A Scalable High-Performance Computing Solution for Networks on Chips

THE ECLIPSE NETWORK-ON-A-CHIP ARCHITECTURE USES A SOPHISTICATED PARALLEL PROGRAMMING MODEL, REALIZED THROUGH MULTITHREADED PROCESSORS, INTERLEAVED MEMORY MODULES, AND A HIGH-CAPACITY INTERCONNECTION NETWORK TO SUPPORT SYSTEM-ON-A-CHIP DESIGNS.

Martti Forsell
VTT Electronics

Future systems on chips (SOCs) will have tens or even hundreds of processing elements. It is not practical to organize these elements as a single processor with many functional units, however, because it is difficult to extract large amounts of instruction-level parallelism (ILP) from a single instruction stream. Moreover, the physical constraints of future SOCs—for example, the relatively long distance between units on opposite edges of the SOC—will reduce functional unit utilization and slow communication between them, independent of the interconnection topology.

As an alternative, I propose organizing the elements as multiple processing resources composed of a processor and its local memory. I then fit each resource to an area in which the units can communicate internally within one clock cycle and exploit thread-level parallelism (TLP), which is much easier to extract than ILP. Recently proposed network-on-a-chip (NOC) schemes aim to solve future SOC architectural and design productivity issues by providing a uniform communication network connecting multiple TLP resources and standardizing the handling of various interresource communication needs.[1–3] Other important motivations for the NOC schemes are

- reusability of existing intellectual property blocks,
- physical-architectural-level design integration, and
- platform-based design methodology.

NOC researchers claim that using a current Message Passing Interface (MPI) programming model and a network (such as 2D mesh or fat tree) for intercommunication instead of a single bus guarantees scalability and ease of use.[1–3] The proposed NOCs will not be easy to program, however, nor will they scale as processing resources increase. This is due to limited interresource communication bandwidth, access-pattern-dependent throughput, inefficient synchronization schemes, inability to hide the latency of the internal network, and poor parallel-computing models, requiring programmers to explicitly handle synchronization, data partitioning, and interresource communication. We can avoid these problems by using a more sophisticated parallel-programming model and novel parallel architecture that maximizes both ILP and TLP.

0272-1732/02/$17.00 © 2002 IEEE

Eclipse

The embedded chip-level integrated parallel supercomputer (Eclipse) is a scalable, high-performance computing architecture for NOCs. An Eclipse consists of multithreaded architecture with chaining (MTAC)[4] processors with dedicated instruction memory modules, highly interleaved data memory modules, and a high-capacity sparse mesh interconnection network (see Figure 1, next page). Because Eclipse's memory system is cacheless, it has no cache coherency problems. An Eclipse's structure is homogenous, simplifying design and making it easier to integrate into a larger SOC. Eclipse features a completely software-based design methodology to support flexibility and general-purpose operation.

Programming model

Eclipse provides an easy-to-program, exclusive-read, exclusive-write (EREW) parallel random-access machine (PRAM)-style[4] programming model with many physical threads. The model offers a uniform shared-data memory with single superstep memory access latency and machine instruction-level synchronous execution. Parallel access to an Eclipse shared memory is limited, allowing at most one memory reference per superstep memory location. (See the "Models in parallel computing" sidebar on p. 49 for a description of the two main parallel computing models—the PRAM model and the message-passing model—and their key programming differences.)

The PRAM programming model lets an Eclipse serve as a single-instruction, multiple-data (SIMD) machine, a multiple-instruction, multiple-data (MIMD) machine, or as a combination of several SIMD and MIMD machines. A programmer can also partition the Eclipse and assign multiple sequential and parallel tasks to separate threads.

Execution in Eclipse occurs in supersteps, which are transparent to the user. During a superstep, each thread of each processor alternately executes an instruction. Instructions can include at most one shared memory reference subinstruction.

MTAC processors

MTAC is a very long instruction word (VLIW) processor architecture specifically designed to realize the PRAM model on physically distributed memory architectures.[4] An MTAC processor consists of *a* arithmetic-logic units (ALUs), *m* memory units, a hash address calculation unit, a compare unit, a sequencer, and a distributed register file of *r* registers (see Figure 1d). MTAC has a VLIW-style instruction set with fixed execution ordering of subinstructions reflecting the chain-like organization of functional units; tools for using a subinstruction result as an operand for the next subinstruction in the chain; and a hardware-assisted barrier synchronization mechanism. An MTAC's regular structure makes it easy to superpipeline, so there is no need for forwarding within a clock cycle.

MTAC supports overlapped execution of a variable number of threads. MTAC implements multithreading as a deep, cyclic, hazard-free interthread pipeline for hiding memory system latency, maximizing execution overlap, and minimizing register access delay. Switching between threads occurs in zero time, because threads proceed in the pipeline only during forward time.

The organization of functional units in MTAC aims to exploit ILP during parallel execution supersteps. Therefore, MTAC functional units are connected as a chain, allowing a unit to use the results of its predecessors in the chain. MTAC features fixed ordering—that is, functional units are ordered according to the average ordering of instructions in a basic block. Two-thirds of the ALUs form the beginning of the chain. Next are the memory units and the remaining ALUs. The compare unit and sequencer form the end of the chain, because comparing and branching always occur at the end of basic blocks.

Memory modules

Eclipse has two types of memory modules—*data memory* and *instruction memory* modules—which are isolated from each other to guarantee uninterrupted data and instruction streams to the MTAC processors. Both module types use deep interleaving to eliminate performance loss due to constantly increasing speed differences between processors and DRAM-based memory banks.

A data memory module consists of *B* memory banks attached to *Q*-slot access queues for incoming memory references and a common queue for outgoing memory replies (see Fig-

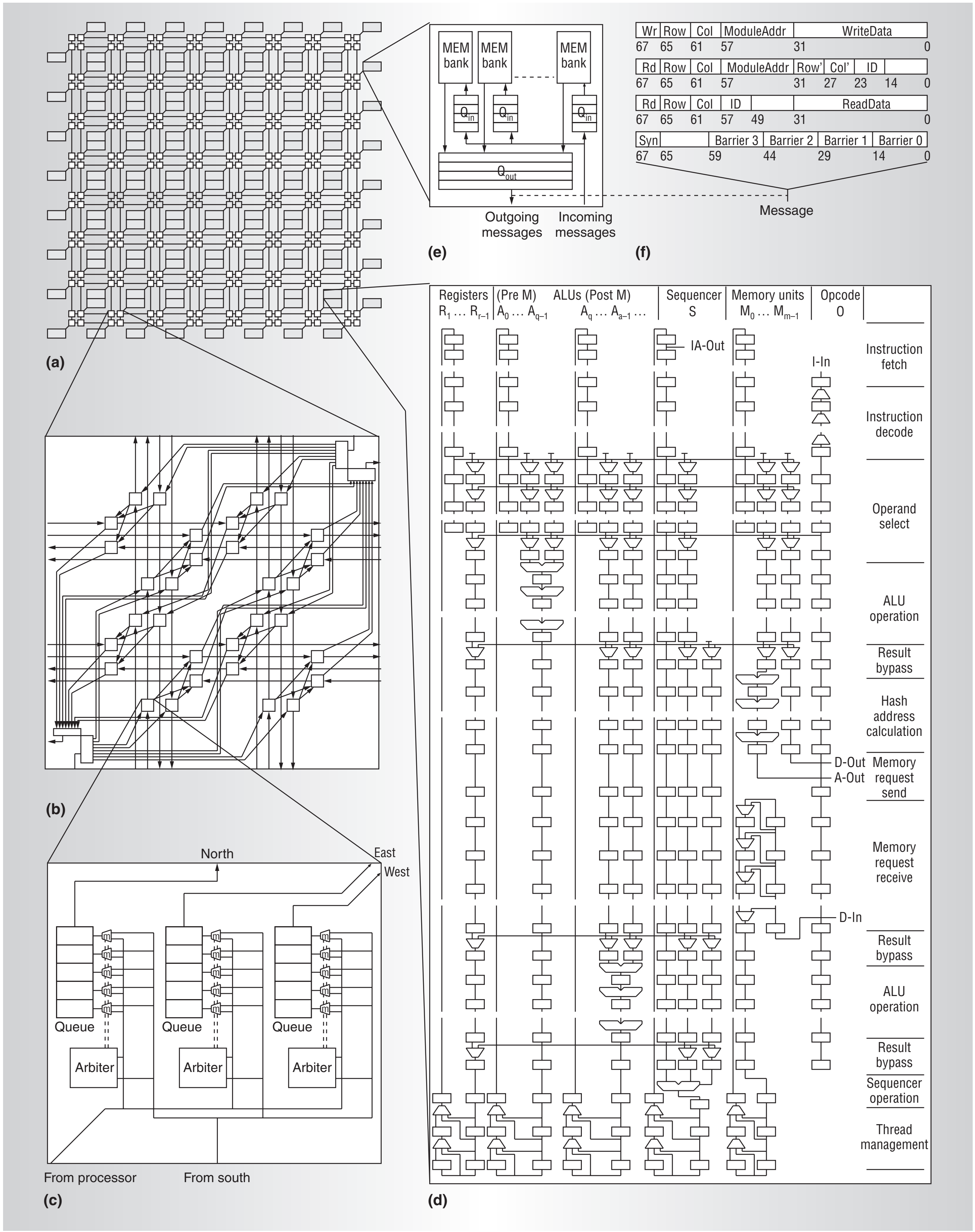

Figure 1. Block diagrams of an Eclipse (a), a superswitch (b), a switch element (c), an MTAC processor (d), and a memory module (e); and message formats for reads, writes, replies, and synchronizations (f).

Models in parallel computing

A parallel computing model is a formal, abstract definition of a parallel computer. It describes the basic components, their properties and available operations, and operation granularity and synchronization. Researchers can use models to analyze an algorithm's intrinsic execution time or memory space while ignoring many implementation issues.

Models

There are two primary parallel computing models—the parallel random access machine (PRAM) model and the message-passing (MP) model. The main difference between these models is in how interaction between processors is organized.

PRAM model

A PRAM is a fine-grained lock-step-synchronous model of parallel computation.[1] It consists of an unbounded set of processors connected to the same clock and shared memory (see Figure A1). All operations, including parallel memory accesses, execute in unit time. Each processor has an ID register with a unique value. A number of memory access variants exist—for example, exclusive write, exclusive read (EREW) and concurrent read, concurrent write. CRCW lets multiple processors access a memory location simultaneously; EREW does not.

Message-passing model

The message-passing model is widely used in parallel computation. A machine using the MP model consists of a set of processors attached to an interconnection network (see Figure A2). Processors communicate by sending messages to each other via a network. Each processor has its own local memory and implements synchronization through message passing.

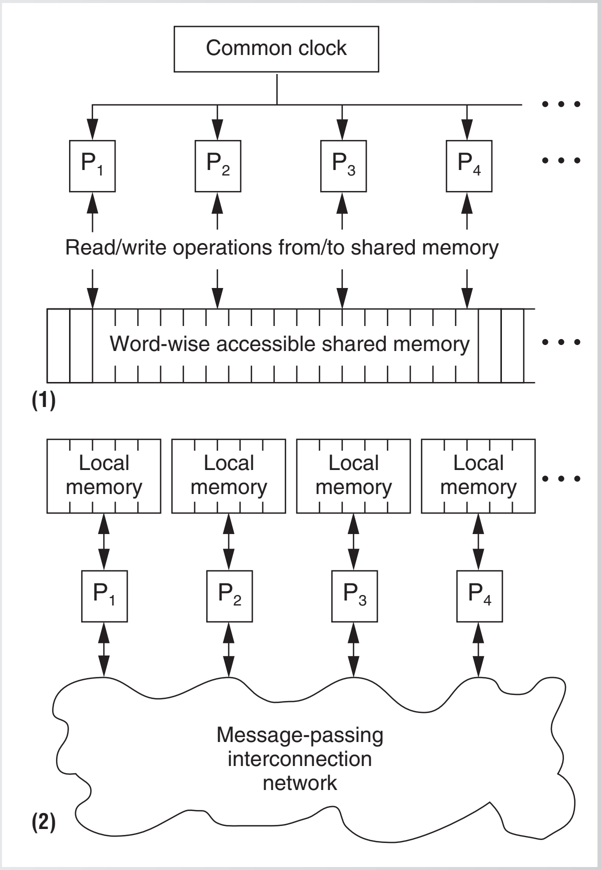
Common clock
P1
P2
P3
P4
Read/write operations from/to shared memory
Word-wise accessible shared memory
(1)
Local memory
P1
P2
P3
P4
Message-passing interconnection network
(2)

Figure A. The PRAM (1) and MP (2) models.

Programming differences

High-level programs written for the PRAM model express interprocessor communication, synchronization, and data partitioning differently from programs written for the MP model. In PRAM, a programmer can simply place data requiring cooperation into the shared memory. In MP, data partitioning and movement must be handled by explicitly inserting partitioning directives as well as send and receive calls to ensure that the right data is in the right place at the right time. Similarly, a PRAM programmer can rely on the model's synchronicity, while an MP programmer must insert explicit synchronization primitives where synchronicity is required. In addition, PRAM algorithmic theory is well known, and there exists a large base of ready-to-use PRAM algorithms.[2] MP algorithms are more difficult to write and their efficiency depends heavily on the underlying MP architecture.

continued on p. 50

ure 1e).[5] For optimum throughput, *B* should be at least the memory bank cycle time divided by the processor cycle time. Unlike ordinary interleaved memories, Eclipse modules use a randomly chosen linear hashing function to map memory locations over a module's banks to minimize message congestion within the module. The idea is to divide a fast stream of processor-generated memory references almost evenly into *B* slow streams, which can be efficiently handled by *B* slow banks.

Two alternative designs for instruction memory modules exist. The *interleaved* alternative is similar to data memory modules except it combines references targeted for the same location.[6] It also requires arming connected MTACs with a pipelined fetch unit, which is similar structurally to an MTAC data memory unit. The *trivial* alternative has a bank for each MTAC thread and works without a pipelined fetch unit.[6] It is very efficient in MIMD-style processing, but wastes a lot of memory in SIMD-style processing because it needs multiple copies of the program. The interleaved alternative keeps only a single copy of each instruction.

continued from p. 49

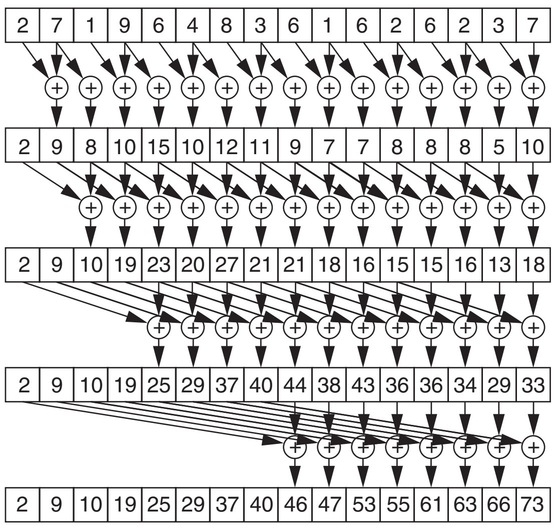

Figure B. A parallel algorithm calculating the prefix sum of a table of integers.

Consider calculating a prefix sum of a table of integers in parallel using the algorithm in Figure B. Figure C shows the PRAM and MP implementations of the algorithm. In the PRAM program (Figure C1), we can conveniently express parallelism using a FOR index: = 1 TO N PARDO <statement>, which executes the statement as N parallel threads so each thread index will have a unique value belonging to range 1..N. In the MP program (Figure C2), explicit SEND, RECEIVE, PARTITION, and SYNCHRONIZE primitives express the required actions.

```
FOR K: = 0 TO ⌈log N⌉ – 1 DO
    FOR J: = 2^K+1 TO N PARDO Table[J]: = Table[J – 2^K] +
    Table[J];
(1)
PARTITION Table[1..N] TO Processors[1..N];
FOR K: = 0 TO ⌈log N⌉ – 1 DO
    IF ID ≤ N – 2^K THEN SEND (ID + 2^K, Table[ID]);
    SYNCHRONIZE(1…N);
    IF ID > 2^K THEN RECEIVE(ID – 2^K, A);
    IF ID > 2^K THEN Table[ID]: = Table[ID] + A;
    SYNCHRONIZE(1…N);
(2)
```

Figure C. The PRAM (1) and MP (2) programs implementing the prefix sum algorithm.

References

1. S. Fortune and J. Wyllie, "Parallelism in Random Access Machines," *Proc. 10th ACM STOC*, ACM Press, New York, 1978, pp. 114-118.
2. J. Keller, C. Kessler, and J. Träff, *Practical PRAM Programming,* John Wiley & Sons, New York, 2000.

Interconnection network

The Eclipse network is a high-bandwidth acyclic variant of a 2D sparse mesh (see Figure 1a) with separate lines for messages from processors to memories, and from memories to processors; two-level switch organization; simple routing; an efficient synchronization mechanism; and randomized hashing of memory locations over the memory modules.

Eclipse's optimized resources—MTAC processors and interleaved memory modules—can easily produce one message per clock cycle. An ordinary 2D mesh network, which some NOC proposals use,[3] does not have enough bandwidth for this heavy communication. For the Eclipse communication architecture, I selected a variant of 2D sparse mesh, in which the number of switches is at least the square of the number of processing resources divided by four, with separate lines for messages going from and returning to processors and memories. This configuration provides enough bandwidth for heavy random communication, and the degree of switches (that is, the number of lines connected to each switch) and interconnection line length are independent of the number of processing resources. Moreover, routing in a mesh easily yields to potentially small switches.

Each switch consists of eight switch elements (see Figure 1c). A switch element is a simple device in which output queues and arbiters route messages. An arbiter detects messages targeted at a nearby queue and checks whether the queue has room for them. To exploit locality, I group the switches related to each resource pair into superswitches (see Figure 1b). This two-level structure lets us send a message from a resource to any of the superswitch switches in a single clock cycle and pipelines switch operation naturally. Logarithmic diameter networks, such as hypercubes and fat trees, provide lower latency;[1,2] however, implementing them in the two dimensions available on silicon requires increasing the interconnection line length proportionally as the number of switches increases. This lowers the communication system's clock rate—an undesirable effect in high-performance systems like Eclipse.

Eclipse uses *slackness of parallel execution* to hide the network and memory module access latency: one processor executes other threads while a thread accesses memory via the network. This works without cache memories or coherency problems. A randomly chosen linear hashing function distributes memory locations around the modules. Investigations and experiments have shown that this kind of hashing can prevent message congestion and hot spots, such that an underlying message-passing machine can time-processor optimally simulate the ideal shared memory of the PRAM model with very high probability.[7–9]

Eclipse emulates the EREW PRAM model shared memory by sending memory requests (reads and writes) and synchronization messages from the processors to the memory modules and vice versa (Figure 1f shows Eclipse message formats). Eclipse routes messages using the simple greedy algorithm with two intermediate targets. The first intermediate target is a randomly chosen switch in a superswitch related to the sending resource. Next, Eclipse routes the message greedily (go to the right row and then go to the right column) to the second intermediate target, which is a randomly chosen switch in the superswitch related to the target resource. Finally, Eclipse routes the message from the second intermediate target to the target resource. Communication deadlocks are not possible because the network is acyclic.

Eclipse uses an advanced *synchronization wave* technique. When a processor has sent all messages belonging to a single superstep, it sends a synchronization message. Synchronization messages from various sources push the read/write messages, spreading to all possible message paths. When a switch receives a synchronization message from one of its inputs, it waits until it has received synchronization messages from all inputs, then forwards the synchronization wave to all of its outputs. The synchronization wave might not bypass any actual messages (and vice versa). When a synchronization wave sweeps over a network, all switches, modules, and processors receive exactly one synchronization message via each input link and send exactly one synchronization message via each output link.

Eclipse also uses synchronization wave to implement multiple simultaneous barrier synchronizations. An MTAC processor determines the number of participating threads in each barrier synchronization during each superstep. It assigns numbers to appropriate fields in the synchronization wave message that it sends at the end of the superstep. As the synchronization messages in the network proceed, switch elements selectively add the numbers in these fields and send the sums forward. By observing the barrier field numbers in returning synchronization waves, the processor can determine when all participating threads have arrived at the synchronization point and allow them to continue execution.

Evaluation

I used detailed analytical performance models and various simulations to evaluate Eclipse.

Analytical performance models

I evaluated Eclipse's performance by analytically modeling the execution time of a parametric benchmark program. To compare Eclipse to a generic NOC architecture, I also defined a baseline NOC architecture, executed a program using the same parameter values on both architectures, and compared the execution times, assuming both architectures are implemented using the same silicon area.

The Eclipse architecture used P_{eclipse} T_{mtac}-threaded, F-functional unit MTAC processors; $S_{\text{eclipse}}=P^2_{\text{eclipse}}/16$ switches; and a baseline NOC based on a 2D mesh network with Q-slot switch buffers and P_{baseline}ARM 9-style processors with data and instruction caches, a local memory bank, and a resource network interface.

I tested the Bench T_p-threaded benchmark program, which I divide into nonoverlapping code portions so that in each portion the distribution (and density) of dependencies between instructions is constant. Assume we can separate these portions into two categories—*independent* and *dependent* parallel. A portion is independent parallel if no data dependencies (such as matrix operations) between threads in the portion exist. A portion is dependent parallel if every read instruction potentially depends on write instructions previously performed by the other threads in the portion—prefix operations and sort, for example. Assume Bench consists of O operations, of which $F_{\text{dp}}O$ belong to dependent parallel portions, and $F_{\text{ip}}O$ belong to independent parallel portions, and $F_{\text{dp}} + F_{\text{ip}} = 1$. Table 1 (next page) summarizes Bench's other parameters. Observe that we can use Bench to model a very large set of computational problems by varying the parameter values.

Assume the baseline NOC clock cycle is D_{cycle} and the memory bank cycle time is D_{mem}. The clock cycle for MTAC processors using the same technology and L_s-stage superpipelining is then

$$(D_{\text{cycle}} + (L_s - 1)\ (D_{\text{fw}} + D_{\text{ld}}))/L_s + D_{\text{bf}}$$

as a function of D_{cycle}, where D_{fw} is the forwarding delay, D_{ld} is the latch delay, and D_{bf} is the balancing term that rounds the pipeline segment execution time

Table 1. Parameters used in the Bench program and their values.

Parameter	Definition	Value
F_{dp}	Fraction of operations belonging to dependent parallel portions	0…1
F_{ip}	Fraction of operations belonging to independent parallel portions	0…1
O	Number of operations	10^{12}
P_{alg}	Probability that a read is unaligned or requires shifting	0.2
P_{b}	Probability that an instruction is a branch	0.2
P_{cm}	Probability that a memory reference misses cache	0.25
P_{m}	Probability that an operation is a memory reference	0.3
P_{rd}	Probability that an instruction is read	0.2
P_{riu}	Probability that the result of a read is immediately used by the next instruction	0.4
T_{p}	Number of threads	10^{6}
U_{f}	Utilization of functional units (this also depends on the architecture)	0.6
D_{cycle}	Baseline NOC clock cycle time	1,000 ps
D_{fw}	Forwarding delay	100 ps
D_{ld}	Latch delay	100 ps
F	Number of functional units in MTAC	8
T_{mtac}	Number of threads in MTAC	512

up to the next gate delay multiple. Assume also that each Bench operation translates to both a single MTAC subinstruction and a single ARM 9 instruction.

Practical studies[9] show that the average number of clock cycles needed for two-way communication or synchronization, denoted here by C_{sync}, in a 2D mesh is $4C_h\sqrt{S}$, where C_h is the average number of processor clock cycles per hop and S is the number of switches. In this case, the average nonlocal memory reference takes

$$C_{mem} = 4C_h\sqrt{S} + \left\lceil \frac{D_{mem}}{\left(\frac{\left(D_{cycle} + (L_s - 1)(D_{fw} + D_{ld}) \right)}{L_s} + D_{bf} \right)} \right\rceil$$

clock cycles in Eclipse. The baseline NOC can use the same equation by assigning $L_s = 1$.

To realize the PRAM model, Eclipse executes all instructions in a single clock cycle and provides true scalability, if we consider threading utilization. Earlier evaluations show that an MTAC processor also provides performance scalability with respect to the number of functional units if there are fewer than 19.[4] Then, Bench's execution time in Eclipse is

$$T_{eclipse} = \frac{O\left(\frac{D_{cycle} + (L_s - 1)(D_{fw} + D_{ld})}{L_s} + D_{bf} \right)}{P_{eclipse} U_t F U_f},$$

where U_f is the utilization of functional units and U_t is the utilization of multithreading, as defined by the equation

$$U_t = \frac{\min(T_p, T_{mtac})}{\max(C_{mem}, T_{mtac})}.$$

I assume that the baseline NOC can execute independent parallel portions locally, although this might require moving large amounts of data between portions to obtain the optimal data partitioning for local execution. The ARM 9 pipeline's inefficiency means that cache misses, which occur with probability $P_m P_{cm}$ and cause delays of C_{cm} cycles, affect local execution. Reads, whose results are immediately used as operands, cause a single clock-cycle delay in the execution of both code portions. Similarly, branches and reads needing aligning or shifting cause two-cycle delays. The probabilities for these occurrences are $P_{rd}P_{riu}$, P_b, and $P_{rd}P_{alg}$, respectively. Thus, I obtain that Bench's execution time in the baseline NOC is at least

$$T_{\text{baseline}} = O\,D_{\text{cycle}}\Big(F_{\text{ip}}\big(1 + P_{\text{m}}P_{\text{cm}}C_{\text{cm}} + 2P_{\text{b}} + P_{\text{rd}}\big(P_{\text{riu}} + 2P_{\text{alg}}\big)\big) + F_{\text{dp}}\big(1 + P_{\text{m}}\big(C_{\text{mem}} + C_{\text{sync}}\big) + 2P_{\text{b}} + P_{\text{rd}}\big(P_{\text{riu}} + 2P_{\text{alg}}\big)\big)\Big) / P_{\text{baseline}}$$

To balance the comparison, I assume that an Eclipse resource pair and a baseline NOC resource take equally large amounts of silicon area. By counting the number of interconnection lines, I estimate that a switch in Eclipse takes four times the area of a switch in the baseline NOC. If a resource is K^2 times larger than a baseline NOC switch, the number of resources in the baseline NOC expressed as the function of the number of resources in the Eclipse is

$$P_{\text{baseline}} = P_{\text{eclipse}}\left(\frac{1+\frac{1}{K}\sqrt{\frac{P_{\text{eclipse}}}{4}}}{1+\frac{1}{K}}\right)^2$$

I compared the Eclipse architecture to the baseline NOC architecture by calculating Bench's execution time as a function of the memory bank cycle time D_{mem}, the area constant K, the number of clock cycles per hop C_{h}, the level of superpipelining L_{s}, the number of resources in Eclipse P_{eclipse}, and the fraction of dependent parallel portions F_{dp} using the values shown in Table 1. These parameters represent properties of current components and technology. I based the parameters on Semiconductor Industry Association technology roadmaps and quantitative research experiments on switches,[1] basic reduced-instruction-set computers (RISCs), and MTAC processors.[4] Figure 2 (next page) shows the results of this comparison.

The comparison shows that Eclipse provides up to two decades better performance than the baseline NOC. As expected, Eclipse's performance seems relatively independent of memory speed, switch delay, and fraction of dependent parallel portions. Increasing the level of superpipelining or the number of processors enhances Eclipse's performance, while only increasing the number of processors enhances the baseline NOC.

Simulations

I performed preliminary system-level simulations and more thorough subsystem-level simulations, which I report in more detail elsewhere.[4–6]

First, I evaluated the potential ILP exploitation capability of various MTAC processor configurations with an ideal memory system.[4] The results were good: An MTAC with four functional units runs a suite of simple integer benchmarks 2.7 times faster than a basic five-stage pipelined RISC processor with four functional units. A six-unit, 10-unit, and 18-unit MTAC processor performed 4.1, 8.1, and 16.6 times faster than the four-unit RISC processor, respectively. The functional unit chaining improved the exploitation of raw ILP to the level where the achieved speedup corresponds to the number of functional units in a processor if the number of functional units is fewer than 19.

Second, I evaluated various data and instruction memory module organizations using memory reference patterns extracted from real parallel programs and the specint92 suite.[5,6] Eclipse's data memory module architecture performed well, providing overheads as low as 6 percent. An exception was in random pattern tests, where interference with the hashing function caused the overhead to climb to 55 percent, even assuming a huge off-chip memory latency of 90 clock cycles and 16-slot queues. The trivial alternative of the instruction memory module architecture performed ideally, but suffered from 7.5-fold memory consumption in the MIMD case; the interleaved alternative featured overheads of 0.2 percent and 21 percent, again assuming huge off-chip latencies of 60 and 120 clock cycles, respectively, and 16-slot queues. Thus, Eclipse's memory module organizations gave the best practical results, hiding the speed difference between processors and memory banks efficiently even with relatively short queues.

Third, I conducted a set of experiments to ensure Eclipse's practical overall performance. I measured execution time and average utilization of functional units for five commonly used parallel-computing primitives, both in an Eclipse and in a theoretical EREW PRAM machine, assuming similar configurations and the same instruction set. Figure 2 (next page) shows the experimental results as execution time, cost of EREW PRAM simulation, and utilization of functional units. As

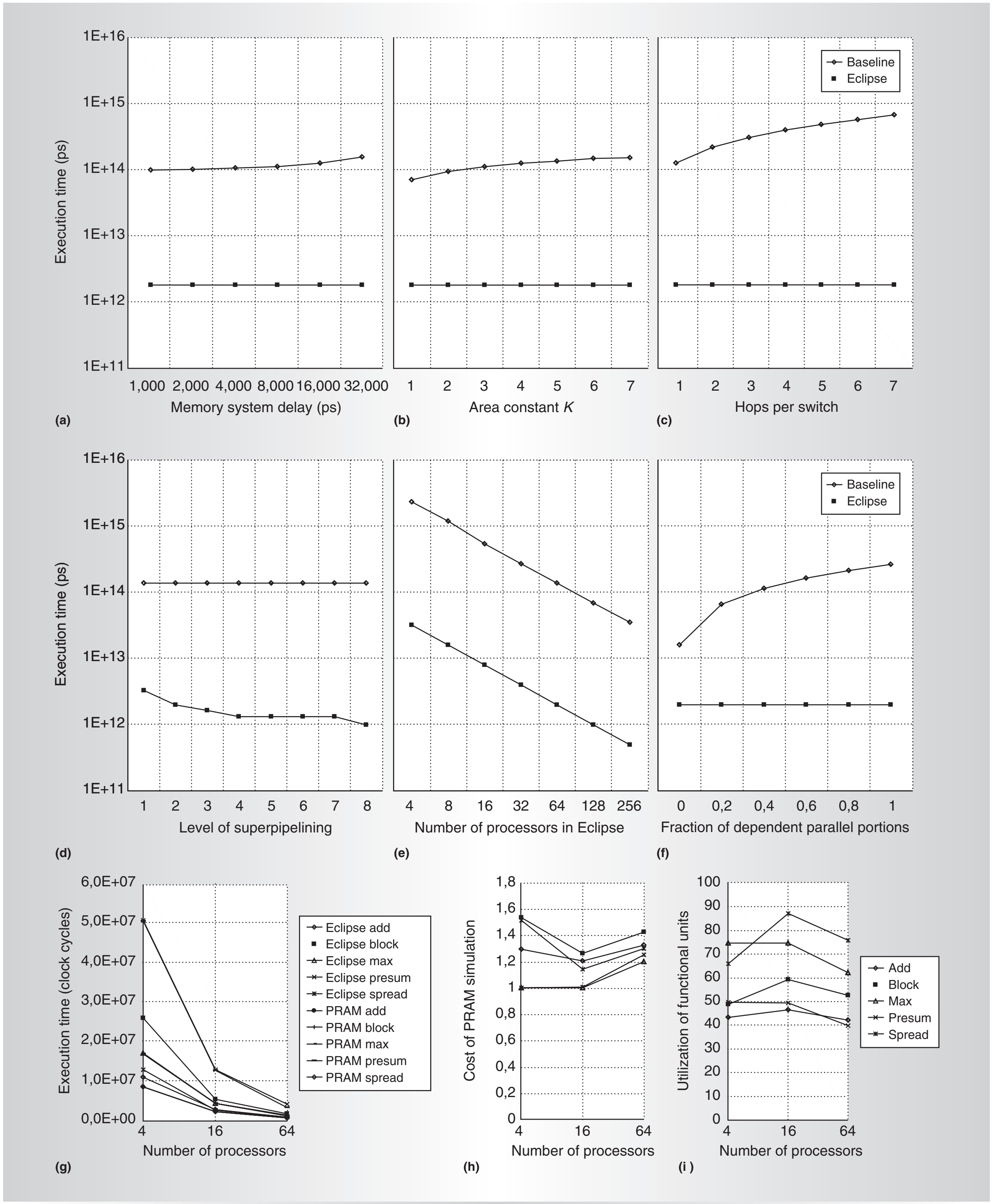

Figure 2. Results of the comparison of an Eclipse and a baseline NOC. The parameter values (except those shown on *x*-axes) are D_{mem}=16,000 picoseconds, K = 3.7, C_h = 1, L_s = 2, $P_{eclipse}$ = 64, and F_{dp} = 0.5 (a – f). Simulation results for 4, 16, and 64-processor Eclipse and PRAM as the execution time (g), the cost of PRAM simulation (h), and the utilization of functional units (i), assuming that D_{mem}/D_{cycle} = 16, C_h = 1, T_{mtac} = 512.

expected, execution time scaled almost ideally. The 16-processor Eclipse was 4.5 times faster than the four-processor Eclipse, and the 64-processor Eclipse was 15.3 times faster than the four-processor Eclipse.

The average cost of PRAM simulation was 1.22. In general, because I used a constant value for T_{mtac} rather than values scaled from the latency, the cost increased slightly as the number of processors increased. The cost in the four-processor case was, however, higher than in the 16-processor case, because a smaller number of communication lines is more sensitive to uneven linear hashing functions. The simulation cost was much lower than 2.20, the cost of the baseline NOC simulating the sequential computing model—the random access machine. I can reduce the cost of PRAM simulation in Eclipse by increasing the number of hardware threads or switches per processor. The average utilization of functional units was 58.2 percent, which is in line with the earlier ILP measurements[4] and is again much better than typical ARM 9 utilizations, which lie around 25 percent.

Eclipse clearly provides a novel approach to SOC design, avoiding the pitfalls of other recently proposed NOC schemes. Most of the functionality that traditionally required dedicated logic should be implemented as parallel software, providing flexibility and ease of use. Furthermore, the proposed architecture promises to bring easy-to-use, truly scalable high-performance computing to chip-level designs.

Future work includes more thorough simulations, minimum clock cycle and area estimations, a compiler environment, and a prototype. I will also investigate architectures for implementing concurrent-read, concurrent-write (CRCW) PRAM on-chip to allow future SOCs to trade even more hardware for parallel software. An MTAC processor already provides this functionality. MICRO

Acknowledgments

This work was supported by the Teknologian edistämiskeskus (the National Technology Agency of Finland), Verket för Innovationssystem (the Swedish Agency for Innovation Systems), Nokia, Ericsson, VTT Electronics, and Spirea.

References

1. P. Guerrier and A. Greinier, "A Generic Architecture for On-Chip Packet-Switched Interconnections," *Proc. Design, Automation, and Test in Europe* (DATE), IEEE CS Press, Los Alamitos, Calif., 2000, pp. 250-256.
2. L. Benini and G. De Micheli, "Networks on Chips: A New SoC Paradigm," *Computer*, vol. 35, no. 1, Jan. 2002, pp. 70-78.
3. S. Kumar et al., "A Network on Chip Architecture and Design Methodology," *Proc. IEEE Computer Soc. Ann. Symp. VLSI* (ISVLSI), IEEE CS Press, Los Alamitos, Calif., 2002, pp. 117-124.
4. M. Forsell, "MTAC: A Multithreaded VLIW Architecture for PRAM Simulation," *J. Universal Computer Science*, vol. 3, no. 9, 1997, pp. 1037-1055.
5. M. Forsell and V. Leppänen, "Memory Module Structures for Shared Memory Simulation," *Proc. Int'l Conf. Advances in Infrastructure for Electronic Business, Science, and Education on the Internet*, Scuola Superiore G. Reiss Romoli (SSGRR), L'Aquila, Italy, 2002, pp. 1-12.
6. M. Forsell, "Cacheless Instruction Fetch Mechanism for Multithreaded Processors," *World Scientific and Eng. Academy and Soc.* (WSEAS) *Trans. Comm.*, vol. 1, no. 1, 2002, pp. 150–155.
7. J. Keller, C. Kessler, and J. Träff, *Practical PRAM Programming*, John Wiley & Sons, New York, 2000.
8. M. Dietzfelbinger et al., "Dynamic Perfect Hashing: Upper and Lower Bounds," *SIAM J. Computing*, vol. 23, no. 4, 1994, pp. 738-761.
9. V. Leppänen, "Studies on the Realization of PRAM," diss. 3, Turku Centre for Computer Science, Univ. of Turku, Finland, 1996.

Martti Forsell is a senior research scientist at VTT Electronics, Oulu, Finland. His research interests include processor and computer architectures, parallel computing, performance analysis, and compiler technology. Forsell has a PhD in computer science from the University of Joensuu, Finland.

Direct questions and comments about this article to Martti Forsell, VTT Electronics, Box 1100, FIN-90571 Oulu, Finland; Martti.Forsell@VTT.Fi.