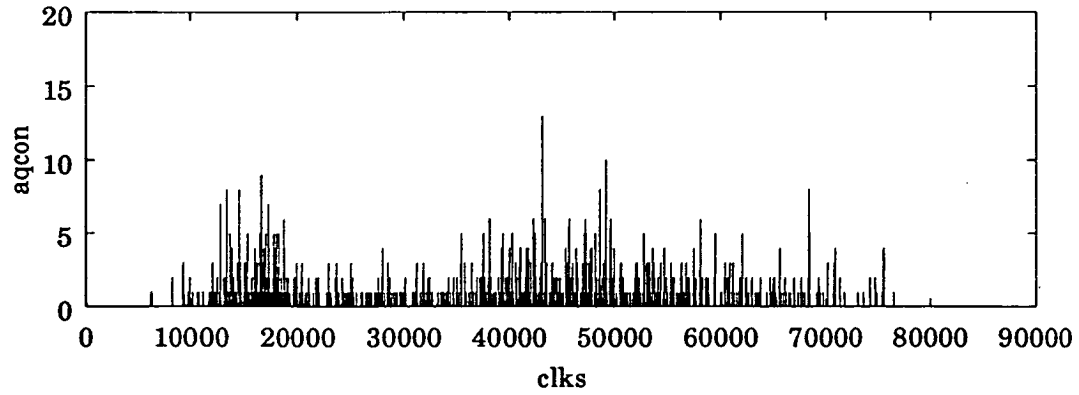


Figure 6.60: Execution profile for Moore showing the number of acknowledge queue conflicts.

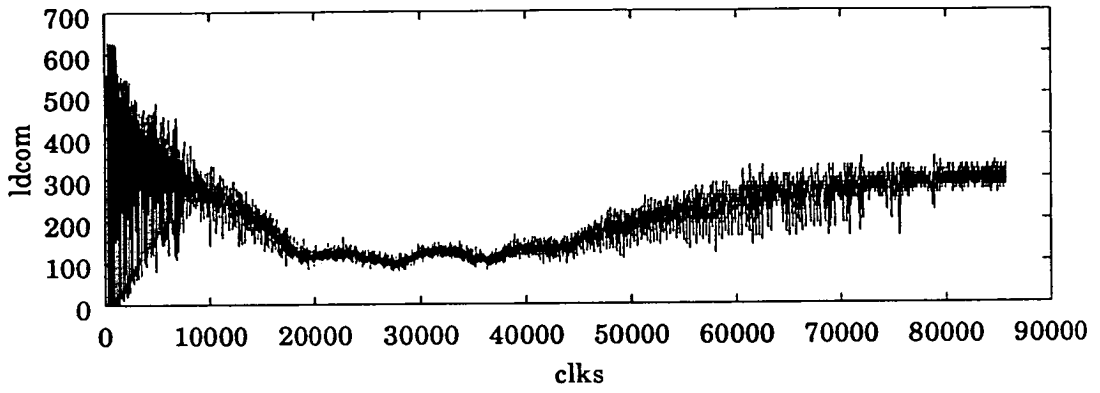


Figure 6.61: Execution profile for Moore showing the number of load combinations.

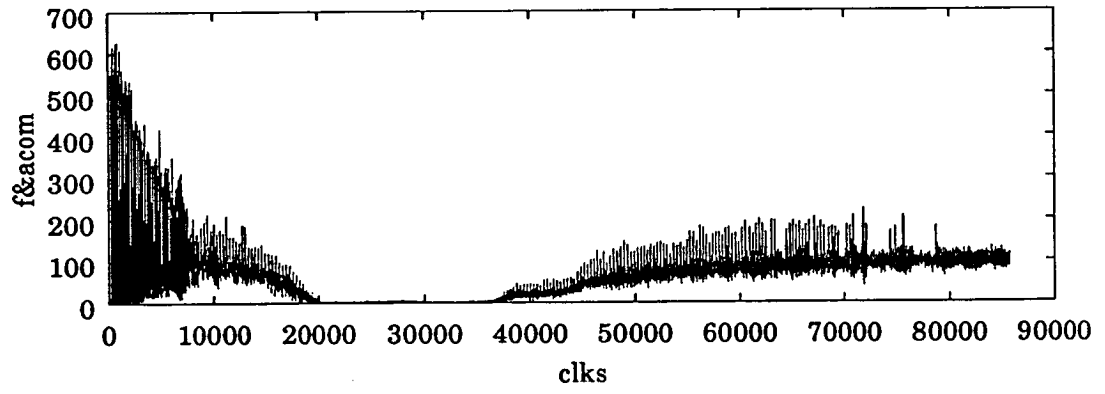


Figure 6.62: Execution profile for Moore showing the number of f&a combinations.

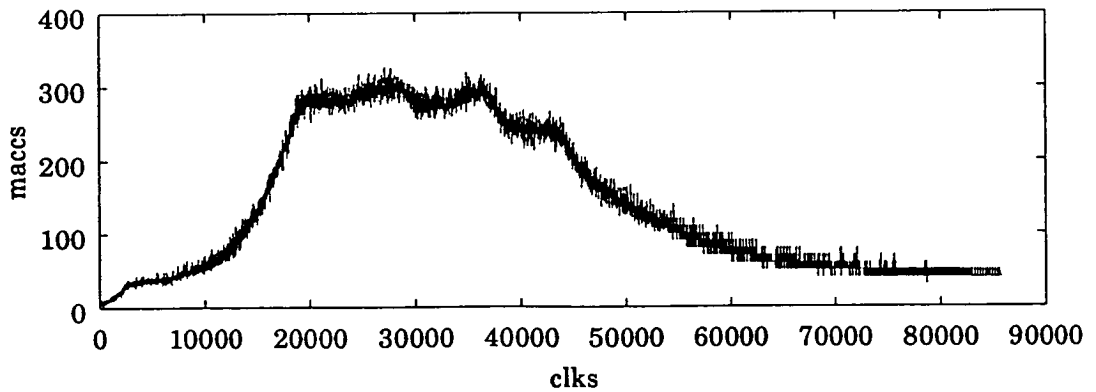


Figure 6.63: Execution profile for Moore showing the number of memory accesses.

oscillations which then average out as the number of active processes increases. Both the request and acknowledge queue conflicts (Figures 6.59 and 6.60) show an increase as the number of active processes increases. The load combinations and f&a combinations (Figures 6.61 and 6.62) both show large initial oscillations, and decrease as more processes become active and there is less busy-waiting. The number of memory accesses increases (Figure 6.63) as more processes do useful work.

6.7 Comments

The execution profiles convey information which cannot be gleaned from the averaged execution statistics, such as the occurance of transient peaks in the memory latency. Such information can be used to improve the implementation of programs, for example the bitonic merge sort profile shows that the time between each iteration is almost as long the iteration itself. During this time most of the processes are idle and spin on a semaphore. It may be possible to change the bitonic program so this is avoided, and so improve the execution time.

This concludes the simulation experiments chapter; in the final chapter the simulation results are summarised, and suggestions for further experiments made.

Chapter 7

Conclusion

In this thesis the Eppl MIMD shared memory multiprocessor design has been described, and its performance evaluated by simulation. The bounds on processor utilisation with varying interleaving level and memory latency, and a synthetic instruction mix, were first investigated using a serial single processor simulator. Next the bounds on network throughput and latency with varying queue size, network load and hot-spot percentage were investigated using a serial isolated network simulator. After this an instruction level simulation of the complete Eppl system, capable of executing realistic programs, was implemented in Occam on a distributed memory multiprocessor using a distributed time-driven simulation methodology. Parallel versions of the matrix multiply, bitonic merge sort and Moore shortest path algorithms were implemented in the Eppl assembly code, using fetch-and-add based process control mechanisms. These benchmark programs were used as system loads for the simulator. Simulation experiments were then carried out using the Eppl simulator in which the speed-up and execution profile of the benchmark programs were investigated, as well as the effect on the performance of the Eppl system of switch queue and wait-buffer size, combining level, interleaving level and relative module speed.

The simulation results are summarised below, and then some suggestions for

future work are made.

7.1 Summary of results

The prominent features of the Eppl design are that it has instruction interleaved processors, the processors and memories are connected by a combining multi-stage network, and the fetch-and-add instruction is used as the synchronisation primitive. A large amount of numerical data has been collected in the Eppl simulations, and these results are drawn together here in two sections focusing on instruction interleaving and combining, and the simulation methodology is also discussed in a third section.

7.1.1 Instruction interleaving

Instruction interleaving has been implemented in a number of single processor designs, the HEP multiprocessor machine, and also more recent hybrid message-passing multiprocessor designs, and is likely to receive further attention in the future. The main advantage of instruction interleaving is that it gives the processor a degree of latency tolerance, since other processes can be executed while memory accesses take place, increasing the processor utilisation. A subsidiary advantage, assuming only a single instruction is issued per process at any one time, is that the processor can be heavily pipelined despite performance limiting branch and data dependencies (it would be possible to issue multiple instructions per stream assuming it can be easily checked that no dependencies are violated). If the parallelism of the program to be executed is not greater than the interleaving level, then there are also no context-switching costs, since the context of all the processes is stored in the processor. Of course there must be sufficient parallelism in the application programs to be able to use all these processes effectively.

The single Eppl processor simulation results in Chapter 3 showed that the

processor utilisation increases as more processes are added, up to a maximum level dependent on the instruction mix and the memory cycle time. With unit memory latency and an average workload, 9 processes are needed to achieve the maximum utilisation. When the memory latency is increased by increasing the number of memory pipeline stages, more processes must be added to maintain the same utilisation. If more processes are added after the maximum utilisation has been reached then the throughput of each individual process is correspondingly reduced.

The complete Eppl system simulations in Chapter 6 showed, in the speed-up simulations, that the processor utilisation falls as the system size is increased since the fixed interleaving level of 16 processes is not enough to compensate for the increasing memory latency. With 256 processors the average memory latency is between 18 and 20 cycles, with transient peaks reaching values higher than this. The relative module speed simulations showed that as the network speed is increased the processor utilisation quickly reaches the maximum, and therefore fewer processes are necessary to maintain the same utilisation.

The interleaving level and execution profile simulations showed that each benchmark program has an optimal interleaving level related to the program parallelism. For all three programs the best performance was achieved with a system parallelism less than the program parallelism. For the Moore program this was because the majority of processes were not doing useful work, while for the matrix multiply and bitonic programs the highest interleaving level (of 64 processes) caused increased contention within the processors and network.

The interleaving level required to utilise the processor pipeline fully is related to the length of the execution pipeline and the memory latency. In an implementation of the processor the interleaving level would be fixed, and thus the size of the system could only be scaled within this fixed limit. Also in small systems the interleaving level would be higher than required by the memory latency. One possible solution would be to dynamically vary the interleaving level by scheduling

new processes on demand, as in the HEP processor.

The processor design assumed that integer multiply and divide operations could be done in a single pipeline stage, which is not realistic. In an implementation a number of pipeline stages would be required, and the interleaving level would have to be increased to maintain the processor utilisation. Similarly the benchmark programs were hand coded and included no subroutines or explicit context-switching, thus reducing their execution time considerably. The simulation results can be regarded as optimistic because of these assumptions, and represent the best performance one could expect from an implementation of this design.

7.1.2 Combining and fetch-and-add

Multistage networks have received considerable attention in the context of shared memory multiprocessor designs, and have been implemented in the HEP and RP3 machines. A multistage network can provide similar throughput to a full crossbar but with higher latency. Memory hot-spots can seriously degrade the performance of these networks by causing tree-saturation. Combining requests to the same memory location reduces the number of accesses to the hot memory, thereby reducing the access time to the hot-spot and improving overall network throughput. The fetch-and-add instruction and associated algorithms have been developed by the Ultracomputer project, and with combining can provide efficient synchronisation.

The isolated Eppl network simulations in Chapter 3 showed that with a uniform memory address distribution the network throughput reaches above 90% as the buffer size is increased to 32 packets, but that the network latency also increases. A small buffer size of 4 to 8 packets is thus optimal. When a hot-spot is introduced, which can be as small as 1 in every 200 memory accesses, the network throughput is much reduced and does not improve when the queue size is increased.

The complete system Eppl simulations in Chapter 6 showed, in the queue size simulations, that with pairwise combining a queue size of 4 to 8 packets also provides acceptable performance. The wait-buffer size simulations showed that if the wait-buffers are not large enough to accept most combinations then performance is reduced as the queue size is increased, due to the increased memory latency. As the system size is increased the wait-buffer size must obviously be increased to cope with the increased number of combinations. A wait-buffer size of 16 packets or larger is required for the 256 processor system. The combining level simulations showed that with no combining the network performance is much reduced. With 16 or more processors none of the programs finished within the simulation time available. Overall, pairwise combining appears to be sufficient to maintain performance for the simulated system sizes and benchmark programs, and increasing the combining level does not produce a worthwhile improvement in performance.

Because each processor executes a number of processes the number of possible concurrent requests to the same memory location is higher than in a machine with single instruction stream processors. Larger wait-buffer sizes are therefore necessary, and the number of combinations increases with increased interleaving level. The execution profiles show there is more load combining than fetch-and-add combining, the majority of which are instruction fetches. Spinning on a lock with fetch-and-add does cause the average memory latency to increase, although no tree-saturation occurs.

The switch design assumed that a packet could be combined and clocked through the switch in a single cycle, which is also not realistic. In an implementation the switch would be pipelined, which could increase the network latency. Pairwise combining switches with one-input queues have already been designed and fabricated [67, 68]. The high hardware cost of multistage networks has prompted investigation of alternative networks [133], but regardless of the network topology some form of combining will still be necessary.

7.1.3 Time-driven simulation

Simulation of large Eppl systems at the instruction level would not have been feasible using a conventional workstation due to the memory as well as the CPU requirements, and so a parallel simulator was used. To the author's knowledge the Eppl simulator is the largest simulator of its kind, and has provided detailed and accurate simulation results of larger system sizes than has been previously achieved. The simulator speed-up figures show that the performance of the simulator becomes communication bound using the larger domain sizes. Simulating a 64 processor system on a 64 transputer domain results in an order of magnitude speed-up over simulating the same system on a 4 transputer domain.

In a distributed time-driven simulation the simulator modules synchronise using message-passing. This allows more parallelism than a shared memory time-driven simulation using barrier synchronisation, because connected modules synchronise directly by the exchange of packets. The clocking and synchronisation method described here could be used for other time-driven simulators to be implemented on distributed memory multiprocessors or shared memory multiprocessors supporting message-passing primitives.

The Occam language allows the parallelism of the Eppl design at the module level to be easily expressed, and the packet communication of the Eppl interconnection network can be naturally implemented using message-passing. The main drawback, due to the static nature of Occam and the transputer domains, is the re-compilation of the simulator necessary for each different domain size.

7.2 Future work

The suggestions for future work are divided into practical improvements which could be undertaken immediately, and more long term directions for further research.

7.2.1 Immediate improvements

Both the Eppl design and simulator could benefit from improvements. In the case of the Eppl design to increase it's functionality without changing its structure, and in the case of the simulator to increase it's functionality and performance.

The Eppl processor should have floating point instructions, which could be simply added in the same manner as the current integer arithmetic functions. Some extra flags to deal with floating point exceptions, and instructions for type conversion would also be required. The combining mechanism in the switch nodes should be extended so that all different types of memory access can be combined, not just load and fetch-and-add accesses. Also a more general fetch-and-ϕ operation could be added (where ϕ is one of or, min, max, add, *etc* [72]), allowing the most efficient instruction for the application to be used.

Further developments to the simulator should include trying to improve its performance by reducing the number of packets that are sent between the modules, and also by optimising the placement of the modules (though this may be a non-trivial problem [134]). The number of statistics that the simulator collects should be increased to include the number of ready signals sent, the general instruction type executed, and the number of memory requests and accesses should be broken down into the individual types. The ability to collect memory address traces efficiently should be added, so these can be used to drive higher performance trace-driven simulators. The collection of the wait-buffer full events should be corrected so it counts all wait-buffer conflicts. The simulator should also allow the size of the acknowledge queues to be separately set.

The addition of floating-point to the processors would allow a wider range of benchmark programs to be implemented, and the simulator could also be used to investigate parallel algorithms for real process scheduling [76, 77] and dynamic memory allocation [135] for example.

7.2.2 Further research

Areas for further research involve adding to the structure of the Eppl design. An obvious next step would be to investigate the caching of instructions and/or data in each processor. Caching would have the effect of reducing network traffic and latency, and therefore fewer processes would be required per processor. In the situation where all the processes are executing the same program (as in the benchmark programs) a high hit rate could be expected from an instruction cache. An instruction cache would also be the simplest to add as there is no consistency problem. With data caches some form of cache coherence is necessary [136, 137]. A possible solution would be not to cache fetch-and-add accesses, or to tag memory words which may not be cached. The RP3 for example allows pages to be marked non-cacheable, and cache lines can be selectively invalidated [3]. The easiest cache organisation to include in the design would be a fully-associative cache with line size 1, which would require no change to the network design. If the cache units were added between the processor interface and the network then no change to the processor would be required either.

To investigate paged virtual memory using the Eppl design would require the addition of a translation look-aside buffer (TLB) to the processors, and some sort of block transfer page i/o mechanism to a (simulated) mass storage device. A current problem is maintaining consistency between the TLB's, and various mechanisms have been proposed [138, 139]. Other topics for research include operating system issues, for example process scheduling, process protection, and memory allocation. Some investigations into using fetch-and-add based operating primitives have already been carried out [76, 77], and an implementation of a UNIX-like operating system on the RP3 is being undertaken [140].

This concludes the final chapter of the thesis; the appendices following this chapter contain the assembly code listings of the benchmark programs, and the detailed simulation results from the complete Eppl system simulations.

Bibliography

[1] Arvind and R. Ianucci, *Parallel computing in Science and Engineering*, vol. LNCS 295, ch. Two fundamental issues in multiprocessing, pp. 61–88. Springer-Verlag, 1987.

[2] L. Bhuyan, Q. Yang, and D. Agrawal, "Performance of multiprocessor interconnection networks," *IEEE Computer*, pp. 25–37, Feb. 1989.

[3] W. Brantley, K. McAuliffe, and J. Weiss, "RP3 Processor-Memory Element," in *Proceedings of the International Conference on Parallel Processing*, pp. 782–797, Aug. 1985.

[4] M. Dubois, F. Briggs, I. Patil, and M. Balakrishnan, "Trace-driven simulations of parallel and distributed algorithms in multiprocessors," in *Proceedings of the International Conference on Parallel Processing*, pp. 909–916, Aug. 1986.

[5] A. Norton and G. Pfister, "A methodology for predicting multiprocessor performance," in *Proceedings of the International Conference on Parallel Processing*, pp. 772–781, Aug. 1985.

[6] P. Tang, P.-C. Yew, and C.-Q. Zhu, "Processor self-scheduling in large multiprocessor systems," in *2nd. SIAM Conference on Parallel Processing for Scientific Computing*, Nov. 1985.

[7] J. Archibald and J. Baer, "Cache coherance protocols: Evaluation using a multiprocessor simulation model," *ACM Transactions of computer systems*, pp. 273–298, Nov. 1986.

[8] T. Feng, "A Survey of Interconnection Networks," *IEEE Computer*, vol. 14, pp. 12–27, Dec. 1981.

[9] C. Ramamoorthy and H. Li, "Pipeline Architecture," *ACM Computing Surveys*, vol. 9, pp. 61–102, Mar. 1977.

[10] K. Hwang and F. Briggs, *Computer architecture and parallel processing.* McGraw-Hill, 1985.

[11] N. Topham, A. Omondi, and R. Ibbett, "On the design and performance of conventional pipelined architectures," *Journal of Supercomputing*, no. 1, pp. 353–393, 1988.

[12] S. McFarling and J. Hennessy, "Reducing the Cost of Branches," in *Proceedings of the International Symposium on Computer Architecture*, pp. 396–403, 1986.

[13] D. Lilja, "Reducing the branch penalty in pipelined processors," *IEEE Computer*, vol. 21, pp. 47–55, July 1988.

[14] R. Hockney and C. Jesshope, *Parallel Computers.* Adam Hilger, 1981.

[15] J. Goodman, J.-T. Hsieh, K. Liou, A. Plezkun, P. Schechter, and H. Young, "PIPE: A VLSI Decoupled Architecture," in *Proceedings of the International Symposium on Computer Architecture*, pp. 20–27, 1985.

[16] R. Lee, P.-C. Yew, and D. Lawrie, "Data prefetching in shared memory multiprocessors," in *Proceedings of the International Conference on Parallel Processing*, pp. 28–31, Aug. 1987.

[17] T. Chen, "Parallelism, Pipelining and Computer Efficiency," *Computer Design*, pp. 69–74, Jan. 1971.

[18] M. Flynn, A. Podvin, and K. Shimizu, "A multiple instruction stream processor with shared resources," in *Parallel processor system technologies and applications*, pp. 251–286, 1970.

[19] N. Topham, "A Parallel Machine Description," Internal Document, Department of Computer Science, University of Edinburgh, 1986.

[20] C. Staley and S. Butner, "A Feasibility Study of the Circulating Context Multiprocessor (CCMP)," in *Proceedings of the International Conference on Parallel Processing*, pp. 455–462, Aug. 1986.

[21] A. Hartmann and J. Ullman, "Model Categories for Theories of Parallel Systems," Technical Report PP-341-86, MCC, 1986. reprinted in Parallel Computing by Lipovski and Malek.

[22] Arvind and D. Culler, "Dataflow architectures," *Annual Review of Computer Science*, vol. 1, pp. 225–253, 1986.

[23] Arvind and R. Ianucci, "A Critique of Multiprocessing Von Neumann Style," in *Proceedings of the International Symposium on Computer Architecture*, pp. 426–436, 1983.

[24] D. Gajski, D. Padua, D. Kuck, and R. Kuhn, "A second opinion on dataflow machines and languages," *IEEE Computer*, pp. 58–69, Feb. 1982.

[25] R. Hockney, "Performance of parallel computers," in *High speed computation* (J. Kowalik, ed.), pp. 159–175, NATO ASI Series, 1984.

[26] N. Topham, "An Efficient Architecture for Concurrent VLSI Systems," Internal document, Department of Computer Science, University of Edinburgh, 1987.

[27] H. Jordan, "Performance measurements on HEP - a pipelined MIMD computer," in *Proceedings of the International Symposium on Computer Architecture*, pp. 207–218, 1983.

[28] E. Miller, "A Multiple-Stream Registerless Shared-Resource Processor," *IEEE Transactions on Computers*, vol. 23, pp. 277–285, Mar. 1974.

[29] L. Shar and E. Davidson, "A Multiminiprocessor Implemented through Pipelining," *IEEE Computer*, pp. 42–51, Feb. 1974.

[30] W. Kaminsky and E. Davidson, "Developing a Multiple-Instruction-Stream Single-Chip Processor," *IEEE Computer*, pp. 66–76, Dec. 1979.

[31] K. Shimizu, E. Goto, and S. Ichikawa, "CPC(Cyclic Pipeline Computer) - An Architecture Suited for Josephsonand Pipelined-Memory Machines," Technical Report 86-19, University of Tokyo, Dept. Information Science, Nov. 1986.

[32] T. Turton, "A Proposed High-Speed Computer Design," *Computer Architecture News*, pp. 7–21, 1979.

[33] N. Topham, A. Omondi, and R. Ibbett, "Context-Flow: An alternative to conventional pipelined architectures," *Journal of Supercomputing*, no. 2, pp. 29–53, 1988.

[34] D. Rogers and N. Topham, "Implementing a practical context flow machine," Internal document, Department of Computer Science, University of Edinburgh, 1989.

[35] M. Dubois, "Ä cache-based multiprocessor with high efficiency," in *Proceedings of the International Conference on Parallel Processing*, pp. 646–648, Aug. 1985.

[36] T. Mankovich, V. Popescu, and H. Sullivan, "CHoPP principles of operation," in *Proceedings 2nd International Conference on Supercomputing* (L. Kartashev, ed.), pp. 2–10, 1987.

[37] R. Buehrer and K. Ekanadham, "Incorporating dataflow ideas into von Neumann processors for parallel execution," *IEEE Transactions on Computers*, vol. 36, pp. 1515–1522, Dec. 1987.

[38] J. Solworth, "The microflow architecture," in *Proceedings of the International Conference on Parallel Processing*, pp. 113–117, 1988.

[39] R. Iannucci, "Toward a dataflow/von Neumann hybrid architecture," in *Proceedings of the International Symposium on Computer Architecture*, pp. 131–140, May 1988.

[40] M. Pease, "The indirect binary n-cube microprocessor array," *IEEE Transactions on Computers*, vol. 26, pp. 548–573, May 1977.

[41] D. Lawrie, "Access and Alignment of Data in Array Processor," *IEEE Transactions on Computers*, vol. 24, pp. 1145–1155, Dec. 1975.

[42] M. Franklin, D. Wann, and W. Thomas, "Pin Limitations and Partioning of VLSI Interconnection Networks," *IEEE Transactions on Computers*, vol. 31, pp. 1109–1116, Nov. 1982.

[43] M. Franklin and S. Dhar, "Interconnection Networks: Physical Design and Performance Analysis," *Journal of Parallel and Distributed Computing*, vol. 1, no. 3, pp. 352–372, 1986.

[44] H. Siegel and R. McMillen, "The Multistage Cube: A Versatile Interconnection Network," *IEEE Computer*, vol. 14, pp. 65–76, Dec. 1981.

[45] W. Fuchs, J. Abraham, and H. Kuang-Hua, "Concurrent Error Detection in VLSI Interconnection Networks," in *Proceedings of the International Symposium on Computer Architecture*, pp. 309–315, 1983.

[46] L. Goke and G. Lipovski, "Banyan Networks for Partitioning Multiprocessor Systems," in *Proceedings of the International Symposium on Computer Architecture*, pp. 21–28, 1973.

[47] J. Patel, "Performance of Processor-Memory Interconnections for Multiprocessors," *IEEE Transactions on Computers*, vol. 30, pp. 771–780, Oct. 1981.

[48] C. Wu and T. Feng, "On a Class of Multistage Interconnection Networks," *IEEE Transactions on Computers*, vol. 29, pp. 696–702, Aug. 1980.

[49] K. Batcher, "The Flip network in Staran," in *Proceedings of the International Conference on Parallel Processing*, pp. 65–71, Aug. 1976.

[50] A. Tripathi and G. Lipovski, "Packet Switching in Banyan Networks," in *Proceedings of the International Symposium on Computer Architecture*, pp. 160–167, Apr. 1979.

[51] A. Gottlieb, R. Grishman, C. Kruskal, K. McAuliffe, L. Rudolph, and M. Snir, "The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Machine," in *Proceedings of the International Symposium on Computer Architecture*, pp. 27–42, Apr. 1982.

[52] S. Lundstrom, "Applications considerations in the system design of highly concurrent multiprocessors," *IEEE Transactions on Computers*, vol. 36, pp. 1292–1309, Nov. 1987.

[53] A. Gottlieb and J. Schwartz, "Networks and Algorithms for Very-Large-Scale Parallel Computation," *IEEE Transactions on Computers*, pp. 27–36, Jan. 1982.

[54] P.-Y. Chen, D. Lawrie, P.-C. Yew, and D. Padua, "Interconnection networks using shuffles," *IEEE Computer*, pp. 55–64, Dec. 1981.

[55] M. Kumar and J. Jump, "Performance Enhancement in Buffered Delta Networks using Crossbar Switches and Multiple Links," *Journal of Parallel and Distributed Computing*, vol. 1, pp. 81–103, 1984.

[56] D. Dias and J. Jump, "Analysis and Simulation of Buffered Delta Networks," *IEEE Transactions on Computers*, vol. 30, pp. 273–282, 1981.

[57] C. Kruskal and M. Snir, "The Performance of Multistage Interconnection Networks for Parallel Processors," *IEEE Transactions on Computers*, pp. 1091–1098, Dec. 1983.

[58] N. Pfister and V. Norton, "Hot-Spot Contention and Combining in Multistage Interconnection Networks," in *Proceedings of the International Conference on Parallel Processing*, pp. 790–797, Aug. 1985.

[59] R. Lee, "On hot spot contention," *Computer Architecture News*, pp. 15–20, Nov. 1985.

[60] T. Lang and L. Kurasaki, "Nonuniform traffic spots (NUTS) in multistage interconnection networks," in *Proceedings of the International Conference on Parallel Processing*, pp. 191–195, Aug. 1988.

[61] L. Wu, "Mixing Traffic in a Buffered Banyan Network," in *Proceedings 9th Data Communications Symposium*, pp. 134–139, Sept. 1985.

[62] M. Kumar and G. Pfister, "The Onset of Hot Spot Contention," in *Proceedings of the International Conference on Parallel Processing*, pp. 29–34, 1986.

[63] C. Staley, *Design and analysis of the CCMP: a highly expandable shared memory parallel computer.* PhD thesis, UC San Diego, 1986.

[64] P.-C. Yew, N.-F. Tzeng, and D. Lawrie, "Distributing hot-spot addressing in large-scale multiprocessors," in *Proceedings of the International Conference on Parallel Processing*, pp. 51–58, Aug. 1986.

[65] R. Thomas, "Behaviour of the Butterfly Parallel Processor in the presence of memory hot spots," in *Proceedings of the International Conference on Parallel Processing*, pp. 46–50, 1986.

[66] R. Rettberg and R. Thomas, "Contention is no Obstacle to Shared-Memory Multiprocessing," *Communications of the ACM*, vol. 29, pp. 1202–1212, Dec. 1986.

[67] S. Dickey, R. Kenner, and J. Solworth, "A VLSI combining network for the NYU Ultracomputer," in *Proceedings IEEE International Conference on Computer Design*, (Port Chester, NY.), pp. 110–113, Oct. 1985.

[68] S. Dickey, A. Gottlieb, R. Kenner, and Y.-S. Liu, "Designing VLSI network nodes to reduce memory traffic in a shared memory parallel computer," Tech. Rep. Ultracomputer note 125, NYU Ultracomputer project, 1986.

[69] H. Park, *Smart switching node in an MIMD architecture.* PhD thesis, NY Polytechnic University, Dec. 1986.

[70] S. Wang, Y. Hsu, and C. Tan, "A high performance VLSI message switch for multi-processor systems," Tech. Rep. RC 14718, IBM Research division, 1989.

[71] A. Gottlieb, B. Lubachevsky, and L. Rudolph, "Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors," *ACM Transactions on Programming Languages and Systems*, vol. 5, pp. 164–189, Apr. 1983.

[72] C. Kruskal, L. Rudolph, and M. Snir, "Efficient synchronisation on multiprocessors with shared memory," *ACM Transactions on Programming Languages and Systems*, vol. 10, pp. 579–601, Oct. 1988.

[73] G. Lee, C. Kruskal, and D. Kuck, "The effectiveness of combining in shared memory parallel computers in the presence of hot spots," in *Proceedings of the International Conference on Parallel Processing*, pp. 35–41, 1986.

[74] G. Lee, "A performance bound of multistage combining networks," *IEEE Transactions on Computers*, vol. 38, pp. 1387–1395, Oct. 1989.

[75] G. Andrews and F. Schneider, "Concepts and Notations for Concurrent Programming," *ACM Computing Surveys*, vol. 15, pp. 3–41, Mar. 1983.

[76] L. Rudolph, *Software structures for ultraparallel computing.* PhD thesis, New York University, NY, 1982.

[77] J. Wilson, *Operating system data structures for shared-memory MIMD machines with fetch-and-add.* PhD thesis, New York University, June 1988.

[78] T. Axelrod, "Effects of synchronisation barriers on multiprocessor performance," *Parallel Computing*, no. 3, pp. 129–140, 1986.

[79] G. Almasi and A. Gottlieb, *Highly parallel computing.* Benjamin Cummings, 1989.

[80] P. Heidelberger and S. Lavenberg, "Computer performance evaluation methodology," *IEEE Transactions on Computers*, vol. 33, pp. 1195–1220, Dec. 1984.

[81] M. Ajmone-Marsan, G. Balbo, and G. Conte, *Performance models of multiprocessor systems.* MIT Press, 1986.

[82] A. Pashtan, "A prolog implementation of an instruction-level processor simulator," *Software Practice and Experience*, vol. 17, pp. 309–318, May 1987.

[83] D. Coelhol, *The VHDL Handbook.* Kluwer Academic, 1989.

[84] D. May and C. Keane, "Compiling OCCAM into silicon," Technical Note 23, Inmos Ltd., Bristol, UK, Feb. 1987.

[85] D. Shepherd, "The role of OCCAM in the design of the T800," Technical Note 47, Inmos Ltd., Bristol, UK, Sept. 1988.

[86] C. Stunkel and W. Fuchs, "TRAPEDS: Producing traces for multicomputers via execution driven simulation," in *Proceedings of the International Conference on Measurement and Modelling of Computer Systems*, pp. 70–78, May 1989.

[87] K. So, F. Darema, D. George, and V. Norton, "PSIMUL - A system for parallel simulation of parallel systems," Tech. Rep. RC 11674, IBM Research Division, 1987.

[88] S. Baylor and B. Rathi, "A study of the memory reference behaviour of Engineering/Scientific applications in parallel processors," in *Proceedings of the International Conference on Parallel Processing*, vol. 1, pp. 78–82, Aug. 1989.

[89] M. Kumar and K. So, "Trace driven simulation for studying MIMD parallel computers," in *Proceedings of the International Conference on Parallel Processing*, vol. 1, pp. 68–72, Aug. 1989.

[90] F. Darema-Rogers, G. Pfister, and K. So, "Memory access patterns of parallel scientific programs," in *Proceedings ACM Sigmetrics conference on measuring and modelling computer systems*, pp. 46–58, May 1987.

[91] S. Abraham, A. Gottlieb, and C. Kruskal, "Simulating shared memory parallel computers," Tech. Rep. Ultracomputer note 70, Ultracomputer research laboratory, Apr. 1984.

[92] J. Peacock, J. Wong, and E. Manning, "Distributed simulation using a network of processors," *Computer Networks*, vol. 3, pp. 44–56, 1979.

[93] M. Franklin, D. Wann, and K. Wong, "Parallel machines and algorithms for discrete event simulation," in *Proceedings of the International Conference on Parallel Processing*, pp. 449–458, Aug. 1984.

[94] J. Misra, "Distributed discrete-event simulation," *ACM Computing Surveys*, vol. 18, pp. 39–65, Mar. 1986.

[95] D. Reed, A. Malony, and B. McCredie, "Parallel discrete event simulation using shared memory," *IEEE Transactions on Software Engineering*, vol. 14, pp. 541–553, Apr. 1988.

[96] R. Chamberlain and M. Franklin, "Hierarchical discrete-event simulation on hypercube architectures," *IEEE Micro*, pp. 10–20, Aug. 1990.

[97] M. Davoren, "A structural mapping for parallel digital logic simulation," in *Proceedings of the SCS Multiconference on Distributed Simulation*, pp. 179–182, Mar. 1989.

[98] W. Hahn, H. Anger, A. Hagerer, and B. Schuster, "A multi-transputer-net as a hardware test bed," in *Proceedings of the European Simulation Multiconference*, pp. 359–364, June 1990.

[99] Q. Yu, D. Towsley, and P. Heidelberger, "Time-driven parallel simulation of multistage interconnection networks," in *Distributed simulation*, pp. 191–196, 1989.

[100] P. Goli, P. Heidelberger, D. Towsley, and Q. Yu, "Processor assignment and synchronisation in parallel simulation of multistage interconnection networks," Tech. Rep. RC 14814, IBM Research Division, 1989.

[101] J. Kowalik, ed., *Parallel MIMD Computation: HEP Supercomputer and its Applications.* Cambridge, Mass.: MIT Press, 1985.

[102] S. Butner and C. Staley, "A RISC Multiprocessor based on Circulating Context," in *IEEE Phoenix Conference on Computers and Communications*, Mar. 1986.

[103] L. Cohn, *A conceptual approach to general purpose parallel computer architecture.* PhD thesis, Columbia University, NY, 1983.

[104] H. Sullivan, T. Bashkow, and D. Klappholz, "A large scale homogeneous fully distributed parallel machine," in *Proceedings of the International Symposium on Computer Architecture*, 1977.

[105] R. Rettberg, W. Crowther, P. Carvey, and R. Tomlinson, "The Monarch parallel processor hardware design," *IEEE Computer*, pp. 18–30, Apr. 1990.

[106] A. Gottlieb, *Experimental parallel computing architectures*, ch. An overview of the NYU Ultracomputer project, pp. 25–95. North-Holland, 1987.

[107] G. Pfister *et al.*, "The IBM RP3: Introduction and Architecture," in *Proceedings of the International Conference on Parallel Processing*, pp. 764–771, Aug. 1986.

[108] T. Lees, *Context flow architecture.* PhD thesis, University of Edinburgh, 1990.

[109] P. Kogge, "The Microprogramming of Pipelined Processors," in *Proceedings of the International Symposium on Computer Architecture*, pp. 63–69, Mar. 1977.

[110] J. Emer and E. Davidson, "Control Store Organisation for Multiple Stream Pipelined Processors," in *Proceedings of the International Conference on Parallel Processing*, pp. 43–48, Aug. 1978.

[111] T. Lees, "Context streams - A theoretical basis for a generic form of MIMD pipelining," in *Proceedings 2nd IEEE symposium on parallel and distributed processing*, Dec. 1990.

[112] D. Patterson and C. Sequin, "A VLSI RISC," *IEEE Computer*, pp. 8–20, Sept. 1982.

[113] D. Dias and J. Jump, "Packet Communication in Multistage Shuffle-Exchange Networks," in *Proceedings of the International Conference on Parallel Processing*, pp. 327–328, Aug. 1980.

[114] L. Bhuyan and D. Agrawal, "Design and performance of generalised interconnection networks," *IEEE Transactions on Computers*, vol. 32, pp. 1081–1090, Dec. 1983.

[115] Inmos, *Occam 2 Reference Manual.* Prentice Hall, 1988.

[116] C. Hoare, "Communicating Sequential Processes," *Communications of the ACM*, vol. 21, pp. 666–677, Aug. 1978.

[117] D. May and R. Shepherd, "The transputer implementation of OCCAM," Technical Note 21, Inmos Ltd., Bristol, UK.

[118] Inmos, "IMS T800 Architecture," Tech. Rep. 6, Inmos Ltd., 1986.

[119] K. Bowler, R. Kenway, and D. Wallace, "The Edinburgh Concurrent Supercomputer: project and applications," in *Proceedings IEE conference on the design and application of parallel digital processors*, Apr. 1988.

[120] A. Hey, "Supercomputing with transputers - past, present and future," in *Proceedings of the International Conference on Supercomputing*, June 1990.

[121] N. de Bruijn, "A combinatorial problem," in *Proceedings Akademe van Wetenschappen*, vol. 49, pp. 758–764, 1946.

[122] F. Baude, F. Carre, P. Clere, and G. Vidal-Naquet, "Topologies for large transputer networks: theoretical aspects and experimental approach," in *Proceedings 10th OUG Technical Meeting*, pp. 178–197, Apr. 1989.

[123] D. Prior, M. Norman, N. Radcliffe, and L. Clarke, "What price regularity?," *Concurrency: Practice and Experience*, vol. 2, pp. 55–78, Mar. 1990.

[124] L. Clarke and G. Wilson, "Tiny: An efficient routing harness for the INMOS Transputer," Tech. Rep., Edinburgh Parallel Computing Centre, Feb. 1990. to be published in Concurrency: Practice and Experience.

[125] M. Quinn, *Designing efficient algorithms for parallel computers.* McGraw-Hill, 1987.

[126] A. Gibbons and W. Rytter, *Efficient parallel algorithms.* Cambridge University Press, 1988.

[127] S. Akl, *The design and analysis of parallel algorithms.* Prentice-Hall, 1989.

[128] N. Deo, C. Pang, and R. Lord, "Two parallel algorithms for shortest path problems," in *Proceedings of the International Conference on Parallel Processing*, pp. 244–253, Aug. 1980.

[129] F. Darema, D. George, V. Norton, and G. Pfister, "A single-program-multiple-data computational model for EPEX/Fortran," *Parallel computing*, vol. 7, pp. 11–24, 1988.

[130] K. Batcher, "Sorting Networks and their application," in *Proceedings AFIPS Conference 1968*, pp. 307–314, 1968.

[131] P. Heidelberger, A. Norton, and J. Robinson, "Parallel quicksort using fetch-and-add," *IEEE Transactions on Computers*, vol. 39, pp. 133–139, Jan. 1990.

[132] E. Moore, "The shortest path through a maze," in *Proceedings of the 1957 International Symposium on the Theory of Switching*, vol. 2, pp. 285–292, 1959.

[133] S. Abraham and K. Padmanabhan, "Performance of the direct binary n-cube network for multiprocessors," *IEEE Transactions on Computers*, vol. 38, pp. 1000–1011, July 1989.

[134] P. Thanish and M. Norman, "Minimising message path length in multicomputers," Tech. Rep. EPCC-TR90-17, Edinburgh Parallel Processing Centre, 1990.

[135] C. Schlatter-Ellis and T. Olson, "Algorithms for parallel memory allocation," *International Journal of Parallel Programming*, vol. 17, Aug. 1988.

[136] A. Smith, "Cache Memories," *ACM Computing Surveys*, vol. 14, pp. 473–530, Sept. 1982.

[137] L. Bhuyan, B. Liu, and i. Ahmed, "Analysis of MIN based multiprocessors with private cache memories," in *Proceedings of the International Conference on Parallel Processing*, vol. 1, pp. 51–58, Aug. 1989.

[138] P. Teller, R. Kenner, and M. Snir, "TLB consistency on highly-parallel shared-memory multiprocessors," in *Proceedings 21st International conference on system sciences*, pp. 184–193, Jan. 1988.

[139] B. Rosenburg, "Low-synchronisation translation lookaside buffer consistency in large-scale shared-memory multiprocessors," Tech. Rep. RC 14499, IBM Research division, 1989.

[140] G. Pfister *et al.*, *Experimental parallel computing architectures*, ch. An introduction to the IBM Research Parallel Processor Prototype (RP3), pp. 123–140. Elsevier Science Publishers, 1987.

Appendix A

Assembly code

This appendix contains example Eppl assembly code for the three benchmark programs. Each line in the assembly program has the format:

```
label     instruction     operand(s)
```

The label is optional, and the instruction is one of the Eppl assembly code instructions or an assembler directive. Multiple operands are seperated by commas. The Eppl assembly code instructions have been described in Section 3.1.6, and in addition there are a number of assembler directives:

source include assembly code from given file.

org start assembly from given address.

reg define a register name.

equ define a constant name.

dcl assemble a long constant.

res reserve a block of memory and initialise.

* indicates a comment follows.

In the assembly code listings that follow the register names `r0,...,r13,pc` and `sr` have been predefined and are read in from the file `registers.asm`.

A.1 mmult.asm

```
1   *
2   * Parallel matrix multiply program
3   *
4   * multiplies lxm matrix A with mxn matrix B
5   * to give lxn result matrix C
6   *
7   * i is the row of result matrix
8   * j is the column
9   *
10  * All the elements of C are calculated in parallel
11  *
12          source 'registers.asm'
13  elem    equ 15                  * (ln)-1
14  l       equ 4                   * l
15  n       equ 4                   * n
16  m       equ 4                   * m
17
18          org 16                  * reset address
19
20  start   ldi r1,pc,1             * load semaphore p address
21          dcl p
22          f&aq r0,r1,0            * semaphore
23          cmpq r0,0
24          blt start               * retry
25          f&aq r0,r1,-1           * fdec semaphore
26          cmpq r0,0
27          bge begin               * do mmult
28          f&aq r0,r1,1            * finc semaphore
29          bra start
30          begin   ldq r1,n        * load n
31          div r2,r0,r1            * i=p/n
32          rem r3,r0,r1            * j=p\n
33          ldq r4,0                * k=0
34          ldq r5,0                * t=0
35          ldq r6,l                * load l
36          ldq r7,m                * load m
37          ldi r8,pc,1             * load address a
38          dcl a
39          ldi r9,pc,1             * load address b
40          dcl b
```

```
41          ldi r10,pc,1        * load address c
42          dcl c
43  loop    mul r12,r2,r7       * load a(i*m+k)
44          add r12,r12,r4
45          add r12,r12,r8
46          ldo r12,r12,0
47          mul r13,r4,r1       * load b(k*n+j)
48          add r13,r13,r3
49          add r13,r13,r9
50          ldo r13,r13,0
51          mul r12,r12,r13     * multiply a x b
52          add r5,r5,r12       * t = t + (a x b)
53          addq r4,r4,1        * k = k + 1
54          cmp r4,r7           * k < m ?
55          blt loop
56          mul r12,r2,r1       * store c(i*n+j)
57          add r12,r12,r3
58          add r12,r12,r10
59          sto r12,r5,0
60          ldi r0,pc,1         * done semaphore
61          dcl d
62          f&aq r1,r0,-1
63          cmpq r1,0           * not last one
64          bne start
65          ldq r0,3            * send interrupt
66          sto r0,r0,0
67          bra start           * back to start
68  p       dcl elem            * scheduling count
69  d       dcl elem            * termination count
70  a       source 'array'
71  b       source 'array'
72  c       res 0,0
```

A.2 bitonic.asm

```
1   *
2   * Parallel Batcher Bitonic Merge Sort Program
3   *
4   * Sorts n numbers using n/2 processes.
5   *
6             source 'registers.asm'
7   nn        equ 16          * numbers to sort
8   np        equ 8           * number of processes (nn/2)
9   ns        equ 4           * number of stages (log2 nn)
10  *
11  * Start
12  *
13          org 16
14  reset   ldi r0,pc,1       * data address
15          dcl data
16  bitl0   f&aq r1,r0,0      * k
17          cmpq r1,np        * k>=np?
18          bge bitl0
19          f&aq r1,r0,1      * finc k
20          cmpq r1,np        * k<np?
21          blt bitl1
22          f&aq r1,r0,-1     * fdec k
23          bra bitl0
24  bitl1   addq r2,r0,2      * i address
25          ldo r2,r2,0       * read i
26          mulq r3,r1,2      * 2p
27          addq r4,r3,1      * 2p+1
28          ldq r5,2          * m=(2<<i)-1
29          asl r5,r5,r2
30          subq r5,r5,1
31          not r6,r5         * ~m
32          and r6,r6,r3      * x/\~m
33          and r7,r3,r5      * x/\m
34          asrq r7,r7,1      * >>1
35          ldq r8,1
36          and r8,r3,r8      * x/\1
37          asl r8,r8,r2      * <<s
38          or r9,r7,r8       * \/
39          or r9,r9,r6       * \/
40          not r6,r5         * ~m
```

```
41          and r6,r6,r4     * x/\~m
42          and r7,r4,r5     * x/\m
43          asrq r7,r7,1     * >>1
44          ldq r8,1
45          and r8,r4,r8     * x/\1
46          asl r8,r8,r2     * <<s
47          or r10,r7,r8     * \/
48          or r10,r10,r6    * \/
49          ldo r11,r0,4     * from address
50          add r12,r11,r9   * a address
51          add r13,r11,r10  * b address
52          ldo r5,r12,0     * read a
53          ldo r6,r13,0     * read b
54          ldq r7,1         * find sign
55          asl r7,r7,r2
56          div r7,r1,r7
57          remq r7,r7,2
58          ldo r11,r0,5     * to address
59          add r12,r11,r3   * a address
60          add r13,r11,r4   * b address
61          cmpq r7,0        * sign=0 ?
62          bne bitl2
63          cmp r5,r6        * r5>r6 ?
64          bgt bitl4
65          bra bitl3
66  bitl2   cmp r5,r6        * r5<r6 ?
67          ble bitl4
68  bitl3   sto r12,r5,0     * write a
69          sto r13,r6,0     * write b
70          bra bitl5
71  bitl4   sto r12,r6,0     * write b
72          sto r13,r5,0     * write a
73  bitl5   addq r1,r0,1     * d address
74          f&aq r1,r1,1     * finc d
75          addq r1,r1,1     * d+1
76          cmpq r1,np       * d=np ?
77          bne bitl0
78          ldo r1,r0,2      * read i
79          ldo r2,r0,3      * read j
80          addq r2,r2,1     * j++
81          cmp r2,r1        * j<=i ?
82          ble bitl6
83          ldq r2,0         * j=0
```

```
84          addq r1,r1,1    * i++
85          cmpq r1,ns      * i=ns ?
86          beq bitl7
87          sto r0,r1,2     * store i
88  bitl6   sto r0,r2,3     * store j
89          ldo r1,r0,4     * read from
90          ldo r2,r0,5     * read to
91          sto r0,r1,5     * store from
92          sto r0,r2,4     * store to
93          ldq r2,0        * -np
94          subq r2,r2,np
95          addq r1,r0,1    * d address
96          f&a r1,r1,r2    * d=0
97          f&a r1,r0,r2    * k=0
98          bra bitl0
99  bitl7   ldq r0,3        * done
100         stq r0,0
101         bra reset
102 *
103 * Data
104 *
105 data    dcl 0           * k
106         dcl 0           * d
107         dcl 0           * i
108         dcl 0           * j
109         dcl from        * from
110         dcl to          * to
111 from    dcl 15,14,13,12,11,10,9,8,7,6,5,4,3,2,1,0
112 to      res 16,0
```

A.3 moore.asm

```
1   *
2   * Parallel Moore Shortest Path
3   *
4           source 'registers.asm'
5   nn      equ 64              * no nodes
6   nnm1    equ 63              * (nnode-1)
7   dd      equ 8               * degree
8   es      equ 16              * degree*2
9   s       equ nn              * queue size
10  sm1     equ nnm1            * (size-1)
11  empty   equ -1              * empty
12  *
13  * Code
14  *
15          org 16
16  reset   ldi r3,pc,1         * d address
17          dcl d
18          ldi r4,pc,1         * dis address
19          dcl dis
20          ldi r5,pc,1         * qc address
21          dcl qc
22          ldi r6,pc,1         * iqs address
23          dcl iqs
24          ldi r7,pc,1         * ps address
25          dcl ps
26  wait1   f&aq r8,r5,0        * Qc
27          cmpq r8,0           * Qc<=0?
28          ble wait1
29          f&aq r8,r5,-1       * fdec Qc
30          cmpq r8,0           * Qc>0?
31          bgt cont1
32          f&aq r8,r5,1        * finc Qc
33          bra wait1
34  cont1   addq r8,r5,2        * Qd address
35          f&aq r8,r8,1        * finc Qd
36          remq r8,r8,s        * Qd\Qs
37          addq r9,r5,3        * Q address
38          add r9,r9,r8        * offset
39  wait2   ldo r0,r9,0         * get node
40          cmpq r0,empty       * r0=empty?
```

```
41          beq wait2
42          stq r9,empty      * set empty
43          add r9,r6,r0      * iqsem offset
44          f&aq r9,r9,-1     * fdec iqsem
45          f&aq r9,r7,1      * finc ps
46          ldq r1,0          * i
47          ldq r2,0          * v
48  loop    ldi r8,pc,1       * weights address
49          dcl w
50          mulq r9,r0,es     * scale u*es
51          add r8,r8,r9      * base
52          mulq r9,r1,2      * scale i*2
53          add r8,r8,r9      * offset
54          ldo r2,r8,0       * node v
55          ldo r8,r8,1       * weight
56          add r9,r3,r0      * d offset
57          ldo r9,r9,0       * distance[u]
58          add r8,r8,r9      * new_distance=distance[u]+weight[u][v]
59          add r9,r4,r2      * disem offset
60  wait3   f&aq r10,r9,0     * P(disem[v])
61          cmpq r10,0        * disem<=0?
62          ble wait3
63          f&aq r10,r9,-1    * fdec disem
64          cmpq r10,0        * disem>0?
65          bgt cont2
66          f&aq r10,r9,1     * finc disem
67          bra wait3
68  cont2   add r9,r3,r2      * d offset
69          ldo r9,r9,0       * distance[v]
70          cmp r8,r9         * new_distance<distance[v]?
71          blt write
72          add r9,r4,r2      * disem offset
73          f&aq r10,r9,1     * V(disem[v])
74          bra cont3
75  write   add r9,r3,r2      * d offset
76          sto r9,r8,0       * store new_distance[v]
77          add r9,r4,r2      * disem offset
78          f&aq r10,r9,1     * V(disem[v])
79          add r9,r6,r2      * iqsem offset
80          f&aq r10,r9,1     * finc iqsem[v]
81          cmpq r10,0        * iqsem[v]=0?
82          beq queue
83          f&aq r10,r9,-1    * fdec iqsem[v]
```

```
84          bra cont3
85  queue   addq r9,r7,1      * ws address
86          f&aq r10,r9,1     * finc ws
87          addq r8,r5,1      * Qi address
88          f&aq r8,r8,1      * finc Qi
89          remq r8,r8,s      * Qi\Qs
90          addq r9,r5,3      * Q address
91          add r9,r9,r8      * offset
92  wait4   ldo r10,r9,0      * get node
93          cmpq r10,empty    * node=empty?
94          bne wait4
95          sto r9,r2,0       * put node (v)
96          f&aq r10,r5,1     * finc Qc
97  cont3   addq r1,r1,1      * i=i+1
98          cmpq r1,dd        * i<dd?
99          blt loop
100         f&aq r8,r7,-1     * fdec ps
101         addq r8,r7,1      * ws address
102         f&aq r8,r8,-1     * fdec ws
103         cmpq r8,1         * ws=1?
104         bgt reset
105         ldq r8,3          * interrupt
106         stq r8,0
107         bra reset
108 *
109 * Data
110 *
111 ps      dcl 0             * process count
112 ws      dcl 1             * termination count
113 d       dcl 0             * distance
114         res nnm1,inf
115 dis     res nn,1          * distance semaphores
116 iqs     dcl 1             * inqueue semaphores
117         res nnm1,0
118 qc      dcl 1             * Qc count
119 qi      dcl 1             * Qi pointer
120 qd      dcl 0             * Qd pointer
121         dcl 0             * buffer space
122         res sm1,empty
123 w       source 'weights.asm'
```

Appendix B

Detailed execution statistics

This appendix contains the tables of detailed execution statistics from the simulation experiments described in Chapter 6. The table headings are abbreviated as shown below in Table B.1. Statistics which were not collected for a particular simulation are marked with a dash.

key	meaning	key	meaning
prcrs	processors	mreqs	memory requests
prcs	interleaving level	mlat	memory latency
k-way	combining level	rqcon	request queue conflicts
qu	queue size	wbfull	wait-buffer full events
wb	wait-buffer size	aqcon	acknowledge queue conflicts
clks	clock cycles	ldcom	load combinations
putil	processor utilisation	f&acom	f&a combinations
insts	instructions	mutil	memory utilisation
nutil	network utilisation	maccs	memory accesses

Table B.1: Description of abbreviated table headings

prcrs	clks	putil	insts	nutil	mreqs	mlat	rqcon	wbfull	aqcon	ldcom	f&acom	mutil	maccs
1	4081780	0.86	3517180	1.00	4081790	2.0	0	0	0	0	0	1.00	4081790
2	2045150	0.86	3522850	1.00	4090320	5.0	0	0	0	1907930	4102	0.53	2178290
4	1023170	0.86	3524430	1.00	4092700	7.0	0	0	0	2863680	6144	0.30	1222870
8	559190	0.79	3521410	0.91	4088220	8.0	0	0	0	3337040	7168	0.17	744001
16	308610	0.71	3521270	0.83	4088140	10.1	0	0	0	3575320	7680	0.10	505133
32	166241	0.66	3511300	0.77	4073570	12.1	0	0	0	3680380	7936	0.07	385219
64	91108	0.60	3512320	0.70	4075520	14.0	0	0	0	3741680	8064	0.06	325759
128	49569	0.55	3514370	0.64	4079620	16.0	0	0	0	3904420	8128	0.03	167005
256	26814	0.51	3518460	0.60	4087810	18.0	0	119420	0	3991770	8160	0.01	87627

Table B.2: Execution statistics for matrix multiply (mm4096) with increasing system size (interleaving level 16, 2-way combining, queue size 4, wait-buffer size 16, equal module clock factors).

prcrs	clks	putil	insts	nutil	mreqs	mlat	rqcon	wbfull	aqcon	ldcom	f&acom	mutil	maccs
1	24233200	0.85	20485100	1.00	24233200	2.0	0	0	0	0	0	1.00	24233200
2	12518300	0.82	20547700	0.97	24316700	6.0	591893	0	0	1108170	110752	0.92	23097800
4	6901930	0.75	20672700	0.89	24483500	8.8	338421	0	4723	2192760	212436	0.80	22078300
8	3817530	0.69	20960300	0.81	24866900	8.8	84346	0	6164	4566940	395166	0.65	19904800
16	2081840	0.64	21299200	0.76	25318900	10.9	18561	0	4443	8851630	734241	0.47	15733000
32	1140610	0.59	21707000	0.71	25863100	12.7	7423	0	3338	14224200	1133960	0.29	10504900
64	644810	0.55	22585200	0.66	27034700	14.8	6304	0	4098	19029300	1578160	0.16	6427120
128	382218	0.50	24547100	0.61	29652200	16.3	9647	19	8355	23325800	2320130	0.08	4006090
256	241301	0.46	28444900	0.56	34852500	18.1	22767	16019	16715	28448200	3667270	0.04	2736490

Table B.3: Execution statistics for bitonic (bit4096) with increasing system size (interleaving level 16, 2-way combining, queue size 4, wait-buffer size 16, equal module clock factors).

prcrs	clks	putil	insts	nutil	mreqs	mlat	rqcon	wbfull	aqcon	ldcom	f&acom	mutil	maccs
1	3892020	0.75	2937690	1.00	3892030	2.0	0	0	0	0	0	1.00	3892030
2	2070030	0.72	2988150	0.96	3959200	6.0	164330	0	0	146256	3683	0.92	3809260
4	1165230	0.66	3086280	0.88	4089890	7.8	155218	0	1437	267092	10144	0.82	3812640
8	656641	0.60	3169910	0.80	4202410	9.6	71459	0	2623	426888	26096	0.71	3749410
16	386187	0.55	3424790	0.74	4544890	11.4	33312	0	2846	736636	72547	0.60	3735680
32	236012	0.51	3876310	0.68	5155250	12.4	17873	0	2706	1324490	193721	0.48	3636980
64	173839	0.48	5322490	0.64	7106960	14.5	11131	0	3446	2815060	584756	0.33	3707020
128	146333	0.44	8319480	0.60	11172200	17.0	7352	27	5998	5968400	1532870	0.20	3670690
256	132597	0.41	13831100	0.55	18669400	19.9	6149	26989	25233	11495600	3261970	0.12	3911310

Table B.4: Execution statistics for Moore (moo4096) with increasing system size (interleaving level 16, 2-way combining, queue size 4, wait-buffer size 16, equal module clock factors).

qu	wb	clks	putil	insts	nutil	mreqs	mlat	rqcon	wbfull	aqcon	ldcom	f&acom	mutil	maccs
32	32	1587	0.81	2573	1.00	3194	4.0	0	0	0	1476	32	0.53	1684
16	32	1587	0.81	2573	1.00	3194	4.0	0	0	0	1476	32	0.53	1684
8	32	1587	0.81	2573	1.00	3194	4.0	0	0	0	1476	32	0.53	1684
4	32	1587	0.81	2573	1.00	3194	4.0	0	0	0	1476	32	0.53	1684
2	32	1587	0.81	2573	1.00	3194	4.0	0	0	0	1476	32	0.53	1684
32	16	1587	0.81	2573	1.00	3194	4.0	0	0	0	1476	32	0.53	1684
16	16	1587	0.81	2573	1.00	3194	4.0	0	0	0	1476	32	0.53	1684
8	16	1587	0.81	2573	1.00	3194	4.0	0	0	0	1476	32	0.53	1684
4	16	1587	0.81	2573	1.00	3194	4.0	0	0	0	1476	32	0.53	1684
2	16	1587	0.81	2573	1.00	3194	4.0	0	0	0	1476	32	0.53	1684
32	8	1587	0.81	2573	1.00	3194	4.0	0	0	0	1476	32	0.53	1684
16	8	1587	0.81	2573	1.00	3194	4.0	0	0	0	1476	32	0.53	1684
8	8	1587	0.81	2573	1.00	3194	4.0	0	0	0	1476	32	0.53	1684
4	8	1587	0.81	2573	1.00	3194	4.0	0	0	0	1476	32	0.53	1684
2	8	1587	0.81	2573	1.00	3194	4.0	0	0	0	1476	32	0.53	1684
32	4	1587	0.81	2573	1.00	3194	4.0	0	0	0	1476	32	0.53	1684
16	4	1587	0.81	2573	1.00	3194	4.0	0	0	0	1476	32	0.53	1684
8	4	1587	0.81	2573	1.00	3194	4.0	0	0	0	1476	32	0.53	1684
4	4	1587	0.81	2573	1.00	3194	4.0	0	0	0	1476	32	0.53	1684
2	4	1587	0.81	2573	1.00	3194	4.0	0	0	0	1476	32	0.53	1684
32	2	1587	0.81	2573	1.00	3194	4.0	0	595	0	1476	32	0.53	1684
16	2	1587	0.81	2573	1.00	3194	4.0	0	595	0	1476	32	0.53	1684
8	2	1587	0.81	2573	1.00	3194	4.0	0	595	0	1476	32	0.53	1684
4	2	1587	0.81	2573	1.00	3194	4.0	0	595	0	1476	32	0.53	1684
2	2	1587	0.81	2573	1.00	3194	4.0	0	595	0	1476	32	0.53	1684

Table B.5: Execution statistics for matrix multiply (mm32) on 2 processors with varying queue and wait-buffer size (interleaving level 16, 2-way combining, equal module clock factors).

qu	wb	clks	putil	insts	nutil	mreqs	mlat	rqcon	wbfull	aqcon	ldcom	f&acom	mutil	maccs
32	32	2557	0.83	-	1.00	-	7.5	0	0	0	7154	96	0.29	-
16	32	2557	0.83	-	1.00	-	7.5	0	0	0	7156	96	0.29	-
8	32	2557	0.83	-	1.00	-	7.5	0	0	0	7154	96	0.29	-
4	32	2557	0.83	-	1.00	-	7.5	0	0	0	7154	96	0.29	-
2	32	2562	0.83	-	1.00	-	7.5	8	0	11	7143	96	0.29	-
32	16	2557	0.83	-	1.00	-	7.5	0	0	0	7154	96	0.29	-
16	16	2557	0.83	-	1.00	-	7.5	0	0	0	7154	96	0.29	-
8	16	2557	0.83	-	1.00	-	7.5	0	0	0	7154	96	0.29	-
4	16	2557	0.83	-	1.00	-	7.5	0	0	0	7154	96	0.29	-
2	16	2562	0.83	-	1.00	-	7.5	8	0	11	7143	96	0.29	-
32	8	2557	0.83	-	1.00	-	7.5	0	0	0	7154	96	0.29	-
16	8	2557	0.83	-	1.00	-	7.5	0	0	0	7154	96	0.29	-
8	8	2557	0.83	-	1.00	-	7.5	0	0	0	7154	96	0.29	-
4	8	2557	0.83	-	1.00	-	7.5	0	0	0	7154	96	0.29	-
2	8	2562	0.83	-	1.00	-	7.5	8	0	11	7143	96	0.29	-
32	4	2557	0.83	-	1.00	-	7.0	0	512	0	7147	96	0.29	-
16	4	2557	0.83	-	1.00	-	7.0	0	512	0	7147	96	0.29	-
8	4	2557	0.83	-	1.00	-	7.0	0	512	0	7147	96	0.29	-
4	4	2557	0.83	-	1.00	-	7.0	0	512	0	7147	96	0.29	-
2	4	2562	0.83	-	1.00	-	7.5	15	504	11	7134	96	0.30	-
32	2	2786	0.77	-	0.93	-	9.5	0	1317	0	3458	56	0.61	-
16	2	2786	0.77	-	0.93	-	9.5	0	1317	0	3458	56	0.61	-
8	2	2804	0.76	-	0.92	-	8.8	10	1247	0	3256	57	0.63	-
4	2	2783	0.77	-	0.93	-	8.2	135	1535	3	3568	57	0.60	-
2	2	3140	0.69	-	0.84	-	9.2	2051	1315	1348	2965	55	0.60	-

Table B.6: Execution statistics for matrix multiply (mm64) on 4 processors with varying queue and wait-buffer size (interleaving level 16, 2-way combining, equal module clock factors).

qu	wb	clks	putil	insts	nutil	mreqs	mlat	rqcon	wbfull	aqcon	ldcom	f&acom	mutil	maccs
32	32	2756	0.77	-	0.93	-	8.2	0	0	0	16651	224	0.16	-
16	32	2756	0.77	-	0.93	-	8.2	0	0	0	16651	224	0.16	-
8	32	2756	0.77	-	0.93	-	8.2	0	0	0	16651	224	0.16	-
4	32	2756	0.77	-	0.93	-	8.2	0	0	0	16651	224	0.16	-
2	32	2762	0.76	-	0.92	-	8.5	16	0	3	16672	224	0.16	-
32	16	2756	0.77	-	0.93	-	8.2	0	0	0	16651	224	0.16	-
16	16	2756	0.77	-	0.93	-	8.2	0	0	0	16651	224	0.16	-
8	16	2756	0.77	-	0.93	-	8.2	0	0	0	16651	224	0.16	-
4	16	2756	0.77	-	0.93	-	8.2	0	0	0	16651	224	0.16	-
2	16	2762	0.76	-	0.92	-	8.5	16	0	3	16672	224	0.16	-
32	8	2756	0.77	-	0.93	-	8.2	0	0	0	16651	224	0.16	-
16	8	2756	0.77	-	0.93	-	8.2	0	0	0	16651	224	0.16	-
8	8	2756	0.77	-	0.93	-	8.2	0	0	0	16651	224	0.16	-
4	8	2756	0.77	-	0.93	-	8.2	0	0	0	16651	224	0.16	-
2	8	2762	0.76	-	0.92	-	8.5	16	0	3	16672	224	0.16	-
32	4	3125	0.69	-	0.83	-	10.2	0	1022	0	8933	157	0.47	-
16	4	3108	0.69	-	0.83	-	10.2	23	1042	0	8958	161	0.47	-
8	4	3089	0.69	-	0.84	-	9.8	82	1094	0	9658	169	0.44	-
4	4	3109	0.69	-	0.84	-	10.4	732	1200	25	9935	170	0.43	-
2	4	3409	0.64	-	0.77	-	11.2	5107	1263	2872	8283	144	0.46	-
32	2	3263	0.66	-	0.81	-	13.8	28	2833	0	7439	141	0.51	-
16	2	3242	0.66	-	0.80	-	13.5	193	2954	0	7577	142	0.51	-
8	2	3233	0.66	-	0.81	-	13.2	659	3205	0	7990	144	0.49	-
4	2	3266	0.66	-	0.80	-	12.0	1710	3428	15	7902	143	0.50	-
2	2	3584	0.61	-	0.74	-	12.5	6972	3245	3762	7145	120	0.48	-

Table B.7: Execution statistics for matrix multiply (mm128) on 8 processors with varying queue and wait-buffer size (interleaving level 16, 2-way combining, equal module clock factors).

qu	wb	clks	putil	insts	nutil	mreqs	mlat	rqcon	wbfull	aqcon	ldcom	f&acom	mutil	maccs
32	32	5348	0.71	-	0.84	-	10.1	0	0	0	62762	480	0.10	-
16	32	5348	0.71	-	0.84	-	10.1	0	0	0	62762	480	0.10	-
8	32	5348	0.71	-	0.84	-	10.1	0	0	0	62762	480	0.10	-
4	32	5348	0.71	-	0.84	-	10.1	0	0	0	62762	480	0.10	-
2	32	5348	0.71	-	0.84	-	10.1	1	0	0	62768	480	0.10	-
32	16	5348	0.71	-	0.84	-	10.1	0	0	0	62768	480	0.10	-
16	16	5348	0.71	-	0.84	-	10.1	0	0	0	62768	480	0.10	-
8	16	5348	0.71	-	0.84	-	10.1	0	0	0	62762	480	0.10	-
4	16	5348	0.71	-	0.84	-	10.1	0	0	0	62762	480	0.10	-
2	16	5348	0.71	-	0.84	-	10.1	1	0	0	62768	480	0.10	-
32	8	5348	0.71	-	0.84	-	10.1	0	1631	0	62762	480	0.10	-
16	8	5348	0.71	-	0.84	-	10.1	0	1631	0	62762	480	0.10	-
8	8	5348	0.71	-	0.84	-	10.1	0	1631	0	62762	480	0.10	-
4	8	5348	0.71	-	0.84	-	10.1	0	1631	0	62762	480	0.10	-
2	8	5348	0.71	-	0.84	-	10.1	1	1631	0	62768	480	0.10	-
32	4	6163	0.62	-	0.74	-	12.9	1	2726	0	32339	328	0.41	-
16	4	6198	0.62	-	0.74	-	12.6	138	2722	0	32704	331	0.41	-
8	4	6130	0.63	-	0.74	-	12.8	550	2997	0	33711	335	0.40	-
4	4	6198	0.62	-	0.74	-	12.4	5015	3296	111	33781	331	0.39	-
2	4	6882	0.57	-	0.68	-	14.1	25123	3575	12955	30098	303	0.41	-
32	2	6358	0.60	-	0.72	-	20.6	277	8201	0	26962	281	0.45	-
16	2	6384	0.60	-	0.72	-	26.5	888	9198	0	28161	279	0.44	-
8	2	6363	0.60	-	0.72	-	23.1	2834	9897	0	28954	285	0.43	-
4	2	6378	0.60	-	0.72	-	16.9	8281	10478	73	29184	297	0.43	-
2	2	6994	0.56	-	0.67	-	15.3	29444	8874	13864	26625	250	0.43	-

Table B.8: Execution statistics for matrix multiply (mm256) on 16 processors with varying queue and wait-buffer size (interleaving level 16, 2-way combining, equal module clock factors).

qu	wb	clks	putil	insts	nutil	mreqs	mlat	rqcon	wbfull	aqcon	ldcom	f&acom	mutil	maccs
32	32	5765	0.65	-	0.77	-	12.1	0	0	0	132363	992	0.05	-
16	32	5765	0.65	-	0.77	-	12.1	0	0	0	132363	992	0.05	-
8	32	5765	0.65	-	0.77	-	12.1	0	0	0	132363	992	0.05	-
4	32	5765	0.65	-	0.77	-	12.1	0	0	0	132363	992	0.05	-
2	32	5765	0.65	-	0.77	-	12.1	0	0	0	132363	992	0.05	-
32	16	5765	0.65	-	0.77	-	12.1	0	0	0	132363	992	0.05	-
16	16	5765	0.65	-	0.77	-	12.1	0	0	0	132363	992	0.05	-
8	16	5765	0.65	-	0.77	-	12.1	0	0	0	132363	992	0.05	-
4	16	5765	0.65	-	0.77	-	12.1	0	0	0	132363	992	0.05	-
2	16	5765	0.65	-	0.77	-	12.1	0	0	0	132363	992	0.05	-
32	8	6631	0.58	-	0.69	-	13.8	0	5873	0	91771	748	0.26	-
16	8	6641	0.58	-	0.69	-	13.6	145	5947	0	92268	759	0.25	-
8	8	6605	0.58	-	0.69	-	14.2	636	5850	8	93443	770	0.25	-
4	8	6692	0.58	-	0.69	-	13.7	9599	4961	139	89402	747	0.26	-
2	8	7348	0.53	-	0.64	-	17.0	48118	4916	21136	80567	691	0.29	-
32	4	6810	0.57	-	0.67	-	18.3	131	7213	0	78196	718	0.31	-
16	4	6795	0.57	-	0.67	-	16.4	617	7360	0	79025	720	0.31	-
8	4	6818	0.56	-	0.67	-	15.5	2797	8152	6	81493	695	0.30	-
4	4	6885	0.56	-	0.67	-	14.9	12710	7583	181	77127	708	0.32	-
2	4	7549	0.52	-	0.63	-	17.1	59760	7375	21261	75026	637	0.31	-
32	2	7363	0.52	-	0.62	-	29.6	2856	24560	0	70110	574	0.32	-
16	2	7235	0.53	-	0.63	-	32.0	6442	26283	0	70998	601	0.32	-
8	2	7247	0.53	-	0.64	-	29.1	14659	28558	0	72856	614	0.32	-
4	2	7207	0.54	-	0.64	-	21.4	25691	29234	235	73248	616	0.32	-
2	2	7886	0.51	-	0.61	-	19.4	79382	28886	22279	73112	554	0.32	-

Table B.9: Execution statistics for matrix multiply (mm512) on 32 processors with varying queue and wait-buffer size (interleaving level 16, 2-way combining, equal module clock factors).

qu	wb	clks	putil	insts	nutil	mreqs	mlat	rqcon	wbfull	aqcon	ldcom	f&acom	mutil	maccs
32	32	11820	0.60	-	0.70	-	14.0	0	0	0	502721	2016	0.03	-
16	32	11820	0.60	-	0.70	-	14.0	0	0	0	502721	2016	0.03	-
8	32	11820	0.60	-	0.70	-	14.0	0	0	0	502721	2016	0.03	-
4	32	11820	0.60	-	0.70	-	14.0	0	0	0	502721	2016	0.03	-
2	32	11820	0.60	-	0.70	-	14.0	0	0	0	502721	2016	0.03	-
32	16	11820	0.60	-	0.70	-	14.0	0	0	0	502721	2016	0.03	-
16	16	11820	0.60	-	0.70	-	14.0	0	0	0	502721	2016	0.03	-
8	16	11820	0.60	-	0.70	-	14.0	0	0	0	502721	2016	0.03	-
4	16	11820	0.60	-	0.70	-	14.0	0	0	0	502721	2016	0.03	-
2	16	11820	0.60	-	0.70	-	14.0	0	0	0	502721	2016	0.03	-
32	8	13970	0.52	-	0.61	-	15.7	1434	12481	0	346318	1528	0.22	-
16	8	13845	0.52	-	0.62	-	15.6	3595	13740	1	348102	1562	0.22	-
8	8	13756	0.53	-	0.62	-	15.6	7310	10622	55	351886	1556	0.22	-
4	8	14031	0.52	-	0.61	-	15.6	55700	7845	903	341305	1516	0.23	-
2	8	14954	0.49	-	0.58	-	19.9	190323	9378	68999	341967	1434	0.23	-
32	4	14825	0.49	-	0.58	-	24.2	11169	23939	0	328670	1364	0.23	-
16	4	14502	0.50	-	0.59	-	24.6	18048	26318	0	333109	1412	0.23	-
8	4	14296	0.51	-	0.60	-	22.5	32751	27410	235	336169	1405	0.23	-
4	4	14095	0.52	-	0.61	-	18.6	61797	23190	720	334012	1430	0.24	-
2	4	15122	0.49	-	0.58	-	21.6	217582	25321	68167	338765	1318	0.23	-
32	2	16822	0.44	-	0.52	-	35.3	40441	136797	0	308852	1018	0.23	-
16	2	16637	0.45	-	0.53	-	41.2	87485	144377	0	317192	1048	0.23	-
8	2	15726	0.47	-	0.56	-	35.9	127619	151706	161	323533	1124	0.23	-
4	2	15118	0.49	-	0.58	-	29.7	159489	156500	1896	327801	1217	0.24	-
2	2	16426	0.46	-	0.55	-	23.9	361583	166779	70310	340680	1086	0.23	-

Table B.10: Execution statistics for matrix multiply (mm1024) on 64 processors with varying queue and wait-buffer size (interleaving level 16, 2-way combining, equal module clock factors).

qu	wb	clks	putil	insts	nutil	mreqs	mlat	rqcon	wbfull	aqcon	ldcom	f&acom	mutil	maccs
32	32	12853	0.55	-	0.64	-	16.0	0	0	0	1029010	4064	0.02	-
16	32	12853	0.55	-	0.64	-	16.0	0	0	0	1029010	4064	0.02	-
8	32	12853	0.55	-	0.64	-	16.0	0	0	0	1029010	4064	0.02	-
4	32	12853	0.55	-	0.64	-	16.0	0	0	0	1029010	4064	0.02	-
2	32	12853	0.55	-	0.64	-	16.0	0	0	0	1029010	4064	0.02	-
32	16	12853	0.55	-	0.64	-	16.0	0	0	0	1029010	4064	0.02	-
16	16	12853	0.55	-	0.64	-	16.0	0	0	0	1029010	4064	0.02	-
8	16	12853	0.55	-	0.64	-	16.0	0	0	0	1029010	4064	0.02	-
4	16	12853	0.55	-	0.64	-	16.0	0	0	0	1029010	4064	0.02	-
2	16	12853	0.55	-	0.64	-	16.0	0	0	0	1029010	4064	0.02	-
32	8	15646	0.47	-	0.56	-	19.6	6902	28398	0	815166	3285	0.15	-
16	8	15271	0.48	-	0.56	-	19.1	13092	29336	0	810440	3366	0.15	-
8	8	15161	0.48	-	0.57	-	17.9	29329	24357	165	810993	3409	0.15	-
4	8	15257	0.48	-	0.56	-	18.0	117534	12013	2067	796322	3372	0.15	-
2	8	16472	0.45	-	0.54	-	21.8	449101	21642	123916	806928	3030	0.15	-
32	4	24077	0.33	-	0.40	-	29.5	232919	103556	0	825701	2367	0.14	-
16	4	20184	0.38	-	0.45	-	41.7	266527	128170	0	801553	2767	0.14	-
8	4	16084	0.46	-	0.54	-	26.6	106798	107569	144	790171	3119	0.15	-
4	4	15581	0.47	-	0.55	-	25.9	161920	100109	2401	788835	3196	0.16	-
2	4	16856	0.44	-	0.53	-	24.5	572088	111377	128644	805782	2925	0.15	-
32	2	34303	0.25	-	0.30	-	78.5	867180	389406	0	778009	1726	0.13	-
16	2	36203	0.25	-	0.31	-	86.2	1594850	523223	0	884299	1870	0.12	-
8	2	27670	0.30	-	0.37	-	70.8	1647390	568176	862	841812	2042	0.13	-
4	2	24580	0.34	-	0.41	-	49.0	1713910	654296	2413	888817	2421	0.13	-
2	2	20012	0.39	-	0.47	-	35.4	1431000	576956	130680	824515	2307	0.14	-

Table B.11: Execution statistics for matrix multiply (mm2048) on 128 processors with varying queue and wait-buffer size (interleaving level 16, 2-way combining, equal module clock factors).

qu	wb	clks	putil	insts	nutil	mreqs	mlat	rqcon	wbfull	aqcon	ldcom	f&acom	mutil	maccs
32	32	30989	0.81	50487	1.00	61860	6.0	0	0	0	11094	2029	0.79	48735
16	32	30989	0.81	50487	1.00	61860	6.0	0	0	0	11094	2029	0.79	48735
8	32	30983	0.81	50480	1.00	61851	5.5	4	0	0	11257	2035	0.78	48556
4	32	30995	0.81	50445	1.00	61804	6.0	130	0	0	11057	1999	0.79	48745
2	32	32207	0.78	50292	0.96	61602	5.0	2832	0	0	10548	1713	0.77	49340
32	16	30989	0.81	50487	1.00	61860	6.0	0	0	0	11094	2029	0.79	48735
16	16	30989	0.81	50487	1.00	61860	6.0	0	0	0	11094	2029	0.79	48735
8	16	30983	0.81	50480	1.00	61851	5.5	4	0	0	11257	2035	0.78	48556
4	16	30995	0.81	50445	1.00	61804	6.0	130	0	0	11057	1999	0.79	48745
2	16	32207	0.78	50292	0.96	61602	5.0	2832	0	0	10548	1713	0.77	49340
32	8	30989	0.81	50487	1.00	61860	6.0	0	0	0	11094	2029	0.79	48735
16	8	30989	0.81	50487	1.00	61860	6.0	0	0	0	11094	2029	0.79	48735
8	8	30983	0.81	50480	1.00	61851	5.5	4	0	0	11257	2035	0.78	48556
4	8	30995	0.81	50445	1.00	61804	6.0	130	0	0	11057	1999	0.79	48745
2	8	32207	0.78	50292	0.96	61602	5.0	2832	0	0	10548	1713	0.77	49340
32	4	30989	0.81	50487	1.00	61860	6.0	0	26	0	11094	2029	0.79	48735
16	4	30989	0.81	50487	1.00	61860	6.0	0	26	0	11094	2029	0.79	48735
8	4	30984	0.81	50482	1.00	61853	6.5	1	45	0	11525	2034	0.78	48291
4	4	30995	0.81	50445	1.00	61804	6.0	130	21	0	11057	1999	0.79	48745
2	4	32207	0.78	50292	0.96	61602	5.0	2832	0	0	10548	1713	0.77	49340
32	2	31035	0.81	50444	1.00	61804	5.5	0	4796	0	10625	2004	0.79	49172
16	2	31035	0.81	50444	1.00	61804	5.5	0	4796	0	10625	2004	0.79	49172
8	2	31107	0.81	50594	1.00	62002	6.5	4	4866	0	10836	1959	0.79	49204
4	2	31026	0.81	50357	0.99	61686	5.5	274	5103	0	11311	1932	0.78	48440
2	2	32540	0.78	50482	0.95	61847	5.0	3253	3574	0	10379	1669	0.76	49797

Table B.12: Execution statistics for bitonic (bit32) on 2 processors with varying queue and wait-buffer size (interleaving level 16, 2-way combining, equal module clock factors).

qu	wb	clks	putil	insts	nutil	mreqs	mlat	rqcon	wbfull	aqcon	ldcom	f&acom	mutil	maccs
32	32	43856	0.77	134967	0.94	165317	8.0	0	0	0	51011	8158	0.60	106140
16	32	43856	0.77	134967	0.94	165317	8.0	0	0	0	51011	8158	0.60	106140
8	32	43864	0.77	134990	0.94	165344	7.8	17	0	0	51065	8185	0.60	106086
4	32	43820	0.77	134712	0.94	164972	7.2	647	0	81	50712	8098	0.61	106154
2	32	48226	0.70	134883	0.86	165204	8.2	23845	0	29053	50588	9371	0.55	105237
32	16	43856	0.77	134967	0.94	165317	8.0	0	0	0	51011	8158	0.60	106140
16	16	43856	0.77	134967	0.94	165317	8.0	0	0	0	51011	8158	0.60	106140
8	16	43864	0.77	134990	0.94	165344	7.8	17	0	0	51065	8185	0.60	106086
4	16	43820	0.77	134712	0.94	164972	7.2	647	0	81	50712	8098	0.61	106154
2	16	48226	0.70	134883	0.86	165204	8.2	23845	0	29053	50588	9371	0.55	105237
32	8	43856	0.77	134967	0.94	165317	8.0	0	0	0	51011	8158	0.60	106140
16	8	43856	0.77	134967	0.94	165317	8.0	0	0	0	51011	8158	0.60	106140
8	8	43864	0.77	134990	0.94	165344	7.8	17	0	0	51065	8185	0.60	106086
4	8	43820	0.77	134712	0.94	164972	7.2	647	0	81	50712	8098	0.61	106154
2	8	48226	0.70	134883	0.86	165204	8.2	23845	1	29053	50588	9371	0.55	105237
32	4	43890	0.77	134905	0.94	165230	7.8	0	3588	0	50382	8166	0.61	106673
16	4	43890	0.77	134905	0.94	165230	7.8	0	3588	0	50382	8166	0.61	106673
8	4	43944	0.77	135045	0.94	165418	7.5	13	3577	0	50723	8170	0.61	106517
4	4	43858	0.77	134673	0.94	164922	7.5	690	3212	59	50217	8107	0.61	106591
2	4	48116	0.70	134581	0.86	164800	10.2	24269	5466	27771	50481	9180	0.55	105130
32	2	44888	0.75	134118	0.91	164185	9.2	0	20450	0	46598	7322	0.61	110253
16	2	44888	0.75	134118	0.91	164185	9.2	0	20450	0	46598	7322	0.61	110253
8	2	44729	0.75	133682	0.91	163604	9.0	63	20506	0	46638	7226	0.61	109732
4	2	44882	0.75	134147	0.91	164216	8.5	4722	21188	103	47448	7367	0.61	109392
2	2	49240	0.69	134968	0.84	165317	8.2	30378	25445	23666	46892	7958	0.56	110459

Table B.13: Execution statistics for bitonic (bit64) on 4 processors with varying queue and wait-buffer size (interleaving level 16, 2-way combining, equal module clock factors).

qu	wb	clks	putil	insts	nutil	mreqs	mlat	rqcon	wbfull	aqcon	ldcom	f&acom	mutil	maccs
32	32	61709	0.70	348020	0.86	426244	9.2	0	0	0	186180	25664	0.43	214386
16	32	61709	0.70	348020	0.86	426244	9.2	0	0	0	186180	25664	0.43	214386
8	32	61682	0.70	347938	0.86	426145	9.5	4	0	0	186061	25611	0.43	214458
4	32	61712	0.70	347957	0.86	426160	9.1	902	0	110	186139	25549	0.43	214453
2	32	67276	0.64	344204	0.78	421172	11.0	53686	0	80958	175925	29197	0.40	216032
32	16	61709	0.70	348020	0.86	426244	9.2	0	0	0	186180	25664	0.43	214386
16	16	61709	0.70	348020	0.86	426244	9.2	0	0	0	186180	25664	0.43	214386
8	16	61682	0.70	347938	0.86	426145	9.5	4	0	0	186061	25611	0.43	214458
4	16	61712	0.70	347957	0.86	426160	9.1	902	0	110	186139	25549	0.43	214453
2	16	67276	0.64	344204	0.78	421172	11.0	53686	0	80958	175925	29197	0.40	216032
32	8	61709	0.70	348020	0.86	426244	9.2	0	5	0	186180	25664	0.43	214386
16	8	61709	0.70	348020	0.86	426244	9.2	0	5	0	186180	25664	0.43	214386
8	8	61682	0.70	347938	0.86	426145	9.5	4	4	0	186061	25611	0.43	214458
4	8	61712	0.70	347957	0.86	426160	9.1	902	2	110	186139	25549	0.43	214453
2	8	66947	0.64	343574	0.78	420323	11.5	52478	389	78643	175602	28987	0.40	215711
32	4	61974	0.70	348523	0.86	426922	9.2	0	11782	0	184406	25548	0.44	216952
16	4	61974	0.70	348523	0.86	426922	9.2	0	11782	0	184406	25548	0.44	216952
8	4	61920	0.70	348316	0.86	426646	9.2	29	12323	0	185175	25510	0.44	215942
4	4	61973	0.70	348416	0.86	426783	9.2	1034	12383	124	184169	25492	0.44	217104
2	4	68012	0.63	343574	0.77	420322	12.6	74148	29886	80647	173057	28317	0.40	218924
32	2	65014	0.67	348528	0.82	426924	10.8	0	72010	0	159113	23373	0.47	244418
16	2	65050	0.67	348597	0.82	427019	11.0	228	71889	0	159440	23390	0.47	244168
8	2	64859	0.67	347892	0.82	426088	10.9	2437	76219	0	163740	23186	0.46	239137
4	2	64465	0.67	346953	0.82	424833	10.8	16752	81970	231	169566	23217	0.45	232024
2	2	73434	0.60	350591	0.73	429672	13.4	149720	93374	78553	161873	25778	0.41	242001

Table B.14: Execution statistics for bitonic (bit128) on 8 processors with varying queue and wait-buffer size (interleaving level 16, 2-way combining, equal module clock factors).

qu	wb	clks	putil	insts	nutil	mreqs	mlat	rqcon	wbfull	aqcon	ldcom	f&acom	mutil	maccs
32	32	84513	0.64	871593	0.79	1067440	10.9	0	0	0	608364	79420	0.28	379628
16	32	84513	0.64	871593	0.79	1067440	10.9	0	0	0	608364	79420	0.28	379628
8	32	84508	0.64	871590	0.79	1067420	10.9	70	0	0	608021	79390	0.28	379973
4	32	84555	0.64	871946	0.79	1067920	10.8	1001	0	221	609116	79613	0.28	379153
2	32	90384	0.59	855461	0.72	1045920	13.8	111789	0	164514	564462	83025	0.28	398405
32	16	84513	0.64	871581	0.79	1067430	10.9	0	0	0	608357	79419	0.28	379624
16	16	84513	0.64	871581	0.79	1067430	10.9	0	0	0	608357	79419	0.28	379624
8	16	84508	0.64	871590	0.79	1067420	10.9	70	0	0	608021	79390	0.28	379973
4	16	84555	0.64	871946	0.79	1067920	10.8	1001	0	221	609116	79613	0.28	379153
2	16	90384	0.59	855461	0.72	1045920	13.8	111789	0	164514	564462	83025	0.28	398405
32	8	84508	0.64	871427	0.79	1067210	11.2	0	2317	0	608141	79306	0.28	379726
16	8	84508	0.64	871427	0.79	1067210	11.2	0	2317	0	608141	79306	0.28	379726
8	8	84643	0.64	873022	0.79	1069340	10.8	67	2276	0	608569	79805	0.28	380937
4	8	84493	0.64	871564	0.79	1067400	11.0	1025	2279	250	609453	79445	0.28	378458
2	8	90642	0.59	855956	0.72	1046570	15.0	114980	7771	168784	564385	83077	0.28	399065
32	4	86177	0.64	877887	0.78	1075830	11.2	0	47048	0	585465	79895	0.30	410432
16	4	86142	0.64	877307	0.78	1075060	10.9	243	47907	0	586133	79775	0.30	409121
8	4	85897	0.64	875714	0.78	1072930	11.0	2178	50262	0	590130	79483	0.29	403283
4	4	85417	0.64	873436	0.78	1069880	11.2	6876	51547	384	594890	79114	0.29	395835
2	4	91697	0.58	858124	0.72	1049480	15.2	194368	95342	161828	560529	81130	0.28	407771
32	2	95245	0.58	890088	0.72	1092100	13.4	4326	253160	0	480138	76392	0.35	535514
16	2	95389	0.58	886631	0.71	1087490	13.6	24491	273748	0	499974	75851	0.34	511614
8	2	94465	0.58	881539	0.71	1080700	12.8	69725	293755	80	514072	75099	0.33	491476
4	2	92236	0.59	876839	0.73	1074440	12.9	120237	329379	1278	537987	75203	0.31	461198
2	2	104004	0.53	889536	0.66	1091370	15.2	544476	336037	177553	518081	80571	0.30	492674

Table B.15: Execution statistics for bitonic (bit256) on 16 processors with varying queue and wait-buffer size (interleaving level 16, 2-way combining, equal module clock factors).

qu	wb	clks	putil	insts	nutil	mreqs	mlat	rqcon	wbfull	aqcon	ldcom	f&acom	mutil	maccs
32	32	112975	0.59	2136350	0.72	2616680	12.6	0	0	0	1775430	230069	0.17	611111
16	32	112946	0.59	2135670	0.72	2615770	12.7	16	0	0	1775830	229889	0.17	609981
8	32	112979	0.59	2135740	0.72	2615870	12.6	342	0	0	1774800	229886	0.17	611114
4	32	113009	0.59	2136630	0.72	2617040	12.8	2114	0	700	1777060	230163	0.17	609740
2	32	119080	0.55	2100080	0.67	2568320	15.5	201968	0	305167	1675110	226807	0.17	666329
32	16	112975	0.59	2136370	0.72	2616710	12.6	0	0	0	1775450	230074	0.17	611115
16	16	112946	0.59	2135670	0.72	2615770	12.7	16	0	0	1775830	229889	0.17	609981
8	16	112979	0.59	2135740	0.72	2615870	12.6	342	0	0	1774800	229886	0.17	611114
4	16	113009	0.59	2136630	0.72	2617040	12.8	2114	0	700	1777060	230163	0.17	609740
2	16	119080	0.55	2100080	0.67	2568320	15.5	201968	0	305167	1675110	226807	0.17	666329
32	8	112985	0.59	2135850	0.72	2616020	12.7	0	10264	0	1773610	229872	0.17	612471
16	8	112985	0.59	2135850	0.72	2616020	12.7	0	10264	0	1773610	229872	0.17	612471
8	8	112986	0.59	2136090	0.72	2616340	12.8	74	10352	0	1773690	229951	0.17	612631
4	8	113013	0.59	2136710	0.72	2617170	12.6	1786	10561	627	1775490	230118	0.17	611499
2	8	119119	0.55	2097920	0.67	2565480	13.9	219237	35736	305059	1674070	226753	0.17	664593
32	4	118596	0.57	2182220	0.71	2677850	12.9	535	176828	0	1672520	239756	0.20	765503
16	4	118113	0.58	2173790	0.71	2666590	12.9	9530	188040	0	1685280	237608	0.20	743632
8	4	116739	0.58	2158800	0.71	2646610	12.8	28401	211272	61	1711150	234011	0.19	701380
4	4	115141	0.58	2145270	0.71	2628580	12.7	50713	242280	934	1734930	230946	0.18	662625
2	4	122229	0.54	2109860	0.66	2581370	16.4	530725	330092	312190	1651170	224936	0.18	705171
32	2	155735	0.41	2047160	0.50	2497720	22.2	272383	570528	0	1138120	151968	0.24	1207350
16	2	161057	0.42	2171650	0.52	2663770	22.8	673700	749111	90	1282270	190035	0.23	1191180
8	2	145628	0.46	2156040	0.57	2642920	21.0	817864	925140	370	1381950	202318	0.23	1058450
4	2	132042	0.51	2152750	0.62	2638610	20.0	712721	1181180	5424	1516560	217578	0.21	904324
2	2	147846	0.48	2254330	0.59	2773990	17.0	1937820	1147460	355671	1521510	245202	0.21	1007160

Table B.16: Execution statistics for bitonic (bit512) on 32 processors with varying queue and wait-buffer size (interleaving level 16, 2-way combining, equal module clock factors).

qu	wb	clks	putil	insts	nutil	mreqs	mlat	rqcon	wbfull	aqcon	ldcom	f&acom	mutil	maccs
32	32	148015	0.54	5125190	0.66	6276870	15.1	0	0	0	4703450	605670	0.10	967616
16	32	148015	0.54	5125190	0.66	6276870	15.1	32	0	0	4703450	605670	0.10	967616
8	32	147977	0.54	5123520	0.66	6274680	14.4	565	0	0	4704130	605205	0.10	965212
4	32	147923	0.54	5123340	0.66	6274410	15.2	3543	0	1515	4703460	605234	0.10	965588
2	32	154369	0.51	5080670	0.63	6217560	17.7	388798	0	514201	4572030	595892	0.11	1049490
32	16	148015	0.54	5125190	0.66	6276870	15.1	0	0	0	4703450	605670	0.10	967616
16	16	148015	0.54	5125190	0.66	6276870	15.1	32	0	0	4703450	605670	0.10	967616
8	16	147977	0.54	5123490	0.66	6274630	14.4	565	0	0	4704100	605199	0.10	965208
4	16	147923	0.54	5123340	0.66	6274410	15.2	3543	0	1515	4703460	605234	0.10	965588
2	16	154283	0.51	5080780	0.63	6217690	17.8	391262	12	511221	4572400	595926	0.11	1049220
32	8	148296	0.54	5130030	0.66	6283340	14.6	0	42465	0	4699980	606914	0.10	976308
16	8	148296	0.54	5130030	0.66	6283340	14.6	0	42465	0	4699980	606914	0.10	976308
8	8	148147	0.54	5127650	0.66	6280150	14.5	136	41955	0	4697960	606250	0.10	975803
4	8	148008	0.54	5124970	0.66	6276580	14.8	3417	43886	1656	4699250	605596	0.10	971603
2	8	154623	0.51	5089010	0.63	6228580	17.6	421264	112555	514823	4571750	597748	0.11	1058960
32	4	169118	0.45	4924470	0.56	6009160	22.8	212642	644263	0	3881220	504115	0.15	1623340
16	4	165014	0.47	5014220	0.58	6129000	22.3	321746	831315	0	4145070	543282	0.14	1440290
8	4	160011	0.50	5070880	0.61	6204640	22.0	351896	1110040	9	4358330	571632	0.12	1274420
4	4	156210	0.51	5088800	0.62	6228450	20.0	398434	1353240	3739	4484610	583343	0.12	1160310
2	4	160866	0.49	5075980	0.60	6211330	19.9	1459240	1465050	556309	4445930	582471	0.11	1182760

Table B.17: Execution statistics for bitonic (bit1024) on 64 processors with varying queue and wait-buffer size (interleaving level 16, 2-way combining, equal module clock factors).

qu	wb	clks	putil	insts	nutil	mreqs	mlat	rqcon	wbfull	aqcon	ldcom	f&acom	mutil	maccs
32	32	190268	0.50	12149200	0.61	14881700	16.2	0	0	0	11786600	1516480	0.06	1578400
16	32	190268	0.50	12149200	0.61	14881700	16.2	64	0	0	11786600	1516480	0.06	1578400
8	32	190063	0.50	12137900	0.61	14866600	16.2	1142	0	0	11777200	1512990	0.06	1576230
4	32	190129	0.50	12140400	0.61	14870000	16.5	7403	0	4974	11779500	1513840	0.06	1576420
2	32	197137	0.48	12166200	0.59	14904400	18.8	835472	0	793847	11693700	1520100	0.07	1690370
32	16	190085	0.50	12138500	0.61	14867400	16.4	0	93	0	11777500	1513110	0.06	1576520
16	16	190085	0.50	12138500	0.61	14867400	16.4	64	93	0	11777500	1513110	0.06	1576520
8	16	190356	0.50	12153300	0.61	14887200	16.3	1158	67	0	11791100	1517850	0.06	1577940
4	16	190302	0.50	12151200	0.61	14884500	16.2	7369	63	4929	11788700	1517180	0.06	1578300
2	16	196520	0.48	12150700	0.59	14883700	20.2	835491	398	772797	11688900	1515650	0.07	1678990
32	8	192471	0.50	12210200	0.61	14963100	16.5	132	294027	0	11763000	1532960	0.07	1666920
16	8	192447	0.50	12209700	0.61	14962300	16.5	1998	296032	0	11765000	1532840	0.07	1664280
8	8	192161	0.50	12198600	0.61	14947500	16.5	11216	304735	1	11766000	1529730	0.07	1651550
4	8	191310	0.50	12170300	0.61	14909900	16.5	40159	317584	5085	11771400	1521770	0.07	1616460
2	8	197685	0.48	12142600	0.59	14873000	20.1	1023260	462703	823988	11651600	1512130	0.07	1709010

Table B.18: Execution statistics for bitonic (bit2048) on 128 processors with varying queue and wait-buffer size (interleaving level 16, 2-way combining, equal module clock factors).

qu	wb	clks	putil	insts	nutil	mreqs	mlat	rqcon	wbfull	aqcon	ldcom	f&acom	mutil	maccs
2	32	28464	0.66	37818	0.89	50572	5.0	6376	0	0	4925	1788	0.77	43858
4	32	24837	0.73	36130	0.97	48289	5.5	1208	0	0	5611	2054	0.82	40621
8	32	25781	0.74	38087	0.99	50910	5.5	0	0	0	6423	2415	0.82	42069
16	32	25781	0.74	38087	0.99	50910	5.5	0	0	0	6423	2415	0.82	42069
32	32	25781	0.74	38087	0.99	50910	5.5	0	0	0	6423	2415	0.82	42069
2	16	28464	0.66	37818	0.89	50572	5.0	6376	0	0	4925	1788	0.77	43858
4	16	24837	0.73	36130	0.97	48289	5.5	1208	0	0	5611	2054	0.82	40621
8	16	25781	0.74	38087	0.99	50910	5.5	0	0	0	6423	2415	0.82	42069
16	16	25781	0.74	38087	0.99	50910	5.5	0	0	0	6423	2415	0.82	42069
32	16	25781	0.74	38087	0.99	50910	5.5	0	0	0	6423	2415	0.82	42069
2	8	28464	0.66	37818	0.89	50572	5.0	6376	0	0	4925	1788	0.77	43858
4	8	24837	0.73	36130	0.97	48289	5.5	1208	0	0	5611	2054	0.82	40621
8	8	25781	0.74	38087	0.99	50910	5.5	0	0	0	6423	2415	0.82	42069
16	8	25781	0.74	38087	0.99	50910	5.5	0	0	0	6423	2415	0.82	42069
32	8	25781	0.74	38087	0.99	50910	5.5	0	0	0	6423	2415	0.82	42069
2	4	28464	0.66	37818	0.89	50572	5.0	6376	0	0	4925	1788	0.77	43858
4	4	24837	0.73	36130	0.97	48289	5.5	1208	27	0	5611	2054	0.82	40621
8	4	25758	0.74	38051	0.99	50860	6.0	0	77	0	6441	2384	0.82	42031
16	4	25758	0.74	38051	0.99	50860	6.0	0	77	0	6441	2384	0.82	42031
32	4	25758	0.74	38051	0.99	50860	6.0	0	77	0	6441	2384	0.82	42031
2	2	28374	0.66	37448	0.88	50050	4.5	6718	2065	0	4683	1703	0.77	43661
4	2	25408	0.73	36909	0.97	49320	5.5	1290	3584	0	5720	2137	0.82	41461
8	2	25104	0.74	36924	0.98	49332	6.5	5	3672	0	5720	2155	0.83	41454
16	2	25276	0.74	37209	0.98	49716	5.0	0	3801	0	6034	2170	0.82	41509
32	2	25276	0.74	37209	0.98	49716	5.0	0	3801	0	6034	2170	0.82	41509

Table B.19: Execution statistics for Moore (moo32) on 2 processors with varying queue and wait-buffer size (interleaving level 16, 2-way combining, equal module clock factors).

qu	wb	clks	putil	insts	nutil	mreqs	mlat	rqcon	wbfull	aqcon	ldcom	f&acom	mutil	maccs
2	32	30437	0.59	72437	0.80	97211	8.5	27409	0	18275	22023	9409	0.54	65772
4	32	26933	0.68	73546	0.92	98716	7.8	1035	0	85	22509	9181	0.62	67019
8	32	26067	0.68	71182	0.92	95516	7.5	1	0	0	21521	8615	0.63	65373
16	32	26186	0.68	71550	0.92	96012	8.2	0	0	0	21869	8752	0.62	65383
32	32	26186	0.68	71550	0.92	96012	8.2	0	0	0	21869	8752	0.62	65383
2	16	30437	0.59	72437	0.80	97211	8.5	27409	0	18275	22023	9409	0.54	65772
4	16	26933	0.68	73546	0.92	98716	7.8	1035	0	85	22509	9181	0.62	67019
8	16	26067	0.68	71182	0.92	95516	7.5	1	0	0	21521	8615	0.63	65373
16	16	26186	0.68	71550	0.92	96012	8.2	0	0	0	21869	8752	0.62	65383
32	16	26186	0.68	71550	0.92	96012	8.2	0	0	0	21869	8752	0.62	65383
2	8	29577	0.60	71019	0.81	95280	10.0	25395	14	16897	22575	9327	0.54	63371
4	8	26933	0.68	73546	0.92	98716	7.8	1035	0	85	22509	9181	0.62	67019
8	8	26067	0.68	71182	0.92	95516	7.5	1	0	0	21521	8615	0.63	65373
16	8	26186	0.68	71550	0.92	96012	8.2	0	0	0	21869	8752	0.62	65383
32	8	26186	0.68	71550	0.92	96012	8.2	0	0	0	21869	8752	0.62	65383
2	4	30943	0.59	72958	0.79	97973	9.5	29125	4186	18598	22046	9442	0.54	66475
4	4	26322	0.68	71448	0.91	95870	8.5	1408	3071	78	21130	8644	0.63	66087
8	4	26539	0.68	72389	0.91	97155	7.8	1	3135	0	21700	8924	0.63	66523
16	4	26793	0.68	73067	0.91	98060	8.5	0	2921	0	21958	9034	0.63	67061
32	4	26793	0.68	73067	0.91	98060	8.5	0	2921	0	21958	9034	0.63	67061
2	2	31941	0.58	73674	0.77	98867	9.2	33456	15817	14798	20468	8670	0.55	69722
4	2	27903	0.65	72749	0.87	97634	8.8	5291	13614	193	20181	8359	0.62	69084
8	2	27511	0.65	71432	0.87	95885	9.8	54	12478	0	19038	7846	0.63	68988
16	2	28293	0.65	73471	0.87	98631	9.5	5	13154	0	19547	8253	0.63	70821
32	2	28461	0.65	73824	0.87	99099	9.0	0	13165	0	19696	8300	0.62	71092

Table B.20: Execution statistics for Moore (moo64) on 4 processors with varying queue and wait-buffer size (interleaving level 16, 2-way combining, equal module clock factors).

qu	wb	clks	putil	insts	nutil	mreqs	mlat	rqcon	wbfull	aqcon	ldcom	f&acom	mutil	maccs
2	32	62073	0.55	270870	0.73	363033	13.2	123988	0	80633	90523	34999	0.48	237492
4	32	53569	0.62	265923	0.83	356167	9.2	3071	0	200	92503	30667	0.54	232979
8	32	57016	0.62	282254	0.83	378682	9.5	5	0	0	96992	33574	0.54	248098
16	32	57360	0.62	283900	0.83	380954	9.0	0	0	0	97005	33766	0.55	250166
32	32	57360	0.62	283900	0.83	380954	9.0	0	0	0	97005	33766	0.55	250166
2	16	62073	0.55	270870	0.73	363033	13.2	123988	0	80633	90523	34999	0.48	237492
4	16	53569	0.62	265923	0.83	356167	9.2	3071	0	200	92503	30667	0.54	232979
8	16	57016	0.62	282254	0.83	378682	9.5	5	0	0	96992	33574	0.54	248098
16	16	57360	0.62	283900	0.83	380954	9.0	0	0	0	97005	33766	0.55	250166
32	16	57360	0.62	283900	0.83	380954	9.0	0	0	0	97005	33766	0.55	250166
2	8	62801	0.54	272877	0.73	365935	12.1	126493	252	81601	89272	35380	0.48	241265
4	8	58417	0.62	289199	0.83	387951	9.5	3040	34	307	101371	34931	0.54	251630
8	8	56664	0.62	280408	0.83	376131	9.2	4	26	0	95375	32791	0.55	247949
16	8	56030	0.62	277262	0.83	371886	9.4	0	29	0	93900	32262	0.55	245706
32	8	56030	0.62	277256	0.83	371878	9.4	0	29	0	93898	32261	0.55	245703
2	4	59270	0.53	253627	0.72	339528	12.5	136072	21150	75177	79607	30400	0.48	229504
4	4	57601	0.62	283830	0.83	380555	9.1	3754	10385	301	95529	32876	0.55	252128
8	4	53410	0.62	263143	0.83	352785	9.9	7	9745	0	86678	29302	0.55	236787
16	4	57413	0.62	283317	0.83	380014	9.8	0	10997	0	96865	33460	0.54	249671
32	4	57413	0.62	283317	0.83	380014	9.8	0	10997	0	96865	33460	0.54	249671
2	2	65057	0.51	266666	0.69	357557	13.1	192131	55741	77283	77832	30368	0.48	249335
4	2	57485	0.59	271678	0.79	364361	12.4	18410	48888	1768	83773	28997	0.55	251565
8	2	59510	0.58	278376	0.78	373326	10.4	4385	48101	0	84067	29178	0.55	260057
16	2	60309	0.57	272967	0.76	366103	15.6	401	41417	0	78036	26815	0.54	261222
32	2	55891	0.56	252407	0.76	338349	15.8	12	34338	0	68514	22773	0.55	247029

Table B.21: Execution statistics for Moore (moo128) on 8 processors with varying queue and wait-buffer size (interleaving level 16, 2-way combining, equal module clock factors).

qu	wb	clks	putil	insts	nutil	mreqs	mlat	rqcon	wbfull	aqcon	ldcom	f&acom	mutil	maccs
2	32	59011	0.51	480728	0.68	644700	15.7	192397	0	137513	217884	77494	0.37	349283
4	32	53860	0.57	491834	0.76	659350	11.0	2338	0	446	237257	74662	0.40	347395
8	32	53966	0.57	492823	0.77	660939	11.1	0	0	0	238188	75321	0.40	347391
16	32	53966	0.57	492823	0.77	660939	11.1	0	0	0	238188	75321	0.40	347391
32	32	53966	0.57	492823	0.77	660939	11.1	0	0	0	238188	75321	0.40	347391
2	16	59011	0.51	480728	0.68	644700	15.7	192397	0	137513	217884	77494	0.37	349283
4	16	53860	0.57	491834	0.76	659350	11.0	2338	0	446	237257	74662	0.40	347395
8	16	53966	0.57	492823	0.77	660939	11.1	0	0	0	238188	75321	0.40	347391
16	16	53966	0.57	492823	0.77	660939	11.1	0	0	0	238188	75321	0.40	347391
32	16	53966	0.57	492823	0.77	660939	11.1	0	0	0	238188	75321	0.40	347391
2	8	58726	0.50	473521	0.68	635060	15.2	192173	4770	143579	211832	76546	0.37	346644
4	8	54151	0.57	494228	0.76	662750	10.5	2319	2299	428	240556	76162	0.40	345998
8	8	54847	0.57	500658	0.76	671171	10.9	0	2339	0	243786	77164	0.40	350188
16	8	54847	0.57	500658	0.76	671171	10.9	0	2339	0	243786	77164	0.40	350188
32	8	54847	0.57	500658	0.76	671171	10.9	0	2339	0	243786	77164	0.40	350188
2	4	59738	0.49	469639	0.66	629788	15.3	265906	61336	146682	205861	74030	0.37	349853
4	4	54731	0.56	493277	0.76	661485	10.9	7627	31016	529	231844	73707	0.41	355894
8	4	54340	0.56	488004	0.75	654156	11.1	2485	30346	0	227573	72089	0.41	354455
16	4	55091	0.56	489717	0.75	656933	11.3	1083	30670	0	222955	71523	0.41	362416
32	4	54661	0.56	485557	0.74	651456	11.1	38	31760	0	224674	72579	0.40	354166
2	2	63086	0.46	461804	0.61	618851	13.1	451694	146155	133410	183223	64045	0.37	371528
4	2	56996	0.51	468507	0.69	628146	14.5	93540	141877	10078	197064	62832	0.40	368195
8	2	59399	0.49	467738	0.66	627116	15.5	67079	125435	4305	188093	59488	0.40	379487
16	2	59169	0.48	450126	0.64	603372	16.4	33006	111963	10	175141	54346	0.39	373839
32	2	60170	0.47	448564	0.62	600996	16.2	14101	106804	0	170983	51798	0.39	378164

Table B.22: Execution statistics for Moore (moo256) on 16 processors with varying queue and wait-buffer size (interleaving level 16, 2-way combining, equal module clock factors).

qu	wb	clks	putil	insts	nutil	mreqs	mlat	rqcon	wbfull	aqcon	ldcom	f&acom	mutil	maccs
2	32	74852	0.47	1134620	0.64	1524340	16.3	384268	0	279692	634103	206978	0.29	683190
4	32	72971	0.52	1214420	0.70	1633260	12.8	3119	0	999	717047	222978	0.30	693163
8	32	71317	0.52	1188260	0.70	1596960	12.5	0	0	0	693881	213586	0.30	689427
16	32	71317	0.52	1188260	0.70	1596960	12.5	0	0	0	693881	213586	0.30	689427
32	32	71317	0.52	1188260	0.70	1596960	12.5	0	0	0	693881	213586	0.30	689427
2	16	74852	0.47	1134620	0.64	1524340	16.3	384268	0	279692	634103	206978	0.29	683190
4	16	72971	0.52	1214420	0.70	1633260	12.8	3119	0	999	717047	222978	0.30	693163
8	16	71317	0.52	1188260	0.70	1596960	12.5	0	0	0	693881	213586	0.30	689427
16	16	71317	0.52	1188260	0.70	1596960	12.5	0	0	0	693881	213586	0.30	689427
32	16	71317	0.52	1188260	0.70	1596960	12.5	0	0	0	693881	213586	0.30	689427
2	8	75303	0.47	1129910	0.63	1517540	15.5	395460	28819	292169	632093	207327	0.28	678046
4	8	75495	0.52	1256240	0.70	1687560	12.6	3388	11185	973	749277	231965	0.29	706252
8	8	68535	0.52	1134990	0.70	1525340	12.6	11	11584	0	654124	199761	0.31	671376
16	8	70059	0.52	1161500	0.70	1560690	12.8	0	11521	0	671726	205422	0.30	683469
32	8	70059	0.52	1161500	0.70	1560690	12.8	0	11521	0	671726	205422	0.30	683469
2	4	76490	0.45	1101440	0.60	1479600	18.6	691120	198770	303897	593261	194615	0.28	691648
4	4	71250	0.51	1158030	0.68	1555650	12.6	58887	129993	1059	658093	201190	0.31	696300
8	4	75305	0.50	1203930	0.67	1617210	12.8	53846	137596	39	689015	213357	0.30	714761
16	4	74413	0.49	1161260	0.65	1559450	12.7	31696	127852	0	643070	197820	0.30	718488
32	4	72361	0.47	1083490	0.63	1454540	12.9	21009	111148	0	572513	173251	0.31	708703
2	2	83285	0.40	1071600	0.54	1438890	17.6	1483060	452931	240989	521859	171834	0.28	745088
4	2	82663	0.42	1100230	0.56	1477130	21.6	838787	476388	3876	544218	169915	0.29	762840
8	2	84794	0.39	1049660	0.52	1408750	24.5	694967	393467	30	487585	149831	0.28	771143
16	2	96918	0.36	1111600	0.48	1491460	22.1	485645	383928	0	503945	153690	0.27	833622
32	2	94289	0.34	1017330	0.45	1364330	25.4	238744	307264	0	443691	128798	0.26	791575

Table B.23: Execution statistics for Moore (moo512) on 32 processors with varying queue and wait-buffer size (interleaving level 16, 2-way combining, equal module clock factors).

qu	wb	clks	putil	insts	nutil	mreqs	mlat	rqcon	wbfull	aqcon	ldcom	f&acom	mutil	maccs
2	32	89818	0.44	2526290	0.59	3398860	18.1	709488	0	509895	1701210	527115	0.20	1170410
4	32	85871	0.48	2647010	0.65	3564140	14.5	3462	0	1750	1820800	552823	0.22	1190380
8	32	87458	0.48	2696290	0.65	3629760	14.4	0	0	0	1863280	566741	0.21	1199610
16	32	87458	0.48	2696290	0.65	3629760	14.4	0	0	0	1863280	566741	0.21	1199610
32	32	87458	0.48	2696290	0.65	3629760	14.4	0	0	0	1863280	566741	0.21	1199610
2	16	89527	0.44	2522180	0.59	3392780	17.4	684221	167	515479	1701310	527453	0.20	1163850
4	16	85871	0.48	2647010	0.65	3564140	14.5	3462	0	1750	1820800	552823	0.22	1190380
8	16	87458	0.48	2696290	0.65	3629760	14.4	0	0	0	1863280	566741	0.21	1199610
16	16	87458	0.48	2696290	0.65	3629760	14.4	0	0	0	1863280	566741	0.21	1199610
32	16	87458	0.48	2696290	0.65	3629760	14.4	0	0	0	1863280	566741	0.21	1199610
2	8	92630	0.43	2559870	0.58	3444800	17.6	823450	135973	558367	1709390	534214	0.20	1201060
4	8	82789	0.48	2527030	0.64	3401860	14.5	7209	47428	1891	1708880	514763	0.22	1178090
8	8	86790	0.48	2648410	0.64	3565140	14.4	2329	53837	0	1812090	549003	0.22	1203920
16	8	87215	0.48	2661700	0.64	3582850	14.5	866	54407	0	1831940	557858	0.21	1192920
32	8	87005	0.48	2651000	0.64	3568730	14.3	91	54999	0	1816570	553797	0.22	1198210
2	4	97507	0.41	2535870	0.55	3409380	19.9	2052980	776939	559498	1643710	508115	0.20	1257390
4	4	97327	0.41	2559780	0.55	3442890	21.1	1143730	748310	19368	1663310	504227	0.20	1275160
8	4	93115	0.40	2407620	0.54	3236210	19.1	621173	590632	1313	1525340	456503	0.21	1254160
16	4	101393	0.35	2279680	0.47	3063500	23.9	671709	447523	94	1364630	402573	0.20	1295920
32	4	105719	0.31	2107790	0.42	2829350	27.4	416521	299363	0	1193050	339162	0.19	1296620
2	2	104407	0.33	2225520	0.45	2984490	26.9	5392280	1175620	380432	1280900	387345	0.20	1315970
4	2	103881	0.32	2132390	0.43	2859860	34.1	4227680	1000810	8899	1167200	343331	0.20	1348950
8	2	107629	0.29	1999740	0.39	2679440	53.9	3736050	818706	809	1037030	293384	0.20	1348510
16	2	114521	0.25	1840090	0.34	2463130	84.5	2597280	581678	76	874668	229608	0.19	1358170
32	2	153470	0.20	2001820	0.27	2677680	222.3	2387880	594548	2	948838	248159	0.15	1479970

Table B.24: Execution statistics for Moore (moo1024) on 64 processors with varying queue and wait-buffer size (interleaving level 16, 2-way combining, equal module clock factors).

qu	wb	clks	putil	insts	nutil	mreqs	mlat	rqcon	wbfull	aqcon	ldcom	f&acom	mutil	maccs
2	32	114994	0.41	6051040	0.55	8142330	19.4	1326780	0	980364	4580680	1326770	0.15	2234640
4	32	110503	0.44	6275280	0.60	8452590	17.8	4513	0	6553	4799090	1391410	0.16	2261830
8	32	111031	0.44	6315720	0.60	8506390	17.9	1	0	0	4842010	1404450	0.16	2259700
16	32	111031	0.44	6315720	0.60	8506390	17.9	0	0	0	4842010	1404450	0.16	2259700
32	32	111031	0.44	6315720	0.60	8506390	17.9	0	0	0	4842010	1404450	0.16	2259700
2	16	106109	0.41	5561660	0.55	7487320	19.2	1338100	6712	906008	4124410	1174010	0.16	2188650
4	16	110427	0.44	6268990	0.60	8446080	17.8	4462	83	6507	4792760	1391230	0.16	2261820
8	16	111284	0.45	6358430	0.60	8562770	17.0	2	44	2	4900610	1421790	0.16	2240110
16	16	111029	0.44	6306490	0.60	8500720	17.5	0	13	0	4824600	1406910	0.16	2268950
32	16	111029	0.44	6306430	0.60	8500650	17.6	0	13	0	4824550	1406880	0.16	2268950
2	8	116548	0.40	5959000	0.54	8014750	20.4	1909810	593687	1066630	4465120	1289380	0.15	2259980
4	8	112544	0.42	6087830	0.57	8199160	18.6	294321	486225	25538	4589860	1318840	0.16	2290180
8	8	113043	0.44	6295180	0.59	8480210	17.5	67090	280161	935	4781750	1395340	0.16	2302850
16	8	125796	0.43	6933050	0.58	9339280	18.3	47301	332707	121	5361650	1601750	0.15	2375610
32	8	112867	0.43	6144090	0.57	8267640	18.1	19635	276411	0	4637060	1338050	0.16	2292270
2	4	114618	0.36	5258380	0.48	7073890	27.8	6616200	2013820	898585	3717900	1050210	0.16	2305370
4	4	133924	0.33	5694420	0.45	7653230	34.3	7185500	2152700	33851	4041430	1149860	0.14	2461380
8	4	128314	0.29	4735750	0.39	6353450	50.4	6930060	1215600	5133	3119330	830787	0.15	2402460
16	4	151043	0.23	4386460	0.30	5871560	92.4	6693810	750026	1021	2700480	667703	0.13	2502170
32	4	223868	0.18	5109510	0.24	6828600	113.5	6715250	611899	918	2958640	703103	0.11	3165570

Table B.25: Execution statistics for Moore (moo2048) on 128 processors with varying queue and wait-buffer size (interleaving level 16, 2-way combining, equal module clock factors).

prcrs	k-way	clks	putil	insts	nutil	mreqs	mlat	rqcon	wbfull	aqcon	ldcom	f&acom	mutil	maccs
2	0	1825	0.71	2584	0.88	3212	6.0	433	0	0	0	0	0.88	3207
2	2	1587	0.81	2563	1.00	3180	4.0	0	0	0	1470	32	0.53	1677
2	3	1587	0.81	2563	1.00	3180	4.0	0	0	0	1470	32	0.53	1677
2	4	1587	0.81	2563	1.00	3180	4.0	0	0	0	1470	32	0.53	1677
4	0	3725	0.60	8970	0.74	10969	12.2	4070	0	0	0	0	0.74	10952
4	2	2557	0.83	8483	1.00	10244	7.0	0	0	0	7139	96	0.29	3006
4	3	2557	0.83	8483	1.00	10244	7.0	0	0	0	7141	96	0.29	3004
4	4	2557	0.83	8483	1.00	10244	7.0	0	0	0	7142	96	0.29	3003
8	0	5139	0.47	19439	0.59	24181	21.0	19428	0	11	0	0	0.59	24144
8	2	2756	0.77	16936	0.93	20453	8.2	0	0	0	16624	224	0.16	3593
8	3	2756	0.77	16936	0.93	20453	8.2	0	0	0	16625	224	0.16	3592
8	4	2756	0.77	16936	0.93	20453	8.2	0	0	0	16625	224	0.16	3592
16	0	35863	0.20	116790	0.27	156026	91.2	662430	0	120	0	0	0.27	155936
16	2	5348	0.71	60607	0.84	71806	10.1	0	0	0	62720	480	0.10	8588
16	3	5348	0.71	60607	0.84	71806	10.1	0	0	0	62720	480	0.10	8588
16	4	5348	0.71	60607	0.84	71806	10.1	0	0	0	62720	480	0.10	8588
32	0	52434	0.16	273754	0.22	372334	89.2	2225170	0	367	0	0	0.22	372137
32	2	5765	0.65	120320	0.77	142333	12.1	0	0	0	132284	992	0.05	9040
32	3	5765	0.65	120320	0.77	142335	12.0	0	0	0	132287	992	0.05	9039
32	4	5765	0.65	120320	0.77	142335	12.0	0	0	0	132287	992	0.05	9039
64	0	351805	0.08	1741260	0.11	2461571	395.0	38439800	0	2265	0	0	0.11	2461140
64	2	11820	0.60	453632	0.70	530431	14.0	0	0	0	502675	2016	0.03	25677
64	3	11820	0.60	453632	0.70	530431	14.0	0	0	0	502675	2016	0.03	25677
64	4	11820	0.60	453632	0.70	530431	14.0	0	0	0	502675	2016	0.03	25677
128	2	12853	0.55	907264	0.64	1060863	16.0	0	0	0	1028948	4064	0.02	27724
128	3	12853	0.55	907264	0.64	1060863	16.0	0	0	0	1028948	4064	0.02	27724
128	4	12853	0.55	907264	0.64	1060863	16.0	0	0	0	1028948	4064	0.02	27724
256	2	26814	0.51	3518460	0.60	4087548	18.0	0	0	0	3991513	8160	0.01	87626
256	3	26814	0.51	3518460	0.60	4087548	18.0	0	0	0	3991513	8160	0.01	87626
256	4	26814	0.51	3518460	0.60	4087548	18.0	0	0	0	3991513	8160	0.01	87626

Table B.26: Execution statistics for matrix multiply with increasing combining level (interleaving level 16, queue size 4, wait-buffer size 64, equal module clock factors).

prcrs	k-way	clks	putil	insts	nutil	mreqs	mlat	rqcon	wbfull	aqcon	ldcom	f&acom	mutil	maccs
2	0	37686	0.67	50133	0.81	61385	6.5	13935	0	0	0	0	0.81	61380
2	2	30995	0.81	50434	1.00	61790	5.5	130	0	0	11054	1998	0.79	48736
2	3	31086	0.81	50580	1.00	61983	6.0	161	0	0	11312	2028	0.78	48641
2	4	30998	0.81	50504	1.00	61880	6.0	91	0	0	11914	2032	0.77	47931
4	0	72256	0.52	151124	0.65	186852	10.2	116333	0	15	0	0	0.65	186836
4	2	43820	0.77	134694	0.94	164952	7.2	647	0	81	50705	8097	0.61	106141
4	3	43674	0.77	134755	0.94	165034	7.0	544	0	66	51827	8296	0.60	104904
4	4	43573	0.77	134641	0.95	164876	7.2	512	0	52	52645	8340	0.60	103885
8	0	103138	0.47	389894	0.58	482074	15.8	391777	0	164	0	0	0.58	482029
8	2	61712	0.70	347929	0.86	426126	9.4	902	0	110	186123	25547	0.43	214438
8	3	61528	0.71	348626	0.87	427058	9.0	645	0	118	188858	26179	0.43	212005
8	4	61490	0.71	348468	0.87	426849	9.2	733	0	118	189483	25979	0.43	211372
16	2	84555	0.64	871900	0.79	1067851	10.9	1001	0	221	609077	79606	0.28	379134
16	3	83944	0.65	871221	0.79	1066934	10.9	1061	0	214	615460	79813	0.28	371626
16	4	84076	0.65	872509	0.79	1068652	10.9	1046	0	211	617464	80248	0.28	370906
32	2	113009	0.59	2136580	0.72	2616964	12.8	2114	0	700	1777015	230150	0.17	609728
32	3	112077	0.60	2134010	0.73	2613552	12.6	1994	0	536	1789333	230378	0.17	593773
32	4	112101	0.60	2134580	0.73	2614312	12.8	1978	0	537	1790304	230588	0.17	593358
64	2	147923	0.54	5123230	0.66	6274294	15.1	3543	0	1515	4703369	605210	0.10	965576
64	3	146315	0.55	5120110	0.67	6270121	14.4	3058	0	1069	4732399	606341	0.10	931256
64	4	146257	0.55	5118740	0.67	6268305	14.3	2944	0	1006	4731947	605953	0.10	930271
128	2	190129	0.50	12140300	0.61	14869879	16.5	7403	0	4974	11779443	1513820	0.06	1576413
128	3	187268	0.51	12117900	0.62	14839999	16.3	4127	0	2745	11818147	1510860	0.06	1510713
128	4	187161	0.51	12110900	0.62	14830689	16.1	4374	0	2562	11811109	1508690	0.06	1510676

Table B.27: Execution statistics for bitonic with increasing combining level (interleaving level 16, queue size 4, wait-buffer size 64, equal module clock factors).

prcrs	k-way	clks	putil	insts	nutil	mreqs	mlat	rqcon	wbfull	aqcon	ldcom	f&acom	mutil	maccs
2	0	31428	0.58	36730	0.78	49093	6.5	13684	0	0	0	0	0.78	49088
2	2	24837	0.73	36118	0.97	48275	6.0	1208	0	0	5609	2052	0.82	40612
2	3	25331	0.73	37018	0.98	49462	6.5	1055	0	0	6185	2233	0.81	41042
2	4	25514	0.73	37383	0.98	49950	5.5	933	0	0	6460	2345	0.81	41143
4	0	36162	0.44	63341	0.59	85035	11.8	71884	0	29	0	0	0.59	85019
4	2	26933	0.68	73531	0.92	98693	7.2	1035	0	85	22503	9178	0.62	67004
4	3	27626	0.69	75929	0.92	101857	7.5	898	0	57	23839	9629	0.62	68383
4	4	26861	0.69	73881	0.92	99124	7.2	905	0	51	23111	9399	0.62	66607
8	0	72323	0.41	238250	0.55	318969	17.0	316626	0	167	0	0	0.55	318920
8	2	53569	0.62	265898	0.83	356132	9.4	3071	0	200	92491	30659	0.54	232964
8	3	57943	0.62	288753	0.84	387135	9.4	2489	0	242	102658	35096	0.54	249367
8	4	57819	0.62	288103	0.84	386432	9.0	2419	0	246	102237	35294	0.54	248887
16	0	72730	0.29	335970	0.39	448804	45.2	1084100	0	552	0	0	0.39	448679
16	2	53860	0.57	491802	0.77	659313	11.1	2338	0	446	237235	74658	0.40	347384
16	3	53568	0.57	492292	0.77	660061	10.8	1817	0	325	240185	75177	0.40	344665
16	4	53888	0.58	495941	0.77	664742	10.9	1897	0	368	244239	76436	0.40	344037
32	0	5038280	0.10	15471200	0.13	20630136	1933.0	261250000	0	2762570	0	0	0.13	20629956
32	2	72971	0.52	1214370	0.70	1633194	12.9	3119	0	999	717002	222965	0.30	693151
32	3	75346	0.53	1274050	0.71	1711549	12.4	2295	0	793	778302	240255	0.29	692931
32	4	69419	0.53	1168680	0.71	1570997	12.6	2219	0	780	689325	211309	0.30	670301
64	2	85871	0.48	2647010	0.65	3564134	14.5	3462	0	1750	1820800	552823	0.22	1190375
64	3	87577	0.49	2736450	0.66	3681882	14.4	2690	0	1187	1921082	582302	0.21	1178374
64	4	88021	0.49	2745410	0.66	3697301	14.3	2342	0	1201	1926820	589739	0.21	1180615
128	2	110503	0.44	6275220	0.60	8452505	17.8	4513	0	6553	4799040	1391390	0.16	2261824
128	3	110547	0.45	6389540	0.61	8607945	16.8	3225	0	2270	4961044	1447580	0.16	2199075
128	4	111204	0.45	6421320	0.61	8652800	17.0	3469	0	2243	4971042	1447110	0.16	2234399
256	2	133299	0.41	13914800	0.55	18781441	20.3	6078	0	23098	11602533	3299950	0.11	3878504
256	3	132860	0.42	14216000	0.56	19194554	19.2	3769	0	4798	11933883	3407320	0.11	3852884
256	4	148910	0.42	16015300	0.57	21606227	18.2	3894	0	5076	13662894	4002950	0.10	3939929

Table B.28: Execution statistics for Moore with increasing combining level (interleaving level 16, queue size 4, wait-buffer size 64, equal module clock factors).

prcrs	prcs	clks	putil	insts	nutil	mreqs	mlat	rqcon	wbfull	aqcon	ldcom	f&acom	mutil	maccs
1	16	10043	0.83	8385	1.00	10054	2.0	0	0	0	0	0	1.00	10054
1	24	11279	0.82	9209	1.00	11289	2.0	0	0	0	0	0	1.00	11289
1	32	10021	0.83	8364	1.00	10031	2.0	0	0	0	0	0	1.00	10031
1	40	12519	0.80	10028	1.00	12529	2.0	0	0	0	0	0	1.00	12529
1	48	15008	0.78	11695	1.00	15018	2.0	0	0	0	0	0	1.00	15018
1	56	17481	0.76	13360	1.00	17491	2.0	0	0	0	0	0	1.00	17491
1	64	10099	0.83	8395	1.00	10109	2.0	0	0	0	0	0	1.00	10109
2	16	10046	0.83	16768	1.00	20112	5.0	0	0	0	9343	128	0.53	10639
2	24	11280	0.82	18414	1.00	22580	4.0	0	0	0	10562	144	0.53	11873
2	32	10022	0.83	16726	1.00	20064	5.0	0	0	0	9321	128	0.53	10614
2	40	12519	0.80	20053	1.00	25058	5.0	0	0	0	11783	160	0.52	13114
2	48	15009	0.78	23390	1.00	30038	5.0	0	0	0	14241	192	0.52	15604
2	56	17482	0.76	26716	1.00	34984	5.0	0	0	0	16688	224	0.52	18071
2	64	10100	0.83	16790	1.00	20220	5.0	0	0	0	9398	128	0.53	10693
4	16	17780	0.85	60295	1.00	71160	6.2	0	0	0	49708	384	0.30	21063
4	24	19990	0.83	66192	1.00	80000	7.0	0	0	0	56292	432	0.29	23273
4	32	17753	0.85	60200	1.00	71052	6.5	0	0	0	49632	384	0.30	21033
4	40	22202	0.81	72100	1.00	88848	6.5	0	0	0	62881	480	0.29	25484
4	48	26629	0.79	83906	1.00	106556	7.5	0	0	0	76056	576	0.28	29921
4	56	31043	0.77	95700	1.00	124212	6.2	0	0	0	89212	672	0.28	34325
4	64	17803	0.85	60270	1.00	71252	7.0	0	0	0	49774	384	0.30	21091
8	16	19399	0.78	120437	0.92	142119	8.0	0	0	0	115818	896	0.16	25394
8	24	19991	0.83	132384	1.00	160016	8.5	0	0	0	131373	1008	0.17	27628
8	32	17754	0.85	120400	1.00	142120	8.2	0	0	0	115831	896	0.18	25386
8	40	22203	0.81	144200	1.00	177712	8.2	0	0	0	146746	1120	0.17	29839
8	48	26630	0.79	167810	1.00	213120	8.8	0	0	0	177493	1344	0.16	34275
8	56	31044	0.77	191400	1.00	248440	8.1	0	0	0	208185	1568	0.16	38680
8	64	17804	0.85	120542	1.00	142520	8.5	0	0	0	116168	896	0.18	25449

Table B.29: Execution statistics for matrix multiply on 1–8 processors with increasing interleaving level (2-way combining, queue size 4, wait-buffer size 64, equal module clock factors).

prcrs	prcs	clks	putil	insts	nutil	mreqs	mlat	rqcon	wbfull	aqcon	ldcom	f&acom	mutil	maccs
16	16	40090	0.71	454848	0.83	531440	10.0	0	0	0	464601	1920	0.10	64899
16	24	37408	0.83	499678	1.00	598688	10.1	0	0	0	527495	2070	0.12	69108
16	32	33210	0.86	454816	1.00	531536	10.1	0	0	0	464686	1920	0.12	64915
16	40	41566	0.82	544032	1.00	665232	10.3	0	0	0	589550	2400	0.11	73267
16	48	49870	0.79	632624	1.00	798112	10.1	0	0	0	713651	2880	0.10	81566
16	56	58119	0.77	720686	1.00	930080	10.1	0	0	0	837154	3090	0.10	89821
16	64	33212	0.86	454752	1.00	531568	10.1	0	0	0	464705	1920	0.12	64926
32	16	42961	0.66	904192	0.77	1054810	12.1	0	0	0	952379	3968	0.07	98435
32	24	37409	0.83	999360	1.00	1197470	12.0	0	0	0	1090260	4278	0.09	102902
32	32	33211	0.86	909632	1.00	1063140	12.1	0	0	0	960429	3968	0.09	98708
32	40	41567	0.82	1088060	1.00	1330530	12.2	0	0	0	1218480	4960	0.08	107060
32	48	49871	0.79	1265250	1.00	1596290	12.0	0	0	0	1474950	5952	0.07	115359
32	56	58120	0.77	1441340	1.00	1860190	12.1	0	0	0	1730160	6386	0.07	123613
32	64	33213	0.86	909504	1.00	1063200	12.1	0	0	0	960479	3968	0.09	98720
64	16	91108	0.60	3512320	0.70	4075520	14.0	0	0	0	3741680	8064	0.06	325759
64	24	72185	0.84	3875640	1.00	4620610	14.0	0	0	0	4277570	8694	0.07	334282
64	32	63973	0.86	3525310	1.00	4095100	14.0	0	0	0	3760900	8064	0.08	326076
64	40	80316	0.82	4222510	1.00	5140990	14.2	0	0	0	4788560	9954	0.07	342418
64	48	96340	0.80	4905970	1.00	6166590	14.3	0	0	0	5797500	10584	0.06	358446
64	56	112387	0.78	5590200	1.00	7193540	14.1	0	0	0	6808130	10836	0.05	374504
64	64	64005	0.86	3525760	1.00	4097020	14.0	0	0	0	3762800	8064	0.08	326098

Table B.30: Execution statistics for matrix multiply on 16–64 processors with increasing interleaving level (2-way combining, queue size 4, wait-buffer size 64, equal module clock factors).

prcrs	prcs	clks	putil	insts	nutil	mreqs	mlat	rqcon	wbfull	aqcon	ldcom	f&acom	mutil	maccs
1	16	129047	0.84	107827	1.00	129058	2.0	0	0	0	0	0	1.00	129058
1	24	148318	0.82	122217	1.00	148328	2.0	0	0	0	0	0	1.00	148328
1	32	140885	0.83	116693	1.00	140895	2.0	0	0	0	0	0	1.00	140895
1	40	174433	0.81	141736	1.00	174443	2.0	0	0	0	0	0	1.00	174443
1	48	207989	0.80	166792	1.00	207999	2.0	0	0	0	0	0	1.00	207999
1	56	241386	0.79	191746	1.00	241396	2.0	0	0	0	0	0	1.00	241396
1	64	165036	0.82	134789	1.00	165046	2.0	0	0	0	0	0	1.00	165046
2	16	167311	0.83	278586	1.00	333528	5.5	695	0	0	48534	6003	0.83	278989
2	24	191833	0.82	314851	1.00	382086	6.5	1600	0	0	61728	9033	0.81	311322
2	32	182302	0.83	300899	1.00	363308	6.5	1316	0	0	62379	8895	0.80	292031
2	40	225735	0.81	364811	0.99	448924	7.0	2566	0	0	81246	14181	0.78	353494
2	48	269541	0.80	429294	0.99	535283	6.5	3819	0	0	100912	22098	0.76	412271
2	56	312046	0.79	493051	0.99	620630	6.5	3482	0	0	119456	30480	0.75	470692
2	64	213352	0.81	347031	1.00	424869	6.5	1855	0	0	81915	14629	0.77	328323
4	16	224094	0.78	698142	0.93	835885	8.0	2037	0	184	209482	22941	0.67	603456
4	24	242964	0.80	781347	0.97	947336	12.2	52748	0	234642	312268	45908	0.61	589148
4	32	235127	0.80	751712	0.96	907373	14.2	101729	0	322567	356031	45037	0.54	506294
4	40	290122	0.78	905296	0.96	1113150	12.5	130121	0	356788	421265	75291	0.53	616579
4	48	345798	0.77	1060940	0.96	1321610	14.8	164733	0	429334	511736	114909	0.50	694959
4	56	399079	0.76	1215090	0.96	1527840	15.2	192898	0	498118	606753	153845	0.48	767230
4	64	274863	0.79	864770	0.96	1058220	13.0	136129	0	393276	456425	73928	0.48	527851
8	16	301313	0.71	1713640	0.85	2052490	9.4	2425	0	369	739171	70671	0.52	1242630
8	24	299613	0.79	1899560	0.96	2301560	15.8	80442	0	455536	918389	129649	0.52	1253490
8	32	302135	0.77	1852310	0.93	2237540	19.8	541383	0	1145020	1031340	136384	0.44	1069790
8	40	375373	0.73	2189950	0.90	2689930	18.4	945186	0	1398300	1247910	225501	0.41	1216490
8	48	443481	0.72	2545410	0.89	3165110	25.8	1168660	0	1624010	1512430	327256	0.37	1325390
8	56	508063	0.71	2901930	0.90	3640710	21.6	1335460	0	1846360	1792200	430179	0.35	1418300
8	64	361326	0.74	2131780	0.90	2610370	21.0	1008420	0	1514360	1377750	219286	0.35	1013300

Table B.31: Execution statistics for bitonic on 1–8 processors with increasing interleaving level (2-way combining, queue size 4, wait-buffer size 64, equal module clock factors).

prcrs	prcs	clks	putil	insts	nutil	mreqs	mlat	rqcon	wbfull	aqcon	ldcom	f&acom	mutil	maccs
16	16	394756	0.65	4106010	0.78	4916810	11.0	2829	0	856	2398070	214375	0.36	2304330
16	24	360598	0.80	4590520	0.96	5565770	15.4	66334	0	403491	2992360	351275	0.39	2222090
16	32	362316	0.76	4418500	0.92	5333850	21.5	1171840	0	2398930	2798240	353261	0.38	2182300
16	40	466422	0.70	5225930	0.86	6414890	22.4	3214930	0	3649730	3436110	598856	0.32	2379850
16	48	557377	0.68	6047450	0.84	7512610	23.7	4521480	0	4506750	4152930	856967	0.28	2502650
16	56	630338	0.67	6805350	0.85	8523050	29.5	5282780	0	5059510	4876390	1094680	0.25	2551930
16	64	460719	0.70	5136050	0.85	6291030	35.4	4105320	0	4280350	3804770	590680	0.26	1895530
32	16	507634	0.60	9694680	0.71	11606700	12.9	3891	0	1655	7090480	611457	0.24	3904660
32	24	436147	0.79	10972600	0.95	13317600	14.5	54953	0	146363	8792550	990464	0.25	3534480
32	32	436893	0.75	10495800	0.91	12675600	21.5	2207780	0	4247270	7559660	910955	0.30	4204890
32	40	558101	0.69	12287400	0.84	15073600	26.4	7716780	0	7763670	8895610	1484480	0.26	4693420
32	48	660239	0.66	13922700	0.82	17256400	25.8	12192800	0	10186500	10462300	2021610	0.23	4772300
32	56	741279	0.65	15415900	0.81	19245800	29.5	15029100	0	11641400	12066100	2510280	0.20	4669270
32	64	570212	0.67	12212400	0.82	14965500	26.5	12474000	0	10293800	9881500	1498460	0.20	3585450
64	16	644810	0.55	22585200	0.66	27034700	14.8	6304	0	4098	19029300	1578160	0.16	6427120
64	24	538853	0.74	25600500	0.90	31071300	16.5	87404	0	153943	22756800	2532540	0.17	5781810
64	32	511128	0.75	24537300	0.91	29639100	25.5	3006320	0	6589410	20461900	2255540	0.21	6921470
64	40	648860	0.68	28325500	0.84	34709400	27.4	15720600	0	14570300	22636900	3480000	0.21	8592260
64	48	759099	0.65	31682000	0.81	39186800	30.4	27341800	0	20736600	25769200	4596270	0.18	8821090
64	56	836672	0.64	34299100	0.80	42670200	30.7	35443200	0	24240800	28769300	5463600	0.16	8437080
64	64	685892	0.65	28518800	0.80	34950000	48.1	32604500	0	22819700	24615600	3600530	0.15	6733560

Table B.32: Execution statistics for bitonic on 16–64 processors with increasing interleaving level (2-way combining, queue size 4, wait-buffer size 64, equal module clock factors).

prcrs	prcs	clks	putil	insts	nutil	mreqs	mlat	rqcon	wbfull	aqcon	ldcom	f&acom	mutil	maccs
1	16	29603	0.75	22226	1.00	29614	2.0	0	0	0	0	0	1.00	29614
1	24	41450	0.75	31075	1.00	41460	2.0	0	0	0	0	0	1.00	41460
1	32	48432	0.75	36259	1.00	48442	2.0	0	0	0	0	0	1.00	48442
1	40	61221	0.75	45771	1.00	61231	2.0	0	0	0	0	0	1.00	61231
1	48	73094	0.75	54584	1.00	73104	2.0	0	0	0	0	0	1.00	73104
1	56	85541	0.75	63814	1.00	85551	2.0	0	0	0	0	0	1.00	85551
1	64	96750	0.75	72136	1.00	96760	2.0	0	0	0	0	0	1.00	96760
2	16	75009	0.72	108779	0.96	144660	6.0	4779	0	0	10509	2789	0.88	131361
2	24	90740	0.72	131456	0.96	175099	6.5	6401	0	0	14661	4518	0.86	155917
2	32	109227	0.73	158770	0.97	211673	6.5	6801	0	0	20198	6621	0.85	184852
2	40	129011	0.73	187429	0.97	250417	6.0	7625	0	0	26962	9174	0.83	214280
2	48	151083	0.73	220283	0.97	294436	7.0	7750	0	0	36654	13250	0.81	244531
2	56	170259	0.73	248155	0.98	332434	6.5	8104	0	0	41402	15314	0.81	275716
2	64	195343	0.73	284944	0.98	382095	6.5	8611	0	0	48946	18325	0.81	314822
4	16	72297	0.67	193134	0.89	257019	7.5	8036	0	140	29517	7492	0.76	220003
4	24	85655	0.71	242157	0.94	322902	13.0	28959	0	27183	57293	17848	0.72	247749
4	32	95508	0.71	269629	0.94	360140	17.0	34141	0	35535	67934	22328	0.71	269866
4	40	112709	0.71	318429	0.94	425890	14.2	41107	0	53829	91892	32375	0.67	301608
4	48	133804	0.70	376337	0.94	503848	15.5	53300	0	76312	117318	43264	0.64	343257
4	56	150804	0.71	426346	0.95	571429	17.5	55226	0	88662	140504	53100	0.63	377812
4	64	174838	0.71	493928	0.95	661921	19.0	67682	0	116982	173592	66709	0.60	421604
8	16	89266	0.61	435286	0.81	579484	9.6	7904	0	383	93765	20981	0.65	464720
8	24	97136	0.70	541309	0.93	722109	14.9	66238	0	62719	168441	46160	0.65	507487
8	32	112670	0.69	619228	0.92	827876	23.1	148247	0	153259	221960	68659	0.60	537229
8	40	138360	0.68	755070	0.91	1011180	27.0	217347	0	230553	300230	101036	0.55	609875
8	48	168405	0.68	915066	0.91	1227520	22.8	288299	0	326720	394962	141217	0.51	691310
8	56	190620	0.68	1031630	0.91	1385550	28.0	342846	0	387370	457137	167511	0.50	760867
8	64	218623	0.68	1181080	0.91	1586820	22.0	411834	0	472046	556351	206695	0.47	823743

Table B.33: Execution statistics for Moore on 1–8 processors with increasing interleaving level (2-way combining, queue size 4, wait-buffer size 64, equal module clock factors).

prcrs	prcs	clks	putil	insts	nutil	mreqs	mlat	rqcon	wbfull	aqcon	ldcom	f&acom	mutil	maccs
16	16	119837	0.56	1076880	0.75	1436100	11.0	8352	0	981	369646	91277	0.51	975139
16	24	116661	0.69	1281050	0.92	1712860	17.3	91755	0	93501	574724	153128	0.53	984974
16	32	130769	0.67	1401520	0.90	1875780	21.7	443037	0	472566	682118	197591	0.48	996014
16	40	156270	0.65	1634830	0.88	2192780	27.8	790838	0	718419	839149	258895	0.44	1094690
16	48	191522	0.65	1982440	0.87	2661660	27.9	1115670	0	983375	1106700	365135	0.39	1189760
16	56	210443	0.64	2169650	0.87	2916120	31.3	1264710	0	1076180	1226700	411015	0.38	1278340
16	64	246140	0.64	2518920	0.86	3389490	36.4	1562190	0	1291890	1471950	509754	0.36	1407730
32	16	139805	0.52	2309820	0.69	3080230	12.8	9228	0	1541	964053	204900	0.43	1911210
32	24	134078	0.67	2863420	0.89	3828640	17.9	120067	0	77347	1544940	358238	0.45	1925380
32	32	151982	0.65	3159570	0.87	4229700	26.9	1102040	0	1089920	1822790	445940	0.40	1960870
32	40	201627	0.63	4068710	0.85	5454800	33.7	2471830	0	2049060	2586430	725967	0.33	2142280
32	48	215580	0.62	4257820	0.83	5716470	38.6	3228240	0	2396770	2708510	770942	0.32	2236890
32	56	241643	0.61	4714130	0.82	6335810	35.6	3943110	0	2757650	3067480	902315	0.31	2365870
32	64	280510	0.61	5436310	0.81	7314340	42.1	4761230	0	3239940	3632100	1118320	0.29	2563730
64	16	173839	0.48	5322520	0.64	7107010	14.5	11131	0	3446	2815090	584766	0.33	3707020
64	24	161455	0.64	6583140	0.85	8813260	16.6	162399	0	108073	4227920	971645	0.35	3613530
64	32	187838	0.64	7678330	0.86	10295000	27.8	2216090	0	1995840	5288950	1309210	0.31	3696690
64	40	217039	0.61	8461740	0.82	11355300	36.3	5571610	0	3898590	5956160	1526520	0.28	3872400
64	48	245329	0.59	9316160	0.80	12515200	38.9	8214170	0	5176550	6673770	1787030	0.26	4054120
64	56	276600	0.58	10267400	0.78	13809700	40.3	10663000	0	6153200	7405810	2047150	0.25	4356530
64	64	353274	0.57	12987400	0.77	17481400	39.0	14669500	0	7974740	9806050	2928140	0.21	4746880

Table B.34: Execution statistics for Moore on 16–64 processors with increasing interleaving level (2-way combining, queue size 4, wait-buffer size 64, equal module clock factors).

prcrs	ratio	clks	putil	insts	nutil	mreqs	mlat	rqcon	wbfull	aqcon	ldcom	f&acom	mutil	maccs
2	0.25	6220	0.20	2525	1.00	3110	16.0	0	0	0	1440	32	0.47	1470
2	0.50	3134	0.40	2537	1.00	3136	9.0	0	0	0	1449	32	0.47	1479
2	1.00	1587	0.81	2563	1.00	3180	4.0	0	0	0	1470	32	0.47	1500
2	2.00	1586	0.81	2562	0.50	3174	3.0	0	0	0	1467	32	0.24	1498
2	4.00	1586	0.81	2562	0.25	3172	2.0	0	0	0	1466	32	0.12	1498
4	0.25	10064	0.21	8375	1.00	10064	25.5	0	0	0	7015	96	0.71	7099
4	0.50	5056	0.41	8403	1.00	10120	12.0	0	0	0	7049	96	0.70	7135
4	1.00	2557	0.83	8483	1.00	10244	7.0	0	0	0	7139	96	0.70	7224
4	2.00	2556	0.83	8485	0.50	10232	4.0	0	0	0	7132	96	0.35	7223
4	4.00	2555	0.83	8481	0.25	10224	3.0	0	0	0	7126	96	0.18	7221
8	0.25	10068	0.21	16747	1.00	20136	31.1	0	0	0	16378	224	0.82	16566
8	0.50	5058	0.41	16807	1.00	20256	15.5	0	0	0	16475	224	0.82	16666
8	1.00	2756	0.77	16936	0.92	20453	8.2	0	0	0	16624	224	0.76	16820
8	2.00	2556	0.83	16969	0.50	20472	5.0	0	0	0	16665	224	0.41	16872
8	4.00	2555	0.83	16961	0.25	20448	3.0	0	0	0	16647	224	0.21	16865
16	0.25	17752	0.21	60107	1.00	71008	37.9	0	0	0	61994	480	0.88	62374
16	0.50	8900	0.42	60238	1.00	71248	19.2	0	0	0	62210	480	0.88	62594
16	1.00	5348	0.71	60591	0.84	71790	10.1	0	0	0	62706	480	0.74	63107
16	2.00	4503	0.84	60833	0.50	72096	6.0	0	0	0	63017	480	0.44	63448
16	4.00	4502	0.84	60832	0.25	72048	4.0	0	0	0	62972	480	0.22	63426

Table B.35: Execution statistics for matrix multiply on 2–16 processors with varying network and memory speed ratio (interleaving level 16, 2-way combining, queue size 4, wait-buffer size 64).

prcrs	ratio	clks	putil	insts	nutil	mreqs	mlat	rqcon	wbfull	aqcon	ldcom	f&acom	mutil	maccs
32	0.25	17756	0.21	120219	1.00	142079	45.5	0	0	0	132024	992	0.93	132757
32	0.50	9350	0.40	120446	0.95	142558	23.1	0	0	0	132479	992	0.89	133243
32	1.00	5765	0.65	120320	0.77	142333	12.1	0	0	0	132284	992	0.72	133032
32	2.00	4664	0.81	121441	0.48	143936	7.0	0	0	0	133832	992	0.45	134726
32	4.00	4502	0.84	121665	0.25	144128	4.0	0	0	0	134015	992	0.23	134936
64	0.25	34600	0.20	453498	0.96	530105	53.2	0	0	0	502353	2016	0.91	503730
64	0.50	19214	0.37	453569	0.86	530112	27.0	0	0	0	502361	2016	0.82	503767
64	1.00	11820	0.60	453632	0.70	530431	14.0	0	0	0	502675	2016	0.67	504050
64	2.00	9127	0.78	457216	0.46	535679	8.0	0	0	0	507848	2016	0.44	509549
64	4.00	8400	0.85	458753	0.25	537728	5.0	0	0	0	509888	2016	0.24	511739
128	0.25	38724	0.18	906625	0.86	1059970	61.0	0	0	0	1028060	4064	0.83	1030830
128	0.50	21280	0.33	907009	0.78	1060100	31.0	0	0	0	1028190	4064	0.76	1031110
128	1.00	12853	0.55	907264	0.64	1060860	16.0	0	0	0	1028950	4064	0.63	1031480
128	2.00	9565	0.75	913920	0.44	1070460	9.0	0	0	0	1038510	4064	0.43	1041930
128	4.00	8401	0.85	917505	0.25	1075460	5.0	0	0	0	1043520	4064	0.24	1047230
256	0.25	82664	0.17	3516930	0.77	4085760	69.0	0	0	0	3989730	8160	0.76	3995310
256	0.50	45042	0.31	3517950	0.71	4086270	35.0	0	0	0	3990240	8160	0.69	3996090
256	1.00	26814	0.51	3518460	0.60	4087810	18.0	0	0	0	3991770	8160	0.58	3996340
256	2.00	19395	0.71	3538950	0.41	4118780	10.0	0	0	0	4022700	8160	0.41	4029060
256	4.00	16093	0.86	3540740	0.25	4120060	6.0	0	0	0	4024120	8160	0.24	4031410

Table B.36: Execution statistics for matrix multiply on 32–256 processors with varying network and memory speed ratio (interleaving level 16, 2-way combining, queue size 4, wait-buffer size 64).

prcrs	ratio	clks	putil	insts	nutil	mreqs	mlat	rqcon	wbfull	aqcon	ldcom	f&acom	mutil	maccs
2	0.25	124640	0.20	50635	1.00	62052	25.0	268	0	0	11765	2208	0.77	48076
2	0.50	61920	0.41	50404	1.00	61746	11.5	176	0	0	11571	2066	0.78	48106
2	1.00	30995	0.81	50434	1.00	61790	5.5	130	0	0	11054	1998	0.79	48736
2	2.00	30678	0.41	50114	0.50	61358	3.0	0	0	0	7357	803	0.43	53198
2	4.00	30678	0.20	50114	0.25	61356	2.0	0	0	0	7357	803	0.22	53196
4	0.25	165916	0.20	133021	0.98	162732	50.2	5105	0	39497	58119	10433	0.57	94164
4	0.50	82568	0.40	133400	0.99	163236	20.5	1012	0	19866	53514	9651	0.61	100065
4	1.00	43820	0.77	134694	0.94	164952	7.2	647	0	81	50705	8097	0.61	106141
4	2.00	41159	0.41	134460	0.50	164636	6.0	43	0	29	38037	4733	0.37	121864
4	4.00	41030	0.20	134088	0.25	164124	3.0	0	0	0	39328	4479	0.18	120314
8	0.25	214672	0.20	343547	0.98	420303	50.4	2023	0	45625	190701	28617	0.47	200964
8	0.50	107922	0.40	347999	0.99	426226	20.1	1737	0	1118	200444	28011	0.46	197753
8	1.00	61712	0.70	347929	0.86	426126	9.4	902	0	110	186123	25547	0.43	214438
8	2.00	53034	0.41	346417	0.50	424093	6.0	22	0	21	163779	18396	0.29	241906
8	4.00	52873	0.20	345610	0.25	422992	3.9	0	0	0	161221	17724	0.14	244043
16	0.25	274052	0.20	876150	0.98	1073539	49.1	3363	0	4517	651146	86743	0.31	335610
16	0.50	142840	0.38	875767	0.94	1073004	22.2	1922	0	1194	644043	85087	0.30	343838
16	1.00	84555	0.64	871900	0.79	1067851	10.9	1001	0	221	609077	79606	0.28	379134
16	2.00	67237	0.40	868461	0.49	1063202	6.2	52	0	20	568143	65700	0.20	429331
16	4.00	66258	0.20	866172	0.25	1060128	5.0	0	0	0	561157	63546	0.10	435419

Table B.37: Execution statistics for bitonic on 2–16 processors with varying network and memory speed ratio (interleaving level 16, 2-way combining, queue size 4, wait-buffer size 64).

prcrs	ratio	clks	putil	insts	nutil	mreqs	mlat	rqcon	wbfull	aqcon	ldcom	f&acom	mutil	maccs
32	0.25	355428	0.19	2152450	0.93	2638185	53.6	5016	0	3843	1854829	244695	0.19	538578
32	0.50	191216	0.35	2143840	0.86	2626681	25.8	3188	0	1559	1828017	239112	0.18	559477
32	1.00	113009	0.59	2136580	0.72	2616964	12.8	2114	0	700	1777015	230150	0.17	609728
32	2.00	85410	0.39	2127320	0.48	2604564	7.1	146	0	121	1687663	206929	0.13	709908
32	4.00	81128	0.20	2121030	0.25	2596131	5.7	2	0	0	1683799	201462	0.07	710847
64	0.25	469816	0.17	5153490	0.84	6314727	62.6	7511	0	5518	4806968	627008	0.12	880591
64	0.50	252830	0.32	5145240	0.78	6303694	29.4	4855	0	3176	4777439	620498	0.11	905615
64	1.00	147923	0.54	5123230	0.66	6274294	15.1	3543	0	1515	4703369	605210	0.10	965576
64	2.00	107407	0.37	5109520	0.46	6255852	8.0	529	0	609	4567589	574525	0.08	1113633
64	4.00	97435	0.20	5094680	0.25	6235943	6.0	0	0	0	4554623	562446	0.04	1118815
128	0.25	612328	0.16	12242100	0.77	15005949	66.8	12676	0	11786	11968328	1561420	0.08	1475931
128	0.50	327666	0.29	12190700	0.71	14937229	32.7	9628	0	8131	11901895	1540470	0.07	1494555
128	1.00	190129	0.50	12140300	0.61	14869879	16.5	7403	0	4974	11779443	1513820	0.06	1576413
128	2.00	133568	0.35	12087800	0.43	14799509	9.0	1855	0	2190	11559179	1466070	0.05	1774116
128	4.00	115287	0.20	12053500	0.25	14753499	6.0	20	0	5	11544054	1441460	0.03	1767877

Table B.38: Execution statistics for bitonic on 32–128 processors with varying network and memory speed ratio (interleaving level 16, 2-way combining, queue size 4, wait-buffer size 64).

prcrs	ratio	clks	putil	insts	nutil	mreqs	mlat	rqcon	wbfull	aqcon	ldcom	f&acom	mutil	maccs
2	0.25	101332	0.18	37120	0.98	49605	25.0	1061	0	0	7001	2601	0.79	40001
2	0.50	49838	0.37	36431	0.98	48683	10.5	1157	0	0	6023	2223	0.81	40436
2	1.00	24837	0.73	36118	0.97	48275	6.0	1208	0	0	5609	2052	0.82	40612
2	2.00	25494	0.37	38167	0.50	50992	3.0	0	0	0	2253	653	0.47	48084
2	4.00	25494	0.19	38165	0.25	50988	2.0	0	0	0	2253	652	0.24	48083
4	0.25	98356	0.18	70212	0.96	94225	49.0	5415	0	14505	26243	11132	0.58	56843
4	0.50	50240	0.36	72628	0.97	97499	20.5	2531	0	8398	25666	10753	0.61	61071
4	1.00	26933	0.68	73531	0.92	98693	7.2	1035	0	85	22503	9178	0.62	67004
4	2.00	24681	0.37	73562	0.50	98728	5.5	3	0	0	16217	5506	0.39	77002
4	4.00	24719	0.19	73679	0.25	98880	3.0	0	0	0	16128	5182	0.20	77566
8	0.25	189644	0.18	267692	0.95	358934	55.6	13384	0	23165	103658	35310	0.58	219949
8	0.50	98696	0.35	275922	0.94	370008	20.8	8966	0	2064	105139	34911	0.58	229940
8	1.00	53569	0.62	265898	0.83	356132	9.4	3071	0	200	92491	30659	0.54	232964
8	2.00	48153	0.37	286653	0.50	384276	6.0	39	0	0	81915	26253	0.36	276095
8	4.00	47731	0.19	284771	0.25	381856	4.2	0	0	0	79268	24854	0.18	277729
16	0.25	177484	0.17	487128	0.92	653100	55.1	10262	0	7032	258064	78798	0.45	316199
16	0.50	93968	0.33	493022	0.88	661035	23.1	7044	0	1612	255473	79347	0.43	326177
16	1.00	53860	0.57	491802	0.77	659313	11.1	2338	0	446	237235	74658	0.40	347384
16	2.00	42308	0.36	493972	0.49	662385	6.2	69	0	1	206290	63435	0.29	392638
16	4.00	41492	0.19	494960	0.25	663888	4.9	0	0	0	212062	63933	0.15	387877

Table B.39: Execution statistics for Moore on 2–16 processors with varying network and memory speed ratio (interleaving level 16, 2-way combining, queue size 4, wait-buffer size 64).

prcrs	ratio	clks	putil	insts	nutil	mreqs	mlat	rqcon	wbfull	aqcon	ldcom	f&acom	mutil	maccs
32	0.25	235304	0.16	1204100	0.86	1617640	54.0	10842	0	5552	760958	228385	0.33	628217
32	0.50	122494	0.30	1169400	0.80	1571599	26.1	7369	0	2925	711576	214894	0.33	645057
32	1.00	72971	0.52	1214390	0.70	1633217	12.8	3119	0	999	717016	222967	0.30	693155
32	2.00	53012	0.35	1191970	0.47	1602236	7.1	153	0	101	639044	198343	0.23	764796
32	4.00	52510	0.19	1250280	0.25	1680357	5.8	0	0	0	671212	206899	0.12	802220
64	0.25	291040	0.15	2749490	0.79	3698898	58.4	12450	0	8793	1988878	591095	0.24	1118775
64	0.50	149470	0.28	2632540	0.74	3542751	29.5	7444	0	4984	1864210	555635	0.23	1122756
64	1.00	85871	0.48	2647010	0.65	3564134	14.5	3462	0	1750	1820800	552823	0.22	1190375
64	2.00	63001	0.34	2719920	0.45	3663143	8.0	227	0	195	1780603	553747	0.16	1328719
64	4.00	56452	0.19	2682770	0.25	3613050	6.0	8	0	0	1720866	533647	0.09	1358465
128	0.25	407404	0.14	7042480	0.73	9491915	69.1	14676	0	19932	5657355	1667050	0.17	2167227
128	0.50	194192	0.25	6276920	0.68	8451948	35.0	9623	0	11569	4904876	1399630	0.17	2147160
128	1.00	110503	0.44	6275220	0.60	8452505	17.8	4513	0	6553	4799040	1391390	0.16	2261824
128	2.00	72883	0.32	5990330	0.43	8079171	9.0	521	0	553	4313757	1282280	0.13	2482962
128	4.00	69362	0.19	6582800	0.25	8876026	6.0	82	0	209	4767185	1445580	0.07	2663182
256	0.25	439980	0.12	13569800	0.65	18307736	81.2	23246	0	66977	11470791	3169020	0.13	3667369
256	0.50	267678	0.23	15732700	0.62	21211407	38.0	14532	0	52104	13510368	3911520	0.11	3789062
256	1.00	133299	0.41	13914800	0.55	18781441	20.3	6078	0	23098	11602533	3299950	0.11	3878504
256	2.00	101439	0.31	15922700	0.41	21487377	10.0	787	0	2975	13105543	3919250	0.09	4462158
256	4.00	86602	0.18	16337900	0.25	22054354	6.0	143	0	34	13304195	4025260	0.05	4724791

Table B.40: Execution statistics for bitonic on 32–256 processors with varying network and memory speed ratio (interleaving level 16, 2-way combining, queue size 4, wait-buffer size 64).