P-119

Content Addressable Memory Project
NASA NAG-2 668
Semiannual Progress Report (Sep91-Feb92)

Rutgers University
New Brunswick, NJ 08903

J. Hall (PI) S. Levy (PI) D. Smith K. Miyake
Laboratory for Computer Science Research
Department of Computer Science

March 1,1992

1 abstract

A parameterized version of the tree processor has been designed and tested (by simulation). The leaf processor design is 90% complete. We expect to complete and test a combination of tree and leaf cell design in the next period. Work has been proceeding on algorithms for the CAM, and once the design is complete we will begin simulating algorithms for large problems.

In the last 6 months we have produced four publications that describe various components of our research. They are summarized below.

- J. Storrs Hall, Donald E. Smith, and Saul Levy, **The Practical Implementation of Content Addressable Memory**, LCSR-TR-179, Laboratory for Computer Science Research, Rutgers University, March 92. This was also submitted to the 1992 Frontiers of Massively Parallel Computation Conference.

  LCSR-TR-179 presents a functional description of our CAM architecture and discusses attributes(e.g., density, scalability, data-path width, and coupling) that determine the effectiveness of such architectures. In addition, two examples are presented that demonstrate the use of CAM-based algorithms.

- Donald E. Smith, Keith M. Miyake, and J. Storrs Hall, **Design of a LEAF cell for the Rutgers CAM Architecture**, LCSR-TR-180, Laboratory for Computer Science Research, Rutgers University, March 92.

  LCSR-TR-180 presents a specification of the LEAF cell and its interfaces to other modules in the CAM architecture. It describes the four communicating processors which compose each LEAF cell (i.e., k-bit, 1-bit, IO, and memory) and their respective interfaces.

- Keith M. Miyake, Donald E. Smith, **Circuit Design Tool User's Manual**, LCSR-TR-181, Laboratory for Computer Science Research, Rutgers University, March 92.

  LCSR-TR-181 describes the design tool we have implemented in support of CAM research. The design tool is written for a UNIX software environment and supports the definition of digital electronic modules, the composition of modules into higher level circuits, and event-driven simulation of the resulting circuits. Our tool provides an interface whose goals include straightforward but flexible primitive module definition and circuit composition, efficient simulation, and a debugging environment that facilitates design verification and alteration.

  The unique architectural aspects of the Rutgers CAM uses many of the features *typical* to most design tools; however, it also requires some features that are not widely supported. Our design makes use of many similar, but not identical, modules which puts a premium on design tools that support parameterized modules (e.g., generic entities in VHDL) and strong typing of a module's ports. In addition, since our design is continually evolving, the quality of and control over error handling, in both the design as well as the simulation phase, is very important. In looking for a design tool to support

our research we found that either some critical features were inadequately supported or the tool was much more than we required. In response to this environment we implemented a prototype design tool that supports exactly the features required by the design of the CAM.

- S. Wei and S. Levy, **Design and Analysis off Efficient Hierarchical Interconnection Networks**, LCSR-TR-167, Laboratory for Computer Science Research, Rutgers University, September 91. A shorter version of this paper was published as: S. Wei and S. Levy, **Design and Analysis off Efficient Hierarchical Interconnection Networks**, Proceeding of 1991 Supercomputing Conference, Nov. 91, pgs 390-399.

  LCSR-TR-167 presents a new approach to message-passing architectures based on the general idea of hierarchical interconnects. The approach chooses the appropriate number of interface nodes and clusters based on performance and cost-effectiveness. The report includes both static and queueing analyses of such networks.

March 92

THE PRACTICAL IMPLEMENTATION OF CONTENT ADDRESSABLE MEMORY*

J. Storrs Hall, Donald E. Smith, and Saul Levy

LCSR-TR-179

Laboratory for Computer Science Research
Hill Center for the Mathematical Science
Busch Campus, Rutgers University
New Brunswick, New Jersey 08903

*This work was supported by the Defense Advanced Research Projects Agency and the National Aeronautics and Space Administration under NASA-Ames Research Center grant NAG 2-668.

The Practical Implementation of Content Addressable Memory

J. Storrs Hall
Don Smith
Saul Levy
The Rutgers CAM Project[1]
Laboratory for Computer Science Research
Rutgers University

Keywords

content addressable memory, associative processors, SIMD, processor arrays

Abstract

The notion of using content addressable memory (CAM) to achieve massively parallel processing has resurfaced regularly since it first appeared in the 1960's, but has consistently failed to produce cost-effective general-purpose systems. An analysis of this situation reveals a number of specific design pitfalls regarding memory density, scalability, datapath width, and processor coupling. Once these are avoided, specific functionalities must be included in the design. This paper details the pitfalls and presents an architecture which avoids them. Further, guidelines are developed for estimating CAM's effectiveness as a parallel processor.

[1] Supported by DARPA and NASA under NASA-Ames grant NAG 2-668

Introduction

"Pure" content addressable memory (CAM), such as is used in cache lookup, is a method of addressing where each word of memory has a variable address, explicitly stored with the word. With every memory access, the desired address is compared with the stored address in every word. We are concerned here with an extension of this concept which forms the basis for a method of massively parallel processing. It dates back at least to Falkoff[62] and has been called many things, including content addressable parallel processing (Foster[76]) and associative computing (Potter[88]). We shall use CAM as a generic term, to include parallel processing, and will refer to "pure CAM" if it is necessary to distinguish the simple form.

This paradigm is well explained in Foster[76]. The broadcast value, instead of being compared strictly with the address portion of the word, is compared with the entire word (with a mask to provide "don't care" bit positions). The result of the comparison, rather than immediately causing a read or write of a matching word, is stored in an explicit "response bit". The bit is then used to control subsequent read/write operations; in particular, more than one word can be written into simultaneously. Boolean functions are then synthesized from sequences of tests, and bit-serial aithmetic can be performed on all words in parallel.

At a higher level, the model allows for logic, comparisons, and arithmetic between some global value and a local value stored in each word, or between local values in each word. Individual words may refrain from the operations based on locally determined conditions. This results in an architecture which is equivalent to a SIMD star network, with the CPU as the hub and each word as a leaf processor. Since the hardware in the memory is only a few gates in addition to a flip-flop at each bit, CAM should, or so the theory goes, form the basis for massively parallel processors at densities near those of static RAMs.

We examine two questions: (a) can CAM (or some mechanism that implements the CAM computational model) really be built within a small constant factor in cost of RAM, and (b) if so, how efficiently can it be used?

Typical implementations of CAM

This section explains how, starting from the basic idea of CAM, a designer might ultimately end up with any of a number of existing processor array architectures. This is not to be taken to imply that any of those systems were designed that way, nor indeed had the CAM model in mind at all. It does, however, fairly reflect the authors' own attempts to find a cost-effective realization of the paradigm.

The first problem the CAM designer meets is that CAMs useful for parallel processing require very wide words–256 bits is not unreasonable. Furthermore, each bit position requires a data line and a mask line in the bus, doubling its width. Requiring a 512-bit wide bus is not impossible, but the CAM also requires a connection between each bit in any given word, which ordinary memory does not. Thus it is problematical to split CAM memories onto separate chips, requiring the pinouts of each chip to handle the entire bus width.

What is worse, in the process of most CAM operations, a large portion of the bus is wasted. Most comparisons are of smaller fields within a word (selected by the mask lines), and in the case of the bit-serial, word-parallel arithmetic, only one or two bits are being tested or set at a time! Clearly, an optimization can be performed: all the comparators at each bit can be removed, and in their place a one-bit ALU can be added to the word. The "match" daisy chain and the read/write control lines can be replaced with a one-bit local bus running across the word. Now the (global) bus is much smaller: an opcode for the ALU, a bit address which can be decoded on-chip, a one-bit data bus. What is more, arithmetic is faster; single-bit arithmetic operations are built into the ALU rather than being synthesized out of logic operations which are in turn synthesized out of masking and testing.

At this point there is a strong temptation for the designer to split the architecture across chips, having processor chips which connect to standard memory chips. This has the advantage of making memory much less expensive, but the disadvantage of tying the number of words to the pinout of the processor chip. The relationship to the original CAM paradigm begins to become somewhat strained also; this implementation puts a strong downward pressure on the number of words/processing elements, and a strong upward pressure on PE complexity, inter-PE connectivity, etc. In practice, this has been the best tradeoff point for CAM-like implementations; it characterizes the evolutionary niche occupied by the CM-1, the DAP, the MPP, and even the Illiac. Of course these machines were not (necessarily) designed as an implementation of CAM: they are mentioned to illustrate the point in the design space toward which CAM tends to gravitate.

We should mention, as a counterpoint, the STARAN (Batcher[74]), which *was* designed to implement CAM ideas. STARAN consisted of a number of 256-word arrays of 256-bit words. Today a STARAN array could be put on a single chip, but there would still be the bus to contend with. Perhaps predictably, the major thrust in CAM-like architectures in the interim has been along the processor-array lines.

Criteria

Given these facts, it behooves the architect of a CAM-based system to develop a very strong theory of why the CAM paradigm has failed to produce a cost-effective general-purpose architecture. Here is the theory:

- Density: The basic CAM algorithms are based on the assumption that all of the memory in the system is CAM. No implementation to date has come near this. CAM has been a scarce commodity, backed up by RAM, and data is swapped in and out of CAM to be processed. This is deadly, since CAM at best transforms a linear search or other simple loop to a constant-time operation; swapping the data in or out re-introduces the linear time.
- Scalability: For physical practicality, CAM chips must have constant pinout, independent of the number of words per chip. The size of ordinary RAM is an extraordinarily scalable feature of the von Neumann architecture, varying by more than six orders of magnitude over the range of different systems. If CAM is not also scalable in this very strict sense, it will fail to substitute for RAM.
- Datapath width: In choosing a one-bit processor, a CAM implementation gains faster

arithmetic but loses content addressability. If comparisons are bit serial, CAM is *slower* than conventional indexing structures such as binary trees and hash tables. To be usable as CAM, a memory must be able to compare, add, and subtract in a time comparable to a normal memory access.

- Coupling: The overwhelming tendency in designing a SIMD processor array is to place it as an attached processor to some conventional machine which manages the data not being used in the current parallel computation. This is unworkable; the bottleneck means that unless there are large speedups to be gained from operations on relatively long-lived data, the serial host processor can perform most simple associative operations faster than the CAM, when the time to transfer the data to and from the attached processor is taken into account.

Density

Perhaps the most important criterion is density. The CAM must be usable as memory. If this is met, the basic CAM algorithms can be brought into play. Associative retrieval allows arrays of simple, explicitly-indexed records to be used instead of trees, linked lists, hash tables, priority queues, and inverted indices; what is more, it saves the software designer from having to make choices, with their associated tradeoffs, between these structures. (See Hall[81], Potter[88].)

It would be extremely inefficient to use a "true" parallel processor for search operations. The algorithmic speedup is at best $log\ N$, so its efficiency is $\frac{log\ N}{N}$, e.g. 0.002% in the case of a million-word CAM. Luckily, as distinct from "pure" CAM, the parallel processing CAM model produces significantly better speedups for other operations. If a million-word CAM has the same hardware cost as a fully-connected parallel processor with a thousand nodes, the CAM need only achieve an average 0.1% speedup to equal, in operations per cycle, the true parallel processor with perfect 100% linear speedup. (Caveat: A whole-chip processor will almost certainly have faster cycles than a CAM.)

Even so, we would like to keep the CAM to within some small constant factor of RAM cost. While a somewhat speculative analysis indicates that CAM might be rated as having a processing power proportional to the square root of the number of words, it remains the case that in order to do so it must act as the system's primary memory. Thus, its cost must remain low independent of the processing power it provides. This can be accomplished by starting with a high-density DRAM design and only allowing some constant fraction of the chip to be used for the active elements that implement the CAM model.

Scalability

The only interconnection schemes that meet the strict scalability criterion are a bus, a linear array (daisy chain), and a tree. We find that a bus is desireable to distribute instructions to the words; a tree is essential for implementing the collective functions the CAM computational model requires; and a long shift register is probably the best way to implement an asynchronous I/O capability that does not seriously interfere with other operations.

Width

Consider the following design task: you have 1024 full adders and desire to build a machine to compute 1024 evenly spaced points of a linear function $y = mx + b$ in 32-bit fields. You can add whatever control, memory, and communications you like. There are three strategies:

- First, you could make a serial processor using all the hardware to form a circuit that could multiply in one cycle. Then loop computing $mx + b$ at each iteration, taking 2048 cycles. This can be improved by a strength reduction optimization, starting with b and adding m at each iteration, for a total of only 1024 cycles.
- Second, use 1024 1-bit ALUs to compute the result directly, multiplying in each processor mi (where i is the processor number stored as a constant) and adding b, taking $1024 + 32 = 1056$ cycles. This is better than the naive serial algorithm but comparable to the optimized one.
- Third, one can form 32 32-bit ALUs and take advantage of parallelism *and* strength reduction: in a 32-cycle multiply (and a 1-cycle add), form the number $32im + b$ at each processor (where $32i$ has been stored as a constant) and use that as a starting point for adding m for the next 31 points. This gives you a total of 64 cycles.

This example is illustrative of a class of algorithms, which we call *semi-serial* algorithms, for which ALUs of a width commensurate with the data of interest and capable of simple arithmetic and comparison, but not more, form a local optimum in hardware efficiency. Note that for large problems using a fixed algorithm, the three designs above are equivalent: each can do 1024 multiplies in 1024 cycles.

Coupling

Having attained memory-like density in the CAM, we can use it to advantage by the simple expedient of replacing all the system's RAM with CAM. (An alternative approach would be to use a Harvard-like architecture with RAM for instruction memory and CAM for data memory.) It is a long-established trend for processors to be faster than memory and to run asynchronously. In practice, this may mean that special processors are not necessary for CAM-based systems (although it is certainly possible to design them).

In a RAM, where only one word is being accessed at a time, clock skew across the memory is not a serious problem. In a CAM, it might be. If the connectivity is a tree, outgoing information, i.e. from the CPU to the CAM may arrive at the memory in a skewed fashion harmlessly, since no leaf depends on information from any other leaf. However, ingoing information may need to be synchronized. If the ingoing datapaths are combinational, synchronization consists only of waiting the longest number of gate delays from CAM to CPU. If the delay is consistent, this is not only simpler but faster than any other method.

Other Features

To be effective as CAM, a system must not only avoid these pitfalls but be designed with a cognizance of typical CAM algorithms in order to make most efficient use of its hardware. The following is a list of features we have found to confer substantial algorithmic advantages, while remaining implementable within the constraints above:

- Collective functions: The CAM model is virtually useless without a fairly powerful feedback mechanism from the memory to the processor. After an associative search, the processor may need to know how many (if any) matches were found. In the classical CAM model, operations such as summing all active words, finding the maximum or minimum of a set of values, and the like can be done as word-parallel bit-serial algorithms. These operations are crucial parts of the basic CAM computational model.
- Segmentation: The CAM model has the capability of doing in parallel essentially a simple loop, dealing only with local and global values at each point. The ability to segment the CAM, still doing the same operation everywhere but having a different "global" value in each segment, corresponds to doing nested loops, and extends the range of parallel operations significantly.
- Local addressing: The fields of the CAM words which are going to be operated on by an instruction are, in the basic model, the same for each word. The ability to vary the field choice on the basis of local data allows the CAM to do things like regular expression matching or unification in parallel, a prerequisite to the extension of the ideas of content addressability into higher-level models of computation.

The Rutgers CAM

At this point we shift terminological gears; the specifics of the model are enough more complex that the concept of a "word" splits into two separate terms: a "cell" is the locus of one active element and the unit of activity control; the term "word" hereinafter will mean an addressable unit of simple memory whose width is that of the bus and other datapaths. There are many words per cell.

The design criteria elucidated in the preceding sections interact strongly with particular states of technology to determine the viability of a CAM implementation. As a point of departure, let us assign values to the constants as follows: Word width, 32 bits; density, 1K words per ALU. With these parameters we can devote at least half the silicon to DRAM; the density of the CAM *as memory* will be at least half that of conventional memory in the same technology.

A decade ago, the dominant memory technology was 64 K-bit DRAMs. The above constraints would specify a 32 K-bit chip, with one ALU occupying half the space and 1K 32-bit words on the other half, for a total of one cam cell per chip. A one-megabyte memory (typical for mainframes of the day) would have consisted of 256 chips (and therefore 256 cam cells). Assuming the CAM could be driven at 5MHz and do one operation every 5 cycles, the CAM would have represented a 256 mega-ops peak processing rate.

For the mid-90's we can use a 64-Meg DRAM as a basis and obtain 1K cam cells per chip. (Each cell is still an ALU and 1K 32-bit words; the memory still occupies half the chip.) 16 chips would provide 64 megabytes of memory and, assuming a 10-MHz operations rate (from a 50-MHz system clock), a peak 160 giga-ops. This is a single-board computer.

We have developed a "CAM virtual machine" as a common focus for architectural and algorithmic efforts. This model of the CAM reflects the capabilities of active elements which fit the tradeoffs above, i.e. all the circuitry associated with one CAM cell must be roughly the size of 1K words of DRAM.

The CAM model is related to Blelloch's[87] "scan model of computation", but differs in the fundamental regard that the CAM model does not have a general permutation operator and thus is *not* a complete model for parallel computation. It does include:

- Activity control: all the following can be controlled on a per-cell per-instruction basis. This forms the major difference between CAM and simple vector styles of computation.
- Parallel vector operations to include addition, subtraction, comparison, bitwise boolean functions, and some shifting and byte extraction. These operations can only be done between words of the same cell, but they need not be the same words in each cell. It does *not* include multiplication, division, or floating point, although these can be done in software.
- Broadcast: one of the operands in the above operations can be a "global" constant value (the same in each cell).
- Collective functions: scalar-valued collective functions include the sum, max, and min of all the elements of a vector; vector-valued collective functions are parallel prefix (and suffix) forms of the scalar ones; and skip-shifting. This last moves a value from each active cell to the next active cell, no matter how far away, as a unit-time primitive.
- Segmentation: All of the collective functions and broadcasting can be done in segments, which, like the activity, are defineable on the fly. Each segment can have a different "global" value which comes from some cell in the segment.
- Simple one-cell-at-a-time shifting which ignores activity and segment definitions can be done concurrently with other CAM operations; such shifting requires time proportional to distance shifted.

The physical implementation of the CAM model is, as indicated, by way of a set of active elements along with DRAM. Each chip, regardless of the amount of CAM onboard, has 4 busses (making it more like a processor chip in its packaging). These are one bidirectional data bus, one input-only instruction bus, and two I/O busses, one in and one out. 64 pins dedicated to I/O sound extravagant, but in the system as a whole, they are the most heavily used part. Furthermore, this interface is constant; CAM is a scalable architecture in the strongest sense of the word.

- Each CAM cell consists of an ALU with 16 registers, and 1K (or more) DRAM. The ALU, register, memory, and all datapaths are 32 bits wide. The first 4 registers are mapped into the tree, the memory, the shifter, and the collection of one-bit registers that are the status, activity, segment, and so forth; the rest of the registers are general purpose. Each cell is like a very simple RISC with register-to-register operations and asynchronous load/store.
- The cells form the leaves of a tree of simpler ALU's, each of which has one register. The tree is combinational: that is, each CAM cell presents it with a 32-bit value and two control bits, activity and segment. The tree forms a direct-wired circuit that produces the appropriate value at the root and into each tree node's latch. This allows for virtually any possible clock skew between CAM cells–of course, we pay for this by having tree operations take 5 to 10 times as long as local CAM operations.

  In a scan or shift operation, the tree actually does two operations, one up and one down. Each phase is combinational internally.

- A (unidirectional) instruction bus, emanating from the CPU, which controls the cells and the tree nodes. Depending on chip size and process parameters, the bus may be pipelined: the bus is optimized for throughput, in contrast to the tree, which is optimized for latency.
- A shift register for overlapped I/O and data motion between CAM cells. Like the tree and the DRAM, the shift register operates asynchronously from the CAM cell. CAM efficiency is very dependent on its ability to move data. If a loader (see below) can relocate a program in, say, 1000 cycles but then requires a million cycles to move it from CAM to instruction RAM, the CAM is worthless. This is one of the reasons that our architecture has 1K or more words in each cell – loading and unloading of one problem's data while another problem is being worked on is crucial to CAM's efficiency. Indeed our design calls for a separate datapath for this function. This can be something as simple as a (32-bit wide) shift register with one position for each cam cell. It doesn't even have to be true DMA: in our mid-90's model, for example, the I/O shift register would clock data for 1024 cycles before needing one memory cycle to store it.

CAM Algorithms

We present the following algorithms, in a very high-level form, to give a feeling for both the abilities and the limitations of the CAM.

Consider a very commonly used program, the relocating linking loader. Its initial task, finding the appropriate place in memory to put each of a given set of modules, can be as simple as a single parallel prefix sum. Updating the relative addresses at each point of the code to absolute addresses for execution is a local operation in each cell. Resolving global references, however, depends on the number of distinct symbols referenced (not the total number of occurences). This dominates the rest of the process, which is constant time.

For the next algorithm, we will assume that we have a "small" CAM, on the order of 1000 cells; it is intended to be representative of a simulation and visualization task on a machine at the scale of a workstation. We wish to simulate a number of bouncing particles in some three-dimensional space (at the appropriate scale, molecules in a gas) and display the results on a screen in real time. We will assume that there are enough CAM cells to allocate one per particle, and (separately, not in addition) one per pixel for one scan line.

1. [Advance the particles] $X_{new} = X_{old} + V\Delta T$ for each particle. A purely parallel, local operation.
2. [Find collisions] Naively, this is an $O(N^2)$ sequential time operation, but if the number of particles is large enough, a sophisticated implementation would use spatially-oriented indexing schemes to reduce the complexity to $N \log N$ (e.g. octrees). CAM gives us linear time with the naive algorithm, and for the parameters given, that is sufficient. (For larger problems, similar indexing schemes could allow enough extra use of parallelism to reduce the CAM time to $N^{\frac{2}{3}}$).
3. [Simulate collisions] Once the data for each collision has been brought together, the

new velocities for each particle can be computed in a single parallel step.

4. [Display] For each scan line, perform the following steps:
5. [Select objects] Associatively search for each object whose image intersects the current scan line. For each object from farthest to nearest, do step 6:
6. [Draw] Set the value of each pixel the current object intersects on the current scan line. This step could be split into a sequential and a parallel part like steps 2 and 3 if the rendering algorithm is complex enough to warrant it.

This algorithm exemplifies the cases where conventional worst-case asymptotic complexity analysis is inadequate. Constant factors deriving from the necessity of using sophisticated indexing data structures prevent a real-time implementation on a sequential machine at the same range of clock speeds as a CAM.

CAM also has operations with high constant factors, notably numerical calculations. In many cases, fairly simple algorithmic techniques can move these operations out of loops and do them in parallel.

The final algorithm is intended to demonstrate what could be done with a "large" CAM, on the order of a million cells. (This would represent 4 gigabytes of RAM.) This time the task is a low-level part of an image-understanding process, namely to divide a picture up into regions. (E.g., suppose we had a black and white picture of a collection of polka-dots. We would want to associate with each white pixel a region number of 0, meaning that all the background was a single connected region, and with each black pixel a region number indicating which polka-dot it was in.)

1. [Process scan lines] Assuming the picture is in row-major form, use segmentation to do each scan line in parallel. Do short-distance shifts to localize horizontal neighbor information, and create a vector with a 1 for each edge, 0 elsewhere. Do a plus-scan of this vector; the result is a unique region number within each scan line, with each pixel in the region having a copy of the number.
2. [Process one vertical line] For some vertical line, select (with activity control) only those pixels in that line. Perform step 1. for that line (using skip-shifting, etc.). Returning to line-by-line segmentation, combine vertical and horizontal region numbers into a global region number. This finds all pixels co-regional in horizontally contiguous segments touching the chosen vertical line.
3. [Other vertical lines] Which and how many vertical lines need to be processed is a matter of heuristic. This is considerably assisted by associative search, which can be used to skip vertical lines all of whose horizontal segments have been processed by the action of other vertical lines. A bisection method works well. In the best case the number of vertical lines needed will be on the order of the square root of the number of regions. In the worst case it may be the width of the picture, i.e. having to do every vertical line.

If the CAM were, e.g., a mesh-of-trees instead of simply a tree, we could run step 1 vertically as well as horizontally and be done in two steps.[1] Special-purpose architectures

[1] Actually, a rigorous definition of the algorithmic task can make it arbitrarily complex for either architecture, involving long chains of region coalescence fixups. However, as a basis for a low-level input-processing step for vision, identification of relatively simple regions seems adequate.

for vision invariably have such 2-dimensional connectivity. We consider this algorithm to show that the CAM handles this problem as well as a general-purpose machine might be expected to. Even when heuristics fail, its performance degrades only to $\sqrt{N}$.

Evaluating the Processing Power of CAM

We must stress that it is virtually meaningless to compare the peak ops rate in CAM to serial MIPS. The relationship is extremely problem-dependent, and within a given algorithm, data-dependent, as is clearly shown in the preceding algorithms.

Even more important, perhaps, is that CAM cannot be validly compared with typical parallel processors, with their general interprocessor communications ability. The mid-90's technology estimates above provide a 1K-cell CAM on a chip with roughly 128 data pins. These pins would only provide 1-bit-wide datapaths if a mesh architecture were used (i.e. the perimeter of a 32x32 square); a hypercube architecture would be completely impractical.

A more valid basis for comparing CAM to other architectures is perhaps by pin count. In this scheme one CAM chip might be considered equivalent to a processor chip or 8 DRAM chips. Thus it could be appropriate to compare the processing power of a million-cell CAM to a thousand-processor conventional machine, since they would represent about the same amount of hardware.

But ultimately CAM shouldn't be compared to conventional parallel machines built as networks of microprocessors, because the architectures are orthogonal. Each processor of such a conventional parallel machine could be provided with CAM instead of RAM. Such an arrangement combines the best advantages of each part, synergistically. However, it is beyond the scope of this paper.

Summary and Conclusion

Content Addressable Memory is memory. Our criteria of density, scalability, width, and coupling mark the boundary between a powerful memory and an anemic parallel processor. Within these limits, CAM can be a very cost-effective system component if average effective utilization is near even one percent.

CAM algorithms can be both simpler and faster than conventional ones; with more ingenuity, substantially better utilization can be achieved. The Rutgers CAM design provides collective functions and segmentation, allowing fairly sophisticated parallel algorithms in an architecture which still retains half the density of DRAM in a given technology.

Acknowlegements

The authors wish gratefully to acknowlege the substantial contributions of the Rutgers CAM Project staff, Keith Miyake and Sizheng Wei.

References

[AMD88] Advanced Micro Devices, Inc: *AM99C10 Content Addressable Memory data sheets*, AMD, Sunnyvale, CA, 1988

[Bat74] Batcher, K. E., *STARAN Parallel Processor System Hardware*, Nat. Comp. Conf 1974, pp 405-410.

[Bla87] Blair, Gerard M.: *"A Content Addressable Memory with a Fault-Tolerance Mechanism"*, IEEE JSSC, vol SC-22, no. 4, pp 614-616, Aug 1987

[Ble87] Blelloch, Guy: *"Scans as Primitive Parallel Operations"*, pp 355-362, Proceedings of the 15th International Conference on Parallel Processing, Pennsylvania State University Press, University Park, 1987

[Fal62] Falkoff, A. D.: *Algorithms for Parallel-Search Memories* JACM 9 #10, Oct 1962, pp 488-511.

[Fos76] Foster, Caxton C.: **Content Addressable Parallel Processors**,. Van Nostrand Reinhold, New York, 1976

[Fou87] Fountain, Terry: **Processor Arrays: Architecture and Applications**, Academic Press, London, 1987

[Hal81] Hall, J. S.: *A general-Purpose CAM-based System*, in **VLSI Systems and Computations**, Kung, Sproull, and Steele, ed. pp 379-388, Computer Science Press, Rockville, MD, 1981

[Hal89] Hall, J.S., S. Levy: *von Neumannizing the Multi-Search Content Addressable Memory* in Proceedings of the Symposium on Massively Parallel Processing, pp 27-42, University of South Carolina, Columbia SC 1989

[Hil85] Hillis, W. Daniel: **The Connection Machine**, MIT Press, Cambridge, 1985

[Sch87] Ilgen, Sener and Isaac D. Scherson: *"Parallel Processing on VLSI Associative Memory"*, pp 50-53, Proceedings of the 15th International Conference on Parallel Processing, Pennsylvania State University Press, University Park, 1987

[Koo70] Koo, J. T.: *"Integrated Circuit CAM"*, pp 208-215, IEEE J. Solid State Circuits, SC5, 1970

[Koh80] Kohonen, Teuvo: **Content-Addressable Memories**, Springer-Verlag, Berlin, 1980

[Lan76] Lange, R G: *"High Level Language for Associative and Parallel Computation with Staran"*, Proceedings of the 1976 International Conference on Parallel Processing

[Pot85] Potter, J. L. ed: **The Massively Parallel Processor**, MIT Press, Cambridge, 1985

[Pot88] Potter, Jerry L.: *Data Structures for Associative Supercomputers*, pp 77-84, Proceedings of the Frontiers of Massively Parallel Computation, 1988, IEEE Computer Society Press, order number 892

[Sch88] Scherson, Isaac and Smil Ruhman: "Multi-operand Arithmetic in a Partitioned Associative Architecture", Journal of Parallel and Distributed Computing 5, (1988) pp 655-668.

[Sto86] Stolfo, Salvatore J. and Daniel P. Miranker: *"DADO: A Tree-Structured Architecture for Artificial Intelligence Computation"*, pp 1-18, Annual Review of Computer Science, Annual Reviews, Palo Alto, 1986

March 92

DESIGN OF A LEAF CELL FOR THE RUTGER'S CAM ARCHITECTURE*

Donald E. Smith, Keith M. Miyake, and J. Storrs Hall

LCSR-TR-180

Laboratory for Computer Science Research
Hill Center for the Mathematical Science
Busch Campus, Rutgers University
New Brunswick, New Jersey 08903

*This work was supported by the Defense Advanced Research Projects Agency and the National Aeronautics and Space Administration under NASA-Ames Research Center grant NAG 2-668.

1 Hardware Design

Our Mar91-Aug91 progress report described the Rutger's CAM architecture as a collection tree sitting over a set of LEAF cells each with its own memory. Figure 1 shows this architecture and identifies the two cell types used in its implementation: LEAF cells that are composed of a processor and associated memory, and TREE cells that constitute the Collection Tree. These two cell types serve complementary functions within the architecture: TREE cells provide global processing for data collection, data movement, and parallel prefix(scan) operations over the LEAF cells while LEAF cells implement CAM-like operations and support local SIMD processing.

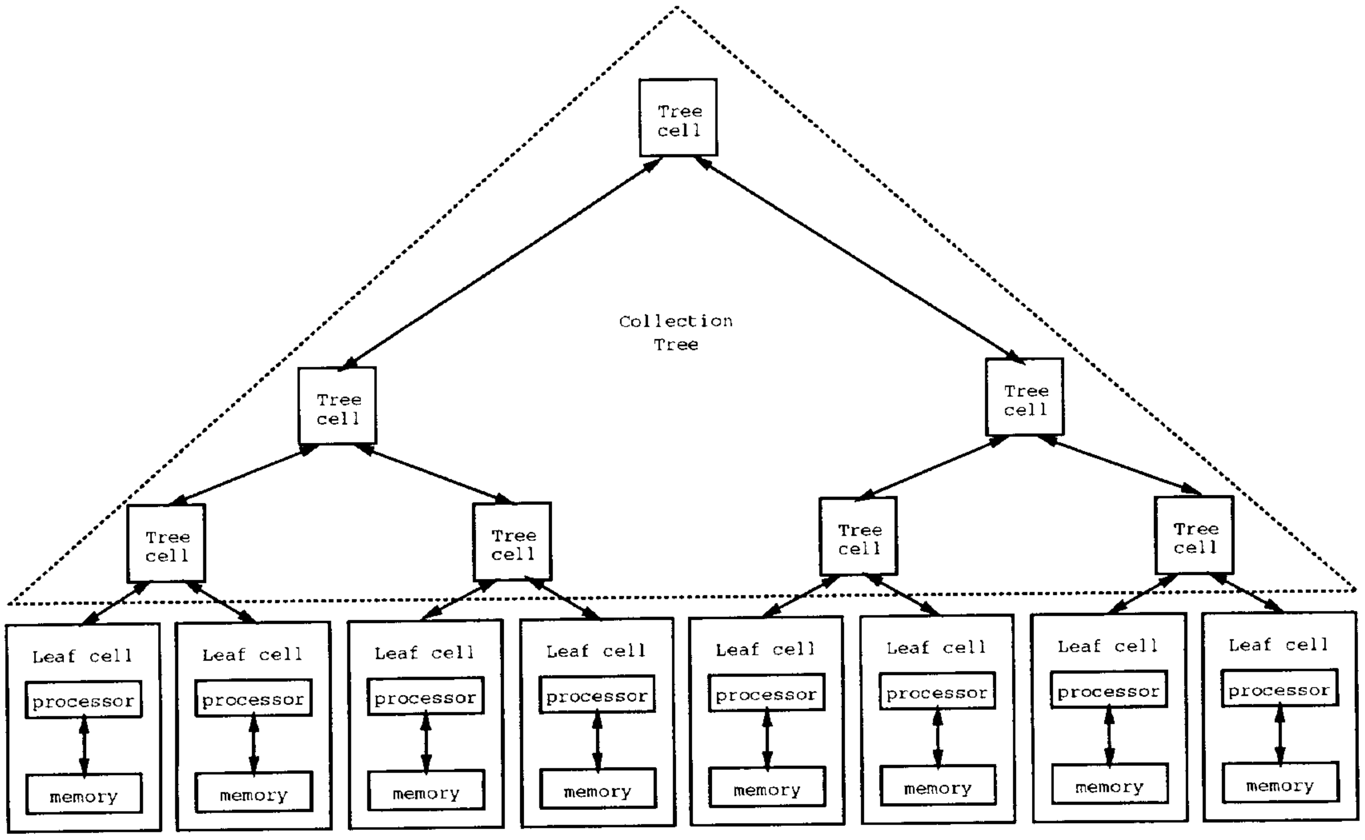


Figure 1: Collection Tree sitting over a set of LEAF Cells

A Register Transfer Level (RTL) description of the tree cell has been completed and tested. It supports both right-to-left and left-to-right integer scan operations, activity controlled and segmented operations, extended integer precision, as well as all tree operations described in our Mar91-Aug91 progress report.

The LEAF cell specification is nearly complete and the first draft of an RTL design is nearing completion. In the next six months, we expect to complete the LEAF cell design, interface it with our TREE cell design, and test these using the shortest path algorithm described in our Sep90-Feb91 progress report.

1.1 LEAF Cell Specification

Each LEAF cell is responsible for performing CAM-like operations and is composed of a main processor connected to three support processors via multi-ported interface registers. These support processors are the tree processor, memory processor, and IO processor (not shown in Figure 1). The LEAF cell's main processor is composed of two communicating components; a k-bit processor[1] for *standard* operations and a 1-bit processor used to control activity within a LEAF cell. Figure 2 shows the interconnection of these components.

1.2 Component Descriptions

The k-bit processor is a three bus (i.e., GB_0, GB_1, GB_2) architecture composed of:

- a k-bit ALU
- five general purpose k-bit registers GR_1, GR_2, GR_3, GR_4, and GR_5
- a dual ported k-bit flag register(FR) directly coupled to the fourteen 1-bit registers
- a dual ported (k+1)-bit tree register(TR) that provides the interface between leaf and tree processor
- a tri-ported k-bit refresh register register(RR) that provides the interface between the leaf, memory, and IO processors.
- a dual ported IO register(IOR) that provides the interface between the IO processor and the refresh register (RR).

The 1-bit processor is a four bus (i.e., BB_0, BB_1, BB_2, AC) architecture composed of:

- a 1-bit ALU
- five general purpose 1-bit registers BR_1, BR_2, BR_3, BR_4, and BR_5
- a dual ported 1-bit segment register(SEG), read by the tree processor, that indicates if a leaf processor is (SEG=1) or is not (SEG=0) the first element in a new segment.
- a dual ported 1-bit status register($VLD\uparrow$), read by the tree processor, that indicates if the tree processor should use ($VLD\uparrow=1$) or ignore ($VLD\uparrow=0$) the data in the tree register.
- five 1-bit status registers(OVERFLOW, ALLZERO, CARRY, SIGN, LOB) that contain the status of the k-bit ALU

[1]We expect the k-bit processor and its associated registers to be 32-bits wide; however, our design is not restricted to 32-bit widths but parameterized as a function of word width.

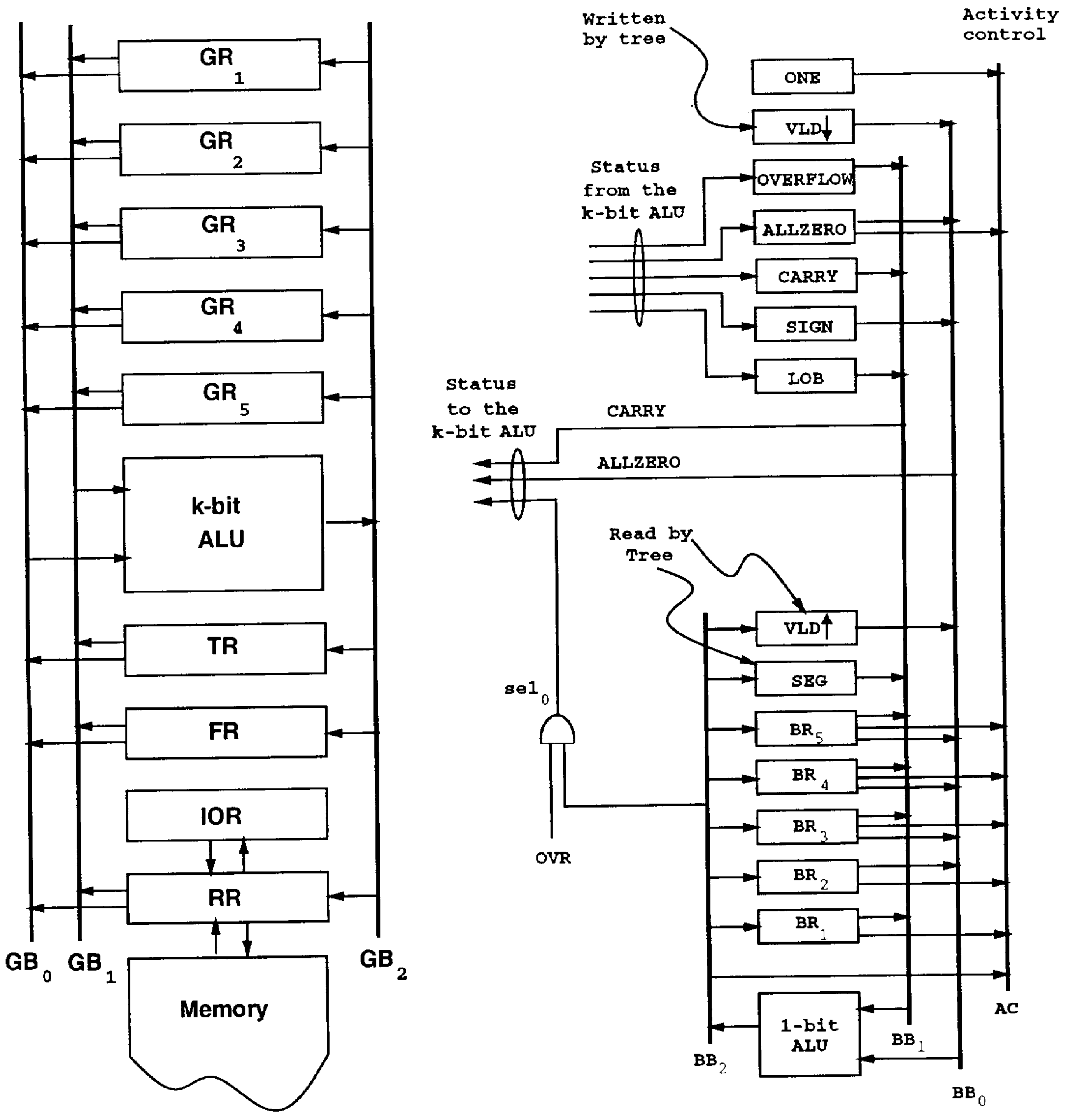


Figure 2: LEAF cell architecture

- a dual ported 1-bit status register(VLD$\downarrow$), written by the tree processor, that indicates if the leaf processor should use (VLD$\downarrow$=1) or ignore (VLD$\downarrow$=0) the data in the tree register
- a 1-bit constant (ONE) used to activate LEAF cells

1.3 Interfaces between the 1-bit and k-bit components

The 1-bit and k-bit components communicate through the processor status registers (i.e., the five 1-bit registers that maintain the status of the k-bit processor), through activity control, and through dedicated connections joining the fourteen 1-bit registers and the low-order 14 bits of k-bit register FR.

Activity control is determined by the value carried on AC in the 1-bit processor and is used to enable or disable the writing of both 1-bit and k-bit registers from their respective output buses (i.e., GB_2 and BB_2). The value placed on AC can be obtained from the output of the the 1-bit ALU (BB_2) or from one of the 1-bit registers BR_1, BR_2, BR_3, BR_4, BR_5, ALLZERO, or ONE.

The dedicated connections between the 1-bit registers and the low-order 14 bits of FR provides an additional interface that increases the bandwidth between the 1-bit and k-bit components. These connections are used by the following operations.

- copy all 1-bit registers into FR_0 through FR_{13}
- copy FR[0:4] into BR_1 through BR_5
- copy FR_5 and FR_6 into SEG and VLD$\uparrow$, *respectively*

1.4 Instruction Fields

Operation of the LEAF cells is specified by the fields summarized in Figure 3. The estimated width of each field is shown in parentheses. These specifications are tentative; refinements to field size and instructions will be made as algorithms are implemented on this architecture.

GOP
GOP encodes the operation to be performed by the k-bit ALU. The exact set of operations has not been determined but includes AND, OR, XOR, addition, and subtraction as well as sel-arg_1, sel-arg_2. These last two instructions select the indicated input argument and route it to the output.

R_{GB_0} and R_{GB_1}
R_{GB_0} and R_{GB_1} encode the refresh register, one of the k-bit general registers, the tree register, or the flag register. Our initial design using 5 general registers is encoded as follows:

GOP(4): encoded operation to be performed by the k-bit ALU

R_{GB_0}(3): encoded designation of the source register driving bus GB_0

R_{GB_1}(3): encoded designation of the source register driving bus GB_1

R_{GB_2}(3): encoded designation of the destination register reading bus GB_2

RRC(2): refresh register control

ADDR(lg(n)): memory address

WR(1): memory processor write control

IOC(1): IO register control

OVR(1): Enable/disable control of k-bit ALU's override capability

BOP(4): encoded operation to be performed by the 1-bit ALU

R_{BB_0}(3): encoded designation of the source register driving bus BB_0

R_{BB_1}(3): encoded designation of the source register driving bus BB_1

R_{BB_2}(3): encoded designation of the destination register reading bus BB_2

R_{AC}(3): encoded designation of the source driving the activity control line AC

Figure 3: Instruction Fields

000:RR   001:GR_1 010:GR_2 011:GR_3 100:GR_4 101:GR_5 110:TR   111:FR

R_{GB_2}

R_{GB_2} encodes a null destination(λ), one of the k-bit general registers, the tree register, or the flag register. The encoded designation determines the register that will record the value on GB_2. The value is recorded if and only if the LEAF processor's activity control, as determined by AC, is set. This field cannot specify the refresh register. Our 5 register design is encoded as:

000:λ 001:GR_1 010:GR_2 011:GR_3 100:GR_4 101:GR_5 110:TR   111: FR

RRC

RRC is a 2 bit field that specifies what data, if any, is written to the refresh register. The field encodes one of four possibilities: a noop(λ), write from GB_2, write from memory, or write from IOR. Specifying this field independent of the R_{GB_2} field allows the refresh register to be written in parallel with any of the destinations specified by R_{GB_2}. It also isolates the leaf processor from the memory and IO processors allowing each to run at its own optimal speed. Activity control is used when RR is being written from GB_2 - it is ignored for all other cases. Our current design encodes RRC as follows:

00: λ 01: RR $\leftarrow GB_2$ 10: RR $\leftarrow$ M[ADDR]   11: RR $\leftarrow$ IOR

ADDR

ADDR is a $\lg(n)$ bit field, where n is the number of words of memory, that specifies the memory location to be read from or written to. This field is required only when a read or write is being performed.

WR

WR is a 1-bit field that indicates when data is to be written from the refresh register to the memory (M[ADDR] $\leftarrow$ RR). This operation can be performed in parallel with other operations that access the refresh register.

IOC

IOC is a 1-bit field that specifies when data is written from the refresh register to the IO register (IOR $\leftarrow$ RR).

OVR

OVR is a 1-bit field that enable or disables the override capability of the k-bit ALU. When enabled the k-bit ALU will perform either the operation specified by field GOP or override that specification and transfer data from input bus GB_0 to output bus GB_2. The override capability is used for computing inclusive scans from exclusive scans as well as providing MUX-like capabilities to the k-bit ALU (see section 1.6).

BOP
BOP encodes the operation to be performed by the 1-bit ALU. The bits are the entries in the truth table of the boolean function being computed.

R_{BB_0}, R_{BB_1}, and R_{AC}
R_{BB_0}, R_{BB_1}, and R_{AC} each encode 8 registers (not the same registers). These encodings are based on the interrelations between the 1-bit registers and a *typical* instruction mix. The specific registers connected to each bus and their encodings will be altered as experience is gained with algorithms on this architecture. The following are design goals which influence these choices as well as an initial assignment of registers to busses.

- Extended precision requires that the CARRY and ALLZERO status outputs from the k-bit processor be fed back into the processor's CARRY and ALLZERO inputs. Consequently, data paths that allow parallel routing of the CARRY and ALLZERO registers to the k-bit processor must be supported.
- Inclusive scans are completed in the LEAF processors using the results of the exclusive scan produced by the tree and two additional steps[2] performed by the LEAF processor. These steps makes use of the 1-bit ALU as well as the override feature of the k-bit ALU and require that 1-bit registers VLD↓ and SEG be presented in parallel to the 1-bit ALU. Details of these operations are described in section 1.6.
- R_{BB_0} encodes one of BR_2, BR_3, BR_4, BR_5, VLD↑, SIGN, ALLZERO, and VLD↓
- field R_{BB_1} encodes one of BR_1, BR_3, BR_4, BR_5, SEG, LOB, CARRY, and OVERFLOW
- The field R_{AC} encodes one of BB_2(i.e., the output bus of the 1-bit ALU) BR_1, BR_2, BR_3, BR_4, BR_5, ALLZERO, and ONE

R_{BB_2}
R_{BB2} encodes one of null(λ), BR_1, BR_2, BR_3, BR_4, BR_5, SEG, VLD↑. This field determines the register that will record the value on BB_2. This value is recorded if and only if the processor's activity control, as determined by AC, is set. Our initial design is encoded as:

000:λ 001:BR_1 010:BR_2 011:BR_3 100:BR_4 101:BR_5 110:SEG 111:VLD↑

1.5 Interaction between the LEAF and memory processors

The read and write commands form the conceptual interface between the LEAF processor and the memory processor. They are executed by the memory processor and cause data to be transferred between RR and the memory. These commands are encoded in two fields, RRC and WR. The read command is encoded in the RRC field as a command to move data from memory to the refresh register. This command causes the memory processor to

[2]Plus scans only require one additional step.

transfer the data at the location specified by ADDR to the refresh register. If the read is destructive, as is the case with a DRAM implementation, the main control unit will issue a write command to rewrite the contents of RR back to memory.

The write command specified by WR indicates that data is to be written from RR to memory and must not conflict with RRC - the WR bit CANNOT specify write-to-memory when RRC specifies read-from-memory. These commands are not affected by the processor's activity control.

Once initiated by either a read or write the memory processor performs the data transfer asynchronously. The LEAF processor should not change RR while the memory processor is busy and is thus limited to using RR only when the memory processor is idle.

1.6 Interaction between TREE and LEAF processors

Most of the interactions between the TREE and LEAF processor are of the *standard* variety; however, there are two types of interactions that require special attention. One is the handling of overflow between these two processors. The other is the communication of scan results between the two processors.

These interactions are handled through the interface provided by TR, VLD$\uparrow$, SEG, and VLD$\downarrow$. TR acts as a bidirectional data port connecting the two processors. When the tree accepts data from the leaves it reads VLD$\uparrow$ and SEG. VLD$\uparrow$ indicates if the data in TR should, or should not, participate in the tree operation while SEG indicates if the leaf is, or is not, the first in a segment. These two 1-bit registers can be read and written by the 1-bit ALU; however, they may only be read by the tree.

When the tree provides data to the leaves it use VLD$\downarrow$ to indicates if the data in TR should, or should not, be used by the leaf. VLD$\downarrow$ can only be read by the LEAF processor and can only be written by the TREE processor. There is dedicated hardware in the tree that computes VLD$\downarrow$ as a function of the SEG, VLD$\uparrow$, and the direction of the scan (left-to-right or right-to-left). Changing any of these three fields will cause VLD$\downarrow$ to change.

1.6.1 Overflow Interactions

Since the TREE and LEAF processors jointly participate in numerical computations, the LEAF processor must be responsive to overflow information generated by the TREE processor. This information is stored in a single bit in TR and must be used by the LEAF processor's k-bit ALU to determine its overflow condition. The overflow condition of a LEAF cell operation (even data movement such as $GR_i \leftarrow TR$) that involves the tree register as a source must be dependent on the overflow condition generated in the tree. If TR indicates an overflow from the tree, the LEAF processor must complete the specified operation and set the overflow status to true. The overflow status must also be set if the LEAF processor performs an arithmetic operation that itself generates an overflow.

In the case when the tree is not supplying data to the leaf (i.e., VLD$\downarrow$ is 0), the overflow field in TR is ignored.

1.6.2 Inclusive and Exclusive Scans

When an inclusive scan is desired, the LEAF processor must compute it from the TREE cells exclusive scan and its own internal data.

These operation are performed using the override capability of the k-bit ALU which permits the ALU to either execute the operation specified or to ignore the operation and replace it with a data transfer. The data transfer routes the data on input bus GB_0 directly to output bus GB_2. When an override is in effect the operation specified by GOP and the data on input bus GB_1 are ignored by the k-bit ALU.

Figure 4 shows a operation and its overridden counterpart. Notice that the data transfer that replaces the specified operations ignores the fields R_{GB_1} and GOP but uses the fields R_{GB_0} and R_{GB_2} without alteration.

	Operation Mnemonic	Operation field specification
Operation	$GR_i \leftarrow (GR_j \text{ op } GR_k)$	R_{GB_0}=j, R_{GB_1}=GR_k, R_{GB_2}=i, GOP=op
Overridden Counterpart	$GR_i \leftarrow GR_j$	R_{GB_0}=j, R_{GB_2}=i

Figure 4: Comparison of normal operation and it overridden counterpart

In order to produce inclusive scans from exclusive scans two cases must be considered. One, when the tree processor provides data on which the inclusive scan depends ($VLD\downarrow=1 \wedge SEG=0$) and two, when it does not ($VLD\downarrow=0 \vee SEG=1$). Notice that these conditions are functions not only of the results produced in the tree but also of the segmentation bit in the leaf processor. This latter dependence is due to the fact that the first leaf processor in a segment (SEG=1) must ignore the data it receives from the tree since this data is from a different segment. The LEAF processor must be able to distinguish between these cases and complete the inclusive scan appropriately. Figure 5 shows the operations required of the the LEAF processor for these two cases.

Tree data should be used	$GR_i \leftarrow (GR_j \text{ op } TR)$
Tree data should be ignored	$GR_i \leftarrow GR_j$

Figure 5: Leaf operations for forming inclusive scans from exclusive scans

Since the second of these operations is an overridden version of the first, the choice between the two operations can be performed by the override capability of the k-bit ALU by using the single instruction show in Figure 6.

OVR	k-bit operation	1-bit operation
on	$GR_i \leftarrow (GR_j \text{ op } TR)$	$\lambda \leftarrow (VAL\downarrow \wedge \overline{SEG})$

Figure 6: Using OVR to form inclusive scans from exclusive scans

1.6.3 Using the k-bit ALU as a Multiplexor for MIN and MAX scans

The OVR field can also be used to cause the k-bit ALU to function as a MUX routing one of its inputs, GB_0 or GB_1, to GB_2. This is accomplished by using OVR and the 1-bit ALU in conjunction with the k-bit operation sel-arg_2 as shown below. Notice that the sel-arg_2 routes the second argument to the output and that its overridden counterpart routes the first argument.

$$GR_i \leftarrow \text{sel-arg}_2(GR_j, GR_k)$$
$$\text{where: sel-arg}_2 \text{ performs } GR_i \leftarrow GR_k$$

The ability to use the k-bit ALU as a MUX also provides support for forming inclusive scans from their exclusive counterparts. It is especially useful for MIN and MAX scans because these operations are performed differently in the TREE and LEAF cells[3]. In a TREE cell MIN and MAX are functions that output either the MIN or MAX of their inputs but in a LEAF cell, MIN and MAX are *simulated* by comparing two data and selecting one based on the result of the comparison. This difference requires that inclusive MIN and MAX scans use the two steps show in Figure 7 to convert an exclusive scan to an inclusive scan.

$$\underline{GR_i \leftarrow MIN(GR_j, TR)}$$

OVR	k-bit operation	1-bit operation
off	$\lambda \leftarrow (GR_j - TR)$	$BR_1 \leftarrow (\overline{VLD\downarrow} \vee SEG)$
on	$GR_i \leftarrow (\text{sel-arg}_2(GR_j, TR)$	$\lambda \leftarrow (BR_1 \vee SIGN)$

Figure 7: LEAF cell steps to complete MIN/MAX inclusive scans

While the comparison (GR_j - TR) is taking place in the k-bit processor, the 1-bit processor decides if TR should be used to complete the scan - this result is stored in BR_1. In the second step the k-bit processor is used as a multiplexor selecting either GR_j or TR and routing it to GR_i. Since OVR is on, the specific selection is determined by the 1-bit ALU computation. GR_j is selected when either the tree data is not to be used or the contents of GR_j is less than TR.

[3]Addition is performed identically in the LEAF and TREE cells.

1.6.4 Inserting identity elements

TREE cells perform exclusive scans (i.e. the initial element in a segment is the identity element for the operation) but do not insert the identity element into the result. In order for the result to contain the identity element the LEAF processor must insert it.

This is accomplished using the MUX-like capabilities of the LEAF processor to choose between the data provided by the tree and the identity element for the operation. Figure 8 shows the constants that are expected to be of special interest. These five constants can be constructed from a 3-bit field in which one bit specifies the high order bit, one the low order bit, and one the internal bits.

Constant			Use
0	00..00	0	identity for +, OR, MAX on positive integers
1	11..11	1	identity for AND, MIN on positive integers ; decrement
1	00..00	0	identity for MAX on 2's complement
0	11..11	1	identity for MIN on 2's complement
0	00..00	1	increment

Figure 8: Important Constants

1.7 Activity control

Activity control is used on all k-bit and 1-bit registers. These registers latch their input values at the end of each execute cycle if and only if they are selected by R_{GB_2} or R_{BB_2} and AC is set. There is no activity controlled write-to-memory; however, the effect of this operation can be obtained by using the standard memory operations in conjunction with an activity controlled operation on RR. Figure 9 shows how this is accomplished.

$\underline{M[ADDR] \leftarrow GR_i \text{ in all active leaf nodes}}$

$RR \leftarrow M[ADDR]$
$RR \leftarrow GR_i$ mediated by AC
$M[ADDR] \leftarrow RR$

Figure 9: Activity Controlled write to memory

March 92

CIRCUIT DESIGN TOOL USER'S MANUAL*

Keith M. Miyake and Donald E. Smith

LCSR-TR-181

Laboratory for Computer Science Research
Hill Center for the Mathematical Science
Busch Campus, Rutgers University
New Brunswick, New Jersey 08903

*This work was supported by the Defense Advanced Research Projects Agency and the National Aeronautics and Space Administration under NASA-Ames Research Center grant NAG 2-668.

Contents

Chapter 1

Introduction

The design of the CAM chip has been done in a UNIX software environment using a design tool that supports the definition of digital electronic modules, the composition of these modules into higher level circuits, and event-driven simulation of these circuits. Our tool provides an interface whose goals include straightforward but flexible primitive module definition and circuit composition, efficient simulation, and a debugging environment that facilitates design verification and alteration.

The tool provides a set of primitive modules which can be composed into higher level circuits. Each module is a C-language subroutine that uses a set of interface protocols understood by the design tool. Primitives can be altered simply by *recoding* their C-code image; in addition new primitives can be added allowing higher level circuits to be described in C-code rather than as a composition of primitive modules - this feature can greatly enhance the speed of simulation[1].

Effective composition of primitive modules into higher level circuits is essential to our design task. Not only are the *standard* features of a description language required but in addition, features such as recursive descriptions of circuit composition, parameterized module descriptions, and strongly-typed port types are essential to efficient circuit design. These features are supported by our design tool's composition language which allows the user to specify a hardware description in a C-like syntax. Parameterized modules, recursive and iterative descriptions, macro-like capability to describe collections of wires (i.e., cables), and decision making support that allows context sensitive module expansion are provided by our tool. In addition, our tool can determine the cost of a circuit based on the costs of its primitive modules. This feature is not *exact* but does provide a good approximation of the complexity of the designed circuit.

Simulation is performed by an event-driven simulator that handles gates as well as tri-state bi-directional busses and provides the user not only with a view of what a circuit is computing but also control over the circuit so that *design flaws* can be effectively isolated

[1]Converting a higher-level circuit into a primitive module is straightforward when the timing of the primitive module need not be identical to the higher level circuit. Higher level circuits can be converted to primitive modules with identical time performance; however, the conversion process is much more complex.

and corrected. The simulator is controlled with a command language which allows the user to *see* a wire or set of wires, as well as change the values on wires. Operations can be done *immediately* (i.e., at the time the user enters them) or scheduled to take place at a specified time. Simulations can be run for a specific period of time or until a certain condition is detected in the hardware. They can be controlled from the keyboard or indirectly from a file.

The design tool consists of two main parts: the command and definition languages. The *definition language* is used to read circuit definitions. The *command language* controls the actions of the simulator. These topics are detailed in sections 2.5 and 2.6.

Following is a brief introduction on the definition and creation of circuit models. It defines many terms used later.

1.1 Circuit Definition

The circuit definition language describes connections between primitive objects. These objects, called *primitive modules*, have functionality predefined in the design tool. Primitive modules have a special set of entry points which are connected when forming the circuit model. These entry points are called *ports* and the connections between them are referred to as *signals* or *wires.* There is a causality between connected modules. Execution of a module may affect modules connected to it.

The design tool reads descriptions using a definition language. The language consists of two types of object definitions: module and cable. *Cable definitions* group related signals together. *Module definitions* specify primitive modules and their connections. A module may define other modules as children of itself, and specify connections between its child modules. In this case the module is referred to as a *composite module.*

The circuit is built from a set of hierarchical module and cable definitions. Flattening the hierarchy produces the basic model of a set of primitive modules connected by wires.

In order to name objects in the hierarchical design, *hierarchical names* are used by the design tool. These names specify objects which cannot be directly referenced within the current context. This is done by supplying a list of names specifying a path to the object. Each field in the composite name is separated by the dot character '.'.

1.2 Model Creation

The creation of a circuit model is performed in phases.

When a cable or module definition is read, its syntax is checked and the definition is stored as a *master definition.* These definitions may have input arguments which need assignment.

When a module is created, input arguments to master definitions are assigned resulting in a new definition type. These definitions, which have specific input arguments, are referred to as *definition instances*. Each definition instance is fully examined, checking the consistency of the connections made and the referenced modules.

The circuit model is made from these definition instances. The model is designed for speed in simulating the functionality of the circuit and contains all structures necessary for simulation. Such a model is called a *generated module* or *simulation instance*.

Chapter 2

The Definition Language

The definition language is used to describe digital electronic circuits by building hierarchical structures connecting primitive modules. A definition file consists of a sequence of module and cable definitions. Modules come in two types: primitive and composite.

Primitive modules are the basic building blocks of the definition language. These objects perform operations defined by C functions which have been precompiled into the design tool. A list of primitive modules is given in appendix B.

A composite module definition defines submodules of itself and connections to be made among their ports, as well as its own ports. Attaching submodule ports causes interactions between the operations of the respective modules.

Cable definitions allow signals to be identified in groups, which simplifies connection of ports.

Module and cable definitions are similar in structure and are analogous to functions in a conventional programming language. They may have formal input arguments and may use other definitions (as well as their own) recursively. Termination of such recursion is not assured.

2.1 Syntax Conventions

The language syntax descriptions used in this manual is a variant of the Backus-Naur form.

Following is a list of syntax rules:

1. **Boldface** type denotes reserved words.
2. Lowercase words, which may have embedded underscores, denote syntactic constructs.
3. Character tokens are shown using `typewriter` type. Most punctuation characters are used as character tokens, with exceptions stated below. Note that the exceptions are printed in Roman type.

4. The vertical bar '|' separates alternate syntax items when it is used at the beginning of a line.

5. Square brackets ('[', ']') enclose optional items.

6. The dollar sign '$' in a syntax rule denotes the remainder of the line as a comment. '$' is not used in the syntax.

2.2 Variables and Assignment

A variable is a name associated with an integer value in a module or cable definition. Variables have no meaning outside the current definition. A variable name may be any valid string token (quoted or unquoted; see appendix A). There are no arrays of variables. A variable may have the same name as signals, modules, or cables since its context is distinct. There are three variable types: input, loop, and assignment. Within a specific definition, a variable may be used as only one type.

Input Variables

Input variables are arguments to a module or cable definition. They are determined at invocation and may not be reassigned within the current object. These variables are valid throughout the current object. Each input variable of a definition must be given a value upon use.

Loop Variables

Loop variables are used in **for** loops in the component section of modules. Each **for** loop controls the assignment of a single loop variable. Loop variables are only valid within the controlling loop, and may not be reassigned within the loop.

Assignment Variables

Assignment variables are used in the component section of modules. They are set using the **assign** statement (string_token <- arith_expr ;). This assigns the current value of the expression to the variable. Once a variable has been assigned to, it is valid until the end of the module. Each subsequent use of the variable gets the assignment value unless the variable has been reassigned. Assignment variables may not be reused as loop variables.

Control flow variations resulting from **if** statements or loops may allow an assignment variable to be referenced prior to assignment.

Cables only have input variables since they have no component section.

2.3 Expressions

Expressions are used in various ways to control the assembly of modules. There are two types of expressions: arithmetic and logical. *Arithmetic expressions* result in integer values. *Logical expressions* return one of the values TRUE or FALSE. Arithmetic and logical expressions are not interchangeable.

Arithmetic Expressions

Arithmetic expressions compute integer values. They may be string tokens (variables), numeric tokens (constants), or may be created by application of an arithmetic operator to one or more arithmetic expressions.

```
arith_expr :=
        string_token                  $ variable value
        | numeric_token               $ constant value
        | - arith_expr                $ arithmetic negation
        | ( arith_expr )              $ arithmetic grouping
        | arith_expr * arith_expr     $ multiplication
        | arith_expr / arith_expr     $ division
        | arith_expr % arith_expr     $ modulus
        | arith_expr + arith_expr     $ addition
        | arith_expr - arith_expr     $ subtraction
```

Division operations return a truncated result (using C convention), since integer division is not exact.

There are three levels of arithmetic operator precedence. Unary operators (negation and grouping) share the highest precedence. Multiplication, division, and modulus (*, /, %) have equal precedence, below that of the unary operators. Addition and subtraction (+, -) share the lowest precedence.

All binary arithmetic operators associate left-to-right.

Logical Expressions

Logical expressions compute the value TRUE or FALSE. They are constructed by the use of relational or logical operators. *Relational operators* produce a logical expression based on the validity of a relational query between two arithmetic expressions. *Logical operators* use one or two logical expressions to produce a single logical expression. There are no logical variables or constants.

```
log_expr :=
        arith_expr > arith_expr        $ greater than
        | arith_expr >= arith_expr     $ greater than or equal to
        | arith_expr < arith_expr      $ less than
        | arith_expr <= arith_expr     $ less than or equal to
        | arith_expr = arith_expr      $ equal to
        | arith_expr ! arith_expr      $ not equal to
        | ~ log_expr                   $ logical negation
        | { log_expr }                 $ logical grouping
        | log_expr & log_expr          $ logical AND
        | log_expr | log_expr          $ logical OR
```

Note that the vertical bar in the logical OR represents the character '|'.

The use of arithmetic expressions as operands eliminates precedence or associativity with regard to relational operators.

There are three levels of logical operator precedence. Unary logical operators (negation and grouping) have the highest precedence, followed by logical AND. Logical OR has the lowest precedence of logical operators.

Logical grouping syntax is distinct from that of arithmetic grouping. This reinforces the idea of noncompatibility between expression types.

2.4 Naming Conventions

Each child object (signal, cable, or submodule) in a definition must be given a unique name. This allows unambiguous signal naming within simulation instances (for design verification). Names of child objects must be string tokens (quoted or unquoted).

An object name may have a single associated array index. This index is specified by an arithmetic expression enclosed in square brackets following the name. The string token, excluding the array index, is called the *root name* of the object.

```
object_name :=
        string_token
        | string_token [ arith_expr ]
```

Note that the square brackets do not represent optional arguments.

Example:
    The root name of an object "a[5]" is simply "a".

It is often useful to name lists of objects. In this case, a modified array notation, called an object list, may be used to specify a range of array indices. This is done by supplying a start and end index for the array, separated by a colon ':'. The notation is equivalent to supplying each object name in order, beginning with the start index, and iterating until the end index is reached. If the start index is less than the end index iteration increments by one, otherwise it decrements by one.

```
object_list :=
        string_token [ arith_expr : arith_expr ]
```

Note that the square brackets do not represent optional arguments.

Example:
"a[1:2]" expands to "a[1]" "a[2]".
"a[2:1]" expands to "a[2]" "a[1]".

A list of objects may contain single object names and object lists.

```
object_name_list :=
        object_name
        | object_list
        | object_name object_name_list
        | object_list object_name_list
```

In a module definition, each internal subcomponent or signal must have a distinct name (root name and index). Also, objects with the same root name must have similar types. This means that signals, components and cables may not share root names. Furthermore, components or cables which share a root name must share the same master definition. These checks are performed during the creation of module definition instances.

It is often necessary to name an object which cannot be directly referenced from the current level. In this case a composite name is used, using the dot character '.' to separate levels. This is referred to as a hierarchical name.

```
hierarchical_name :=
        object_name
        | object_name . hierarchical_name
```

Example:
The hierarchical name "a.b" refers to an object "b" which is a child of object "a", where "a" is a child of the current module.

Hierarchical naming may be used with array expansion, in which case rightmost indices are expanded first.

```
hierarchical_list :=
        object_name
        | object_list
        | object_name . hierarchical_list
        | object_list . hierarchical_list
```

Example:
"a[1:2].b[3:4]" expands to "a[1].b[3]" "a[1].b[4]" "a[2].b[3]" "a[2].b[4]".

The hierarchical analog to an object name list, called a hierarchical list, may now be defined. Note that every hierarchical name is also a hierarchical list.

```
hierarchical_name_list :=
        hierarchical_list
        | hierarchical_list hierarchical_name_list
```

2.5 Cable Definitions

A cable represents an ordered list of signals, each signal having an associated type. Signal typing is used to ensure that the use of a module is consistent with its definition.

```
cable_definition :=
        cable string_token [ ( variable_list ) ] typed_signal_list end

variable_list :=
        string_token
        | string_token , variable_list

typed_signal_list :=
        signal_name_list signal_type
        | signal_name_list signal_type typed_signal_list

signal_name_list :=
        object_name_list
        | cable_use
        | object_name_list signal_name_list
        | cable_use signal_name_list

signal_type :=
        input
        | output
        | inout
```

The string token following **cable** is the cable name. This name is used for future references to the cable. The variable list is a list of input variables for the cable. When the cable is used, each input variable must be given a value. The typed signal list is a list of the wires which comprise the cable. It may include cables uses, which is defined below. Each signal is given one of three allowable types: **input**, **output**, or **inout**. The meaning of the types will be described in section 2.6.

Example:

```
cable c1
   s1 s2 input
   s3 output
   s4 inout
end
```

In the example, signals “s1” and “s2” are both **input**.

After a cable has been defined, it may be used anywhere that a signal may be used. This includes being used in other cable definitions. The following syntax defines a name as a use of a cable.

cable_use :=
  cable string_token [(argument_list)] object_name
  | **cable** string_token [(argument_list)] { object_name_list }

argument_list :=
  arith_expr
  | arith_expr , argument_list

The string token following **cable** is the name of a cable definition. The argument list given must be exactly the same size as the number of input variables to the cable definition. Following the arguments is the list of new cable instance names.

Example:

```
cable c1 ci1
cable c1 { ci2 ci3 }
```

The first use defines a single instance “ci1” of cable “c1”. The second use defines two additional instances, “ci2” and “ci3”, of cable “c1”.

When a cable is used in another cable definition, the type of the resultant signal depends on both the signal type given in the previous definition, and the type given to the cable use.

The following matrix shows the retyping rules:

cable type	*subsignal type*		
	input	**output**	**inout**
input	input	output	inout
output	output	input	inout
inout	inout	inout	inout

After a cable instance has been defined, each use of the instance name represents the list of its component signals in order. Each signal name in the list is a hierarchical name consisting of the cable instance name and the component signal name. Individual signals within the cable may be accessed by naming the signals hierarchically.

Example:

```
cable c2
   s1 input
   s2 output
end
```

```
cable c3
   cable c2 sc1 input
   cable c2 sc2 output
   cable c2 sc3 inout
end
```

In cable "c3", signal "sc1.s1" would be **input** and "sc1.s2" would be **output**. Because of retyping, signal "sc2.s1" would be **output** while "sc2.s2" would be **input**. Both subsignals of "sc3" would be **inout**.

Example:
If we make a instance "ci4" of the cable type "c1", individual signals may be referenced as "ci4.s1", "ci4.s2", "ci4.s3", and "ci4.s4". This set of signals, in order, can be referenced simply as "ci4".

Cable definitions may use other cable definitions, including those which are not yet defined (forward referencing). There is no check for recursive cable references, which do not terminate.

2.6 Module Definitions

Two types of modules (primitive and composite) are used in circuit designs. Primitive modules are objects with predefined functions. Composite modules define connections between primitive and composite modules.

module_definition :=
  module string_token [(variable_list)] [cost_section]
    [port_section] [signal_section] [component_section] **end**

The string token following **module** is the module name. This name is used to reference the module in future use. As with cable definitions, when a module is used each input variable must be given a value.

Additional module examples are given in appendix C.

Cost section

The cost section is used to estimate the relative expense of building modules using several technologies. Each module definition instance has associated cost values. These costs may be explicitly defined in the cost section, or may be implicitly defined as the sums of the costs of its submodules. Primitive modules should define explicit costs with a cost section.

Composite modules should include a cost section if the hardware implementation of the module does not correspond to the functional model represented by its subcomponents.

cost_section :=
  costs cost_pair_list

cost_pair_list :=
  cost_pair
  | cost_pair cost_pair_list

cost_pair :=
  nmos : arith_expr
  | **cmos** : arith_expr
  | **gateInput** : arith_expr

Currently there are three cost criteria: **nmos**, **cmos**, and **gateInput**. If a technology cost is given more than once, the last cost pair is used.

Example:

```
module m1(v1)
   costs
      nmos:  2*v1
      cmos:  10
      gateInput:  20
end
```

Port section

The port section is an ordered list of the external connections of the current module. Ports are special signals which are used to connect to the module in later uses. A module with no ports cannot be referenced by another module. The order of port signals is important and determines proper connection of the module.

port_section :=
  ports typed_signal_list

The typed signal list is the same as used in cable definitions, with the same subsignal retyping rules.

Type information defines the proper use of the signal in the module and what connections are allowed if the module is referenced by a composite module.

input implies that the signal is generated from an external source.

output implies that the signal is generated within the current module.

inout does not state the source of the signal. It causes the signal to be a (bi-directional) bus, which must be driven by tri-state drivers.

The following rules govern valid connections to each type of port signal within the current module:

input: No output signal may be connected to the signal. At least one primitive descendent module must use the signal as an input.

output: At least one primitive descendent module must use the signal as an output.

inout: At least one primitive descendent module must use the signal as an input or output. Additionally, every connected output must be a a tri-state driver (the signal is a bus).

Missing or inconsistently typed signal connections are reported upon creation of module definition instances.

Example:
```
module m2
   ports
      p1 input
      p2 output
      p3 inout
end
```

Signal section

The signal section defines internal signals of the current module. These internal signals must be distinct from port signals and may not be referenced by other modules. Every signal used in a module definition must be defined in either the port or signal section. The order in which internal signals are defined is not important.

signal_section :=
  signal signal_name_list

Each signal defined in the signal section is given a special type of **internal**. If a cable use is defined in this section, all resulting signals are also typed as **internal**.

The **internal** type means that the signal is both generated and used by primitive descendents of the current module.

Example:
```
module m3
   signals
      s1 s2 s3
end
```

Component Section

The component section determines how composite module are built from other modules. This is accomplished by 'executing' component statements in order, similar to conventional programming languages. Primitive modules, whose functions are defined by C code, do not use their component sections.

component_section :=
    components component_statement_list

component_statement_list :=
    component_statement
    | component_stmt component_statement_list

component_statement :=

submodule_statement	$ declare a child module
| assign_statement	$ assign a value to a variable
| join_statement	$ create a link between a group of signals
| error_statement	$ print an error message
| grouping_statement	$ group multiple statements
| if_statement	$ execute statements conditionally
| for_statement	$ execute a statement loop iteratively
| while_statement	$ execute a statement loop conditionally
| break_statement	$ exit from loops

Submodule Statement

Submodule declares a module as a child of the current module. It also designates attachment of signals to the ports of the child module.

submodule_statement :=
    object_name string_token [(argument_list)] hierarchical_name_list ;

The initial object name is the local name given to the submodule. This name is used to refer to the child module within the current module. Specifially, it is used in hierarchical naming. The next string token is the name of a module definition. The number of arguments given must match the number of input variables of the module definition. Next is a list of signals to be attached to the ports of the child module. Because every signal must be declared in the port or signal section, references to cables use hierarchical names (and not cable uses). Each signal in this list will be connected to the corresponding port of the previously defined module in order. The signal list must be the same size as the number of ports of the previously defined module. The port and connecting signal must conform, using the rules stated under the port section.

Example:
```
module m4
   ports
```

```
        m4i input
        m4o output
    signals
        ic1
end

module m5
    ports
        m5i input
        m5o output
    components
        sc1 m4 m5i m5o;
end
```

In the example, module "m5" defines a child module of type "m4" and gives it the local name "sc1". The hierarchical name which refers to the signal "ic1" in "m4" is "m5.sc1.ic1". Note that the ports of "m4" and the connecting signals in "m5" correspond in type.

Assign Statement

Assign associates an integer value with a variable. The target of **assign** may be any unused variable name, or an assignment variable. Execution of **assign** causes the expression value to be computed and assigned to the variable.

assign_statement :=
    string_token <- arith_expr ;

The string token names the variable to be assigned.

Example:

```
module m6
    components
        v1 <- 2;
        v2 <- v1 * 2;
        v1 <- 1;
end
```

The first **assign** creates a new variable "v1" with a value of 2. The second creates "v2" and uses "v1" to compute "v2" as 2*2 = 4. The final **assign** changes "v1" to 1, but does not affect "v2".

Join Statement

Join merges a set of signals to form a single signal. After a **join** has been completed, any member can be used to represent the set in other component statements (including joins).

Every signal used in a **join** must be declared in the port or signal section of the module.

The merging of signals caused by a **join** may introduce non-obvious inconsistencies in the connection of modules. These inconsistencies are reported upon execution of the **join**.

join_statement :=
    join [hierarchical_name_list] ;

Note that the square brackets above do not indicate an optional argument.

Example:

```
module m7
   signals
      s1 s2 s3
   components
      join [s1 s2];
      join [s2 s3];
end
```

The first **join** merges signals "s1" and "s2". The second merges the signal "s3" with the signal which is the **join** of "s1" and "s2".

Error Statement

Error allows the user to print a message during the course of module generation. The message is a single string (no variables), and is designed mainly for identifying situations that should not occur.

error_statement :=
    error string_token ;

Execution of **error** causes activation of an error message with the error flag mask activated. The error mask value is given in appendix D. These messages may be suppressed or may cause program termination by options in the **simrc** file.

Grouping Statement

Grouping allows multiple component statements to act as a single statement lexically. This allows multiple statements to be used as targets in **if**, **for** and **while** statements. **Grouping** has no affect in other contexts.

grouping_statement :=
    { component_statement_list }

Note that there is no semicolon following the grouping statement.

Grouping does not affect the lexical scope of any variable.

If Statement

If allows conditional execution of a statement depending on the result of a logical expression. Multiple statements may be executed by the use of **grouping**.

if_statement :=
    if log_expr component_statement [**else** component_statement] ;

If executes the first component statement when the logical expression is TRUE. When the logical expression is FALSE, the second component statement (in the optional **else** clause) is executed if available.

For Statement

For executes a statement a specified number of times. Multiple statements may be executed by the use of **grouping**.

For evaluates two bounding expressions once to find the inclusive range for its loop control varable. The target statement is then repeatedly executed with the loop variable set to each value in the range. The loop variable is initially set to the value of the first expression. If the first expression is less than the second expression, the loop variable is incremented by one after each iteration; otherwise the variable is decremented by one.

The loop variable is not allowed to be a input or an assignment variable, and may not be assigned within the loop. This guarantees termination of **for**.

for_statement :=
    for string_token = arith_expr , arith_expr component_statement

While Statement

While executes a statement as long as a logical expression remains TRUE. Multiple statements may be executed by the use of **grouping**.

While first evaluates the controlling logical expression. If it is TRUE, the target statement is executed, and **while** is reexecuted. If it is FALSE, execution continues at the statement immediately following the **while**.

Termination of **while** is not guaranteed. There is no check for non-termination.

while_statement :=
    while log_expr component_statement

Break Statement

Break is used to halt processing of **for** and **while** statements. **Break** disregards pending statements in the current target component, and continues execution at the statement immediately following the current **for** or **while** statement.

Break takes an argument which is the number of nested loops to break. A nonpositive argument has no effect. If the argument is larger than the number of nested loops, creation of the module is completed at the **break**. **Break** does not affect parent modules.

```
break_statement :=
        break  arith_expr ;
```

Chapter 3

Command Syntax

The command syntax controls what actions are taken. These commands control the definition and execution of circuit models.

Commands are normally read from standard input. They may be directed from a file by using an input flag or a command statement.

3.1 Filenames

A special syntax is accepted to facilitate the use of filenames. Filenames are allowed to be string tokens separated by periods '.'. This allows specification of most local filenames without having to use quoted strings.

```
file_name :=
        string_token
        | string_token . file_name
```

Quoted strings must be used in order to use the UNIX directory structure.

3.2 Current Generated Module

The name of the last generated module to be referenced is saved. This is known as the *current generated module.* The current generated module is used when commands are issued which omit the optional module name. The current generated module is automatically set by **generate**, and may be changed using **set**.

3.3 Current Submodule

Each generated module has a single *current submodule*. The current submodule is used as a shorthand notation for a single submodule in each generated module. This allows simple reference to the submodule during testing.

The current submodule is referenced by beginning a command name with character '`@`'.

The current submodule of a simulation instance is initially the top level module generated. It may be changed using **set**.

3.4 Parent Constructor

The command naming syntax contains a parent constructor '^'. As each field is read in the left-to-right expansion of a hierarchical name, the partial name corresponds to an object in the current module. When the parent constructor is read, the new object referenced by the partial name is set to the parent of the current object.

The parent constructor is usually used in conjunction with the current submodule '`@`'.

Use of the parent constructor '^' with an array of child modules may cause problems.

3.5 Command Naming

Names in the command syntax are similar to hierarchical names in definitions. There are additional rules which apply to command names:

- The name of the current generated module or the current submodule identifier '`@`' must be the first field in the hierarchical name.
- There is a *parent constructor* '^' which changes the target to the parent of the current target.

We now define an object name in the command syntax.

```
command_object :=
        @
        | string_token
        | @ . command_object_tail
        | string_token . command_object_tail
```

```
command_object_tail :=
        ^
        | object_name
        | ^ . command_object_tail
        | object_name . command_object_tail
```

We also define a list of command objects defined by array expansion. This corresponds to the hierarchical list in the definition syntax.

```
command_list :=
        @
        | string_token
        | @ . command_list_tail
        | string_token . command_list_tail

command_list_tail :=
        ^
        | object_name
        | object_list
        | ^ . command_list_tail
        | object_name . command_list_tail
        | object_list . command_list_tail
```

Finally, a general list of command names is defined. Note that every command object is also a command list.

```
command_object_list :=
        command_list
        | command_list command_object_list
```

3.6 Design Tool Commands

command_statement :=	
source_statement	$ read in a definition file
| clear_statement	$ clear all current definitions
| generate_statement	$ generate a module for simulation
| run_statement	$ simulate a generated module
| reset_statement	$ reset signals in a generated module
| destroy_statement	$ destroy a generated module
| read_statement	$ read commands from a file
| close_statement	$ close an open command file
| pause_statement	$ transfer control
| assignment_statement	$ assign values to signals
| show_statement	$ display signal vectors
| showvector_statement	$ display signal vectors as a group
| timed_statement	$ execute a command during simulation
| set_statement	$ set options
| repeat_statement	$ loop read a command file
| quit_statement	$ exit the program

Source Statement

Source reads in a file of module definitions. The entire file is read using the stated definition language rules. Module syntax is checked as the definition file is read. Definitions are checked for consistency only when referenced by a **generate**.

source_statement :=
    source file_name ;

If **source** causes a module to be redefined, the new definition will be used only in future module definition instances. Previously defined instances will continue to use the previous definition.

Clear Statement

Clear deletes all definitions and simulation instances. This is equivalent to restarting the design tool.

clear_statement :=
    clear ;

Generate Statement

Generate creates a simulation instance of a module definition. The module should have been previously read using a **source**. The simulation model is used to test correctness of designs. Generating a module causes all consistency checks on submodule use to be performed. The consistency rules have been stated along with the definition syntax.

Each generated module is set to a special initial state in which all signals have an undefined value.

Gereration of a module causes it to become the current generated module.

generate_statement :=
    generate string_token [(argument_list)] ;

The string token specifies the module to be generated. The arguments given must match the number of input variables of the module.

Run Statement

Run executes a simulation run of a generated module. Events queued for the module are evaluated until all events have been processed or a halt command has been issued. **Run** is detailed in section 4.

run_statement :=
    run [string_token] ;

The string token specifies the generated module to be run. If omitted, the current generated module is run.

A simulation run may be aborted by an interrupt signal (control-C). Such an interrupt sets command input to the interactive level, or exits the design tool if it is being run in batch mode. Aborting a simulation run does not affect pending events.

Reset Statement

Reset causes a generated module to be set to its special initial state. All signals in the simulation instance are set to the unknown value and all events are removed from the event queue.

reset_statement :=
    reset [string_token] ;

The string token specifies the generated module to be reset. If omitted, the current generated module is reset.

Destroy Statement

Destroy frees a generated module which is no longer needed. Destroying a module does not affect any other generated modules or any definitions.

destroy_statement :=
    destroy [string_token] ;

The string token specifies the generated module to be destroyed. If omitted, the current generated module is destroyed.

If the current generated module is destroyed, it will be ill-defined until reset by a **set** or **generate**.

Read Statement

Read causes commands to be read from a file. The file is read until the end-of-file is reached, or a **pause** is executed in the file. Commands are again read from the current source following exit from the named file.

read_statement :=
    read file_name ;

The file is closed after a **read** if the entire file has been read. If the named file is already open, **read** continues at the current position in the file.

Close Statement

Close closes a file left open by a previous **read**. This allows a file containing a **pause** to be reread from the beginning of the file.

close_statement :=
    close file_name ;

Pause Statement

Pause stops reading of the current source of command input. Command input is then read from the previous source.

pause_statement :=
    pause ;

A **pause** in a command file causes reading of the file to stop. The file is kept open, and a subsequent **read** will continue at the command following the **pause**. In the interactive (top) level, **pause** exits the design tool.

Assignment Statement

Assignment sets a signal value in the current generated module. Assignment events are put into the event queue of the current generated module. These events are completed on the next **run** of the module.

assignment_statement :=
    command_signal_list <- numeric_token ;

The command signal list has been described under command naming.

The numeric token may be a binary, octal, hexadecimal, or decimal number. In all cases, the number is converted to a binary representation then assigned in order with the least significant bit assigned to to the rightmost signal. A 0 bit corresponds to logical low, while 1 corresponds to logical high. Only the logical high and low values may be assigned.

If the numeric value is binary, octal, or hexadecimal, the number of bits given must 'match' the number of signals to be assigned. This means that exactly the minimum number of data bits needed to assign a value to every signal must be given.

Assignment of a value to a bus is not recommended. Assignment of decimal values to signal lists longer than 32 bits is not supported.

Show Statement

Show prints the value of a list of signals in a generated module. Each signal is printed individually, giving the signal value and time of last change.

A signal may have the following values:

0	logical low
1	logical high
U	undefined
X	bad signal value
T	tri-state value

show_statement :=
    show command_signal_list ;

Show works differently when used with a bus. If a primitive output port onto the bus is named, the value of the port is given, otherwise the computed bus value is printed. This enables all inputs to a bus to be printed, as well as the bus value.

Showvector Statement

Showvector prints a numeric equivalent of the signal values for a list of signals. The list of values is interpreted as a binary number, with the least significant bit corresponding to the rightmost element. Logical low corressponds to a 0 bit, while logical high corresponds to a 1. This is consistent with **assignment** rules for signals. The resulting composite value is printed as a decimal number. If any signal has an abnormal value (not logical low or high), each signal is printed individually using **show**.

showvector_statement :=
    showvector command_signal_list ;

Showvector does not support signal lists containing more than 32 elements.

Timed Statement

Timed statements store commands in the event queue of a generated module for execution during a run. Only three types of statements may be used as timed statements: **assignment**, **show**, and **pause**. Timed statements have an initial argument which is the amount of simulation time to pass before the statement is executed. They are put into the event queue for execution.

```
timed_statement :=
        numeric_token : assignment_statement
        | numeric_token : show_statement
        | numeric_token : pause_statement
```

pause functions differently when used as a timed statement. A timed **pause** halts the current simulation run, returning control to the command level which initiated the run (not necessarily the level which produced the **pause**). This is similar to halting a run via an interrupt. The other statement types function normally.

Timed statements compute their target signals before being entered in the event queue. The present value of the current submodule is used for decoding '@'.

Set Statement

Set is used to change values used by the design tool. There are two things which may be changed with **set**: the current generated module, and the current submodule (of the current generated module).

The current generated module is the default used when certain operations do not specify a module name. The current submodule is used as the initial path object in command names which have '@' as the initial field.

```
set_statement :=
        set simulation string_token ;
        | set @ command_object ;
```

In the first variation, the string token refers to a simulation instance. This instance becomes the current generated module.

In the second, the new current submodule '@' is specified by the command object. The command object must be a module, not a signal or cable. The previous value of '@' may be used to specify the new object.

Repeat Statement

Repeat causes repetitive reading of a command file until a test passes. It reads a signal value as a test for completion once each time the file is read.

There are two versions of **repeat**: **while** and **until**.

> **While** checks the signal before reading the file, Execution continues as long as the signal value is logical high.
>
> **Until** reads the file before checking the test signal. It continues execution as long as the value is **not** logical high. **Until** always reads the file at least once.

Note that the two versions use inverse testing conditions.

repeat_statement :=
    repeat file_name **while** command_signal ; $ check before loop
    | **repeat** file_name **until** command_signal ; $ check after loop

If multiple tests are needed for the halting condition, the halting function must be designed in hardware.

There is no check for termination of **repeat** statements.

Quit Statement

Quit causes normal termination of the design tool. No state is retained between execution.

quit_statement :=
    quit ;

Chapter 4

Implementation Details

A simulation run iteratively executes primitive modules affected by changes to their input signals, then updates the value of their output signals. This continues until the simulation instance reaches a steady state, or a halt command is processed.

Each event in a simulation instance has an associated integer *processing time*. Events with the same processing time are completed in a single time step, and are processed before any event with a greater processing time. The last processing time executed is known as the *current processing time*. Simulating in time steps allows the current processing time to serve as an indicator of the amount of time a circuit takes to execute.

Following are specific implementation details of the design tool:

4.1 Primitive Modules

Primitive modules perform functions predetermined by C code. These modules have a uniform delay characteristic $\delta \geq 1$, meaning that a change on any of its inputs causes a change in its outputs exactly δ time units in the future.

The delay characteristic must be positive to satisfy the processing time requirement.

Uniformity ensures consistency in the output of a primitive module. Uniformity is needed because the simulation model does not throw out events. If the delay characteristic was nonuniform, a single module could cause schedule signal value changes on the same wire out of order.

4.2 Simulation Construction

To speed simulation, generated modules are flattened. *Flattening* removes the definition hierarchy from a simulation instance. Only instances of primitive modules and connections between them remain after flattening. This speeds execution, since the definition hierarchy is not traversed during simulation. Flattening constructs connection lists for each signal that specify which primitive instances affect it and are affected by it.

The definition hierarchy is retained and is used to reference the flattened structure.

4.3 Bus Signals

Bus signals, which are driven by tri-state drivers, are built in a special way. Each primitive module on a bus writes to a specific entry point, similar to a port of a module. The bus value is calculated based on the values of its entry points. Every primitive module reading from the bus gets the calculated bus value.

Busses also have special handling for printing. A bus name which corresponds to an output of a primitive module prints information about the corresponding entry point. Any other name corresponding to the bus prints information about the calculated bus value. This allows for easier examination of busses.

4.4 Simulation Events

In order to satisfy the processing time requirement, events are stored in and read from a priority queue. This is implemented in the design tool by a heap.

The queue contains three types of events: signal value, printing and halting.

- Signal value events specify changes in the value of a signal. These events cause affected primitive modules to be executed.
- Printing events cause printing of signal information.
- Halting events stop execution of a simulation run following the current time step, instead of waiting until stable state.

Events may be created by a command statement, or as an effect of executing a primitive module.

4.5 Simulation Runs

Each simulation run reads and processes events until all events have been processed or a halting command has been processed.

Each time step of the run is conducted in phases.

1. All current events are extracted from the priority queue.

- Signal value events cause the target signal to be immediately updated. Each update causes connected busses and primitive modules to be scheduled for evaluation. Modules and busses are kept in separate evaluation lists. If a signal is updated more than once in a single time unit, an error message is printed.
- Printing events get stored in a list for later processing.
- A halting command sets a flag to exit the simulation run following the current time unit.

2. Busses scheduled in the first phase are evaluated, based on the value of all signals connected to it. This may cause schedule additional primitive modules for evaluation.
3. Each primitive module in the evaluation list is processed. The C code for each affected module is executed. This may change internal state and may schedule additional simulation events. Because of the delay characteristic of primitive modules, events are always scheduled for a later processing time.
4. Printing commands are executed. This shows the signal state at the end of the current processing time.

After these phases are completed, the simulation stops if the halting flag is set. Otherwise, the next time step is processed.

Chapter 5

Startup Options

The design tool has a number of options which are set at the beginning of execution. They are separate from command statements and do not change during execution. These options control general input and output characteristics of the design tool.

Normally commands are read from `stdin` and output is written to `stdout`. Error messages are directed to `stderr`. Input and output may be redirected using execution arguments. Error messages may not be redirected.

5.1 Command-line Arguments

A number of options may be set on the command line.

sim [option_list]

Acceptable command line options are:

-i <filename> Read commands from the named file instead of `stdin`. This causes batch mode execution, rather than interactive.

-o <filename> Direct output messages to the named file instead of `stdout`. Output messages result from command statements, specifically the printing statements (**show** and **showvector**). An output file should only be specified when in batch mode (**-i**).

-n Turn off debugging messages. Debugging messages are useful in verifying a circuit design. Debugging causes the design tool to print additional information about each created circuit and signal information each time a signal changes value.

5.2 simrc File

Upon startup, additional information is read from a file named **simrc**, which should be in the current directory upon program execution.

Comments in **simrc** are specified by the pound sign '#', similar to other syntax rules. Numbers in **simrc** are interpreted using the C "strtol" function. These do not conform to the conventions used in other parts of the design tool.

There are four options which may be specified in **simrc**:

Debugging Messages

Debugging messages may be supressed with the single keyword **no_print**. This produces the same result as the **-n** command-line argument.

Fanout

Fanout is a crude measure of the drive/load ratio on signals. Signals with large numbers of inputs or outputs are more likely to have load problems. An rough estimate of signal load is produced by comparing the number of inputs and outputs of each signal to a user-specified number. A warning message is printed for each signal which has a fan-in or fan-out greater than the fanout value.

The fanout value is specified with the keyword **max_fan**, followed by an integer. The number should use C syntax.

Error Printing

Error messages may be supressed by specification of an error printing mask. Only errors specified by the mask get printed. The list of error types and their corresponding mask numbers are shown in appendix D.

The error printing mask is specified with the keyword **print_mask**, followed by an integer. The number should use C syntax.

Error Halting

Execution of the design tool may be halted by use of an error halting mask. Encountering an error specified by the mask causes the design tool to exit. The list of error types and their corresponding mask numbers are shown in the appendix D.

The error halting mask is specified with the keyword **halt_mask**, followed by an integer. The number should use C syntax.

Any error specified by the halting mask always prints before exiting, even if it is not specified for printing (**print_mask**).

Appendix A

The Lexer

A lexer is used to convert input into tokens.

The lexer recognizes four primary types of tokens:
single character tokens
string tokens
reserved words
numeric tokens

The lexer uses spacing characters (space, tab, newline) to separate tokens, but they are not passed along.

Comments

The character ‘#’ is used to signify a comment. When a comment character is read, the remainder of the input line (until the next newline) is disregarded. Commenting does not work within a quoted string.

Single Character Tokens

The single characters tokens recognized by the lexer are:
‘,’, ‘:’, ‘;’, ‘=’, ‘!’, ‘<’, ‘>’, ‘(’,
‘)’, ‘[’, ‘]’, ‘{’, ‘}’, ‘-’, ‘+’, ‘*’,
‘/’, ‘%’, ‘.’, ‘@’, ‘^’, ‘|’, ‘&’

Single character tokens do not need to be separated from other tokens by spacing characters.

Non-alphanumeric characters which are not single character tokens or one of the special characters ‘_’, ‘#’, and ‘"’ are disregarded.

String Tokens

The lexer recognizes two types of string tokens: quoted and unquoted.

An *unquoted string* consists of an initial alphabetic character or underscore '_' followed by any number of alphanumeric characters or underscores. Unquoted strings are checked against the list of reserved words. If an unquoted string matches a reserved word, it is passed to the simulator as the reserved word token.

A *quoted string* is a succesion of characters enclosed within two delimiting quote symbols '"'. Quoted strings allow acceptance of strings which do not qualify as unquoted strings. This is used for filenames and message printing. A quoted string may not cross a line boundary. Quoted strings are not checked against reserved words, so they are always passed as string tokens.

Here is the string syntax given as regular expressions:

```
unquoted_string  := [a-zA-Z_][a-zA-Z_0-9]*
quoted_string    := "?*"
```

In the regular expressions, square brackets denote a choice between characters. '?' represents any single character. '*' means a sequence of zero or more of the previous character or choice of characters.

There is currently no way to pass a string containing the newline character.

Reserved Words

Reserved words are strings which have special meaning in the design tool. Each unquoted string read by the lexer is checked against the list of reserved words. If a string matches a reserved word, it is passed as the reserved word.

There are two categories of reserved words. The first is used when reading definitions, the other when reading commands.

Reserved Definition Words:

`break`	`cable`	`cmos`	`components`	`cost`
`else`	`end`	`error`	`for`	`gateInputs`
`if`	`inout`	`input`	`join`	`module`
`nmos`	`output`	`ports`	`signals`	`ts_inout`
`ts_output`	`while`			

Reserved Command Words:

`clear`	`close`	`destroy`	`generate`	`pause`
`quit`	`read`	`repeat`	`reset`	`run`
`set`	`show`	`showvector`	`simulation`	`source`
`until`	`while`			

Numeric Tokens

Four types of numeric tokens are recognized by the lexer: binary, octal, decimal, and hexadecimal. These correspond to numbers in base 2, 8, 10, and 16 respectively.

Binary, octal, and hexadecimal numbers have '0' as their initial character. The second character specifies the base of the number.

- 'b' or 'B' specifies a binary number. This is followed by a sequence of the characters '0' or '1'.
- 'o' or 'O' specifies an octal number. This is followed by a sequence which may contain characters corresponding to the numbers 0-8.
- 'x' or 'X' specifies a hexadecimal number. This is followed by a sequence which may contain characters corresponding to the numbers 0-9 or alphanumeric characters in the range a-f (upper or lower case). The characters a-f represent the decimal values 10-15 respectively.

If the second character does not fall into the above categories or if the leading character is a number which is not '0', the numeric token is a decimal number. A decimal number is a sequence of characters, each of which corresponds to a number in the range 0-9.

Each syntax is repeated below as a regular expression.

```
binary_number        := 0[bB][01]*
octal_number         := 0[oO][0-8]*
hexadecimal_number   := 0[xX][0-9a-fA-F]*
decimal_number       := [0-9][0-9]*
```

In the regular expressions, square brackets denote a choice between characters. '*' means a sequence of zero or more of the previous character or choice of characters.

All types of numeric tokens are interpreted as having the most significant digit on the left.

Appendix B

Primitive Modules

This appendix contains the current list of predefined primitive modules. Each primitive is shown as a module definition, with associated costs, and is accompanied by a short description.

Improper input values to primitive modules cause uncertain results to occur. These results should not be relied upon. The following rules generally apply:

- **bad** signal values propagate.
- If no **bad** signals are present, **undefined** signals propagate.
- If no **bad** signals are present, **tri-state** signals cause **undefined** output.

Because primitive modules are specially defined, some of their functions cannot be reproduced by general composite modules.

Constant

const allows signals to be hooked to a constant source. The input argument becomes the source value. Valid argument values are '0' (logical low) and '1' (logical high). Use of other values is not recommended.

Constant values cause attached modules to execute on the first run following module generation and after a simulation instance has been reset.

```
module const(v)
# the constant has zero costs
cost nmos: 0  cmos: 0  gateInputs: 0
   ports
    v output
end
```

Inverter

inv does a logical inversion of the input signal. Valid input values for "a" are logical low and high.

- If "a" is low, "x" is set to high.
- If "a" is high, "x" is set to low.

```
module inv
cost nmos: 2  cmos: 2  gateInputs: 1
  ports
    a input
    x output
end
```

Logical NAND

nand computes the logical NAND of the input signals. Valid input values for the inputs "a" and "b" are logical low and high.

- If either signal is low, "x" is set to high.
- If both signals are high, "x" is set to low.

```
module nand
cost nmos: 3  cmos: 4  gateInputs: 2
  ports
    a input
    b input
    x output
end
```

Logical NOR

nor computes the logical NOR of the input signals. Valid input values for the inputs "a" and "b" are logical low and high.

- If either signal is high, "x" is set to low.
- If both signals are low, "x" is set to high.

```
module nor
cost nmos: 3  cmos: 4  gateInputs: 2
  ports
    a input
    b input
    x output
end
```

Delay

delay simply propagates a signal value with a time delay. The output signal is set to the input signal, regardless of the value. The input argument is the time to delay the output, which must be a positive number.

The cost of a **delay** is represented as a pair of inverters.

```
module delay(delta)
cost nmos: 4  cmos: 4  gateInputs: 1
  ports
    d input
    q output
end
```

Transmission Gate

trans_gate sets the output "q" to the value of the input "d" when enabled with the enable signals "e1" and "e2". When not enabled, "q" is set to the tri-state value. The transmission gate is a dual-rail model, which means "e1" should always be the logical inverse of "e2".

- When "e1" is high ("e2" is low), "q" gets the value of "d".
- When "e1" is low ("e2" is high), "q" gets the tri-state value.

"q" must be hooked to a bus signal. This means that all ports which output to the bus must be typed as tri-state. In particular, only **trans_gate** outputs and **SRAM** data lines may output to the same signal as "q".

```
module trans_gate
cost nmos: 1  cmos: 2  gateInputs: 2
  ports
    d input
    e1 input
    e2 input
    q ts_output
end
```

Positive Latch

posLatch is a single bit of non-volatile memory. It uses “l” to control when data is read into memory from “d”. The value in memory is output through “Q”.

- When “l” is low, the latch holds state.
- When “l” is high, the memory value (and “Q”) is set to the value of “d”.

```
module posLatch
cost nmos: 8  cmos: 10
  ports
    d input
    l input
    Q output
end
```

Negative Latch

negLatch is a single bit of non-volatile memory. It uses “l” to control when data is read into memory from “d”. It is called a negative latch (as opposed to positive latch) because the sense of the latch signal “l” is reversed. The value in memory is output through “Q”.

- When “l” is low, the memory value (and “Q”) is set to the value of “d”.
- When “l” is high, the latch holds state.

```
module negLatch
cost nmos: 8  cmos: 10
  ports
    d input
    lb input
    Q output
end
```

Static RAM

SRAM is memory for simulation instances. A static RAM module takes two arguments: the amount of memory and the number of bits in the word. It reads and stores data in addressable memory based on its control signals “rw” and “e”. “e” enables the RAM for an operation, and “rw” selects whether the operation reads from or writes to memory.

- If “e” is low, the memory does nothing.

- If "e" is high, the memory does the specified operation.
- If "rw" is low, the operation is a write.
- If "rw" is high, the operation is a read.

The signals in "D" must be hooked to busses. This means all ports which output to each bus must be typed as tri-state. In particular, only **trans_gate** outputs and other **SRAM** data lines may output to those busses.

There is currently no way to preload data into the memory. All data must be written to memory before it is used.

Data widths larger than 32 bits are not supported.

```
module SRAM(amount, width)
  ports
    rw input
    e  input
    A[1:amount] input
    D[1:width]  ts_inout
end
```

Dynamic Memory Test

D_test is used to simulate dynamic RAM in conjunction with the static RAM module **SRAM**. It keeps track of the last time data was written to the address, however **D_test** does not actually store the data. If data is used too long after it has last been written, an error message is generated.

The "rw" and "e" lines work as described for the static RAM.

```
module D_test(amount)
  ports
    rw input
    e  input
    A[1:amount] input
end
```

Appendix C

Module Examples

This section contains two simple examples which demonstrate certain features in the definition language. The examples have not been optimized.

The first example is a scalable multi-input OR. It is constructed using a two-input OR, which in turn is built from primitives **nor** (logical NOR) and **inv** (inverter). The multi-input OR uses recursion to construct a collection tree. This results in an $O(log(k))$ running time as opposed to $O(k)$ time for a chain.

```
module two_input_OR
  ports
    x y input
    z   output
  signals
    z_bar
  components
    xy_nor nor  x y     z_bar;
    z_comp inv  z_bar   z;
end

module k_input_OR(k)
# compute a multi-input OR by recursion
  ports
    x[1:k] input
    z      output
  signals
    z1 z2 # internal signals for split
  components
    if {k = 1} {
      join [ x[1] z ];  # connect input to output
      break (1);        # end current module; halt recursion
    }
```

```
# compute split information
    k1 <- k/2;
    k2 <- k - k1;

# split in half and recurse on both parts.
    z1_comp k_input_OR(k1)  x[1:k1]    z1;
    z2_comp k_input_OR(k2)  x[k1+1:k]  z2;

# recombine parts
    z_comp two_input_OR  z1 z2  z;
end
```

This example uses recursion to split the tree into two subtrees, and a **two_input_OR** to recombine the subtrees. Recursion is halted when the subtree has only a single input. This is done by using **break** to end the module definition.

The second example is a variable length MIN circuit. It uses a for loop to join a chain of single-bit MIN modules.

```
module two_input_AND
  ports
    x y input
    z   output
  signals
    z_bar
  components
    xy_nand nand  x y    z_bar;
    z_comp  inv   z_bar  z;
end

module a_gre_b
# set z to one if a is greater than b ((a = 1) & (b = 0))
  ports
    a b input
    z   output
  signals
    b_bar
  components
    b_inv  inv   b  b_bar;
    z_comp two_input_AND  a b_bar  z;
end

module MIN
```

```
# compute MIN based on input values and selection inputs
# compute selection outputs for chaining
# (sxi = 1) -> choose x as the MIN
# (syi = 1) -> choose y as the MIN
# sxi = syi = 1 is an impossible state
  ports
    x y input
    z   output
    sxi syi input   # select control input
    sxo syo output  # select control output
  signals
    z1 z2 z3
    sxi_bar syi_bar
    x_gre_y y_gre_x  # used to find out which data input is greater
    sxo_e syo_e      # contains new select information
  components
# compute the min value z
    x_sel  two_input_AND  x sxi  z1;
    y_sel  two_input_AND  y syi  z2;
    xy_and two_input_AND  x y    z3;
    z_comp k_input_OR(3)  z1 z2 z3  z;


    sxi_inv inv  sxi  sxi_bar;
    syi_inv inv  syi  syi_bar;

# check if values are not equal
    x_gre_y_comp a_gre_b  x y  x_gre_y;
    y_gre_x_comp a_gre_b  y x  y_gre_x;

# compute new select information
    sxo_e_comp two_input_AND  y_gre_x syi_bar  sxo_e;
    syo_e_comp two_input_AND  x_gre_y sxi_bar  syo_e;

# compute output selects
    sxo_comp two_input_OR  sxi sxo_e  sxo;
    syo_comp two_input_OR  syi syo_e  syo;
end
```

MIN uses information from the input select lines or by comparing the two signals x and y to compute the output z and the output select lines. Note that reversing the order of signals connected to the ports of the circuit **a_gre_b** changes its function.

```
module k_bit_MIN(k)
# compute a variable length MIN circuit by iteration of
# a chainable single bit MIN
  ports
    x[1:k] y[1:k] input
    z[1:k] output
  signals
    sx[0:k] sy[0:k] low
  components
# Turn off initial select signals
    low_gen const(0)  low;
    join [ low sx[0] sy[0] ];

# Chain MIN circuits together
    for i = 1,k
      bit[i] MIN  x[i] y[i]  z[i]  sx[i-1] sy[i-1]  sx[i] sy[i];
end
```

The chain is initialized by connecting the first set of select inputs to the low signal. The last set of select outputs is left unconnected.

Appendix D

Error Messages

Error messages each have an associated field which describes its type. The error type is used to identify groups of messages for special consideration. Upon startup, a print mask and a halt mask are read from the **simrc** file. The print mask specifies error types which are printed. The halt mask specifies error types which halt design tool execution. Each message that halts execution is automatically printed.

Following is a list of error masks and their associated groupings. Each mask is given as an octal constant.

000001L	Race condition during a simulation run
000002L	Corrected parsing error
000004L	Warning
000010L	Redefinition of a cable or module
000020L	Reference to undefined cable or module
000040L	Conflicting definitions
000100L	Uncorrectable parsing error
000200L	Module generation halted
000400L	Error in primitive module
001000L	Bad data found
002000L	**Error** statement executed
004000L	Memory allocation error
010000L	Error external to program
020000L	Inconsistency in program

September 91

DESIGN AND ANALYSIS OF EFFICIENT HIERARCHICAL INTERCONNECTION NETWORKS*

S. Wei and S. Levy

LCSR-TR-167

Laboratory for Computer Science Research
Hill Center for the Mathematical Science
Busch Campus, Rutgers University
New Brunswick, New Jersey 08903

*This work was supported by the Defense Advanced Research Projects Agency and the National Aeronautics and Space Administration under NASA-Ames Research Center grant NAG 2-668.

Design and Analysis of Efficient Hierarchical Interconnection Networks

Sizheng Wei and Saul Levy

Department of Computer Science
Rutgers University
New Brunswick, NJ 08903

Footnotes

The authors are with the Department of Computer Science, Rutgers University, New Brunswick, NJ 08903.

This work was supported by the Defense Advanced Research Projects Agency and the National Aeronautics and Space Administration under NASA-Ames Research Center grant NAG 2-668.

Index Terms — Hierarchical networks, Interconnection networks, Multicomputer systems, Performance analysis, Traffic distributions.

Abstract

Hierarchical interconnection networks exploit locality in communication to reduce the number of links in the networks. In this paper, we propose a class of general hierarchical interconnection networks for message-passing architectures which are designed using a new approach — choosing the appropriate number of interface nodes and the appropriate size of clusters based on performance and cost-effectiveness measures. We show that the approach considerably reduces not only intercluster traffic density but also intracluster traffic density. It also enhances the fault tolerance capability of the networks. We present a performance analysis of the hierarchical networks based on both static and queueing analyses.

The asymmetric topology of a hierarchical network may degrade performance and reliability because of some heavy traffic links. The traffic distributions in hierarchical networks are thus important, but so far there has been very little analysis of the problem. We analyze the traffic distribution on two-level networks and try to reveal the relationship between traffic density and other performance and cost-effectiveness measures. In addition, we investigate in detail how to construct a cost-effective hierarchical network by setting appropriate design parameters. An associated algorithm is developed.

1 Introduction

Multicomputer systems with hundreds or thousands of processors are expected to have the most potential for the next generation of supercomputers. The interconnection networks in these multicomputer systems play a very important role in determining system performance. Message-passing organization is preferable for these systems due to the simplicity of communications among processors [7] [9] [16] [18].

For a very large system, a critical problem of the interconnection network is that the number of links needed becomes prohibitively large. To tackle the problem, *hierarchical interconnection networks,* which exploit locality in communication to reduce the number of links, have been proposed in the literature. Some examples are Hierarchical Interconnection Networks (HINs) [5], Hypernet [10], Hierarchical Cubic Network (HCN) [8], a cluster structure using shared busses as the basic interconnection media [20], Hierarchical Memory Structure (HMS) using crossbar switches [14], and a two-level mesh hierarchy scheme [3]. Also several systems have been made with hierarchical interconnection networks, such as Cm* [19] and Cedar [12]. Among these networks, most are based on some specific topologies such as hypercube, mesh, bus, etc. Few analyses have been made for general hierarchical networks.

The HINs proposed in [5] are a class of hierarchical networks for message-passing systems. A HIN is constructed in the following way: All nodes in the system are grouped into clusters and each cluster of nodes is linked internally by a level 1 network. One node is selected from each cluster to act as an interface node. These interface nodes may be linked by a level 2 network, or they may be themselves grouped into clusters, with each cluster linked by a separate level 2 network. In the latter case, one node from each cluster at level 2 is selected as an interface node to construct the level 3 network, and so on. A performance analysis was given in [5] for several examples of two-level HINs. It was shown that a HIN is more cost-effective than its non-hierarchical network counterpart if locality in communication exists, i.e., the HIN gains more performance benefit per unit cost. The authors also indicated the disadvantages of HINs, including high traffic density over intercluster links and some intracluster links (a potential degradation in performance) and diminished fault tolerance capability because of the single interface node in each cluster. Replication of intercluster links and more sophisticated routing algorithms were suggested to alleviate the congestion on these links.

In this paper, we propose a class of general hierarchical interconnection networks for message-passing architectures which are designed using a new approach. Unlike the HINs in [5], the proposed hierarchical networks are allowed to select any number of nodes from each cluster as interface nodes. The optimal number of interface nodes and the optimal size of clusters are determined based on performance and cost-effectiveness measures. We will show that increasing the number of interface nodes in each cluster using the same number of links not only reduces the same amount of intercluster traffic density as replication of links does, but also reduces a considerable amount of intracluster traffic density, so that the intracluster traffic can be balanced. In addition, it enhances the fault tolerance capability of the networks.

A major problem with a hierarchical network is that the network is usually asymmetric even if each cluster is symmetric, which results in some heavy traffic links that may become potential communication bottlenecks. Where would congestion take place and how can it be alleviated? What is the relationship between traffic density and other performance and cost-effectiveness measures? To answer these questions one must analyze traffic distributions in hierarchical networks which is difficult. Therefore, one of the objectives of this work is to analyze the traffic distributions so that we can gain a better insight into hierarchical networks.

We evaluate the performance of the proposed networks in terms of diameter, average internode distance, traffic density over links, and queueing delay with contention. We also analyze in detail how to design a cost-effective hierarchical network by choosing appropriate design parameters. An associated algorithm is developed.

This paper is organized as follows: Section 2 outlines the construction of the proposed hierarchical networks. In Section 3, performance and cost-effectiveness measures for the hierarchical networks are studied. Some examples of the hierarchical networks are analyzed and compared in Section 4. Section 5 analyzes how to determine the design parameters to construct a cost-effective network. Finally, the concluding remarks appear in Section 6.

2 Construction of hierarchical networks

The construction of the proposed hierarchical networks can be described as follows. Let N be the total number of nodes in a hierarchical network. The N nodes are divided into K_1 clusters of N/K_1 nodes each. Each cluster of N/K_1 nodes is connected to form a level 1 network. For convenience of analysis, we assume that K_i evenly divides K_{i-1} at level i, with initially $K_0 = N$, and every cluster at the same level is of the same size. The nodes in every cluster are ordered in the same way, i.e., the corresponding nodes in different clusters have the same internal address. Then, I_1 nodes, $1 \leq I_1 \leq N/K_1$, from each cluster are selected to act as the interface nodes. To be symmetric, the same I_1 interface nodes from each cluster are selected. For example, if the i^{th} and j^{th} nodes in cluster 1 are selected, the i^{th} and j^{th} nodes in all other clusters are also interface nodes. There are total of $I_1 \times K_1$ interface nodes at level 1.

To construct level 2 networks, these interface nodes are first divided into I_1 groups. Each group consists of K_1 nodes which are from K_1 different clusters, i.e., one from each cluster with the same internal address. Note that all groups are independent, i.e., there is no connection between any pair of nodes which belong to different groups. Then, the K_1 nodes of each group are again divided into K_2 clusters of K_1/K_2 nodes each. Each cluster is connected to form a level 2 network, and I_2 nodes, $1 \leq I_2 \leq K_1/K_2$, are selected as the interface nodes to construct level 3 networks, and so on.

The interconnection networks used to construct clusters at different levels may have the same or different topologies. Here it is assumed that all clusters at the same level are of the same topology. There are K_1 clusters at level 1, I_1K_2 clusters at level 2, $I_1I_2K_3$ clusters at level 3, and so on. Some examples of proposed networks are shown in Fig. 1. Fig. 1 (a) is a two-level network with $N = 16$, $K_1 = 4$, $I_1 = 2$, and $K_2 = 1$, and each cluster at level 1 is a completely connected network (CC) and that at level 2 is a binary hypercube network (BH). In Fig. 1 (b), both levels are constructed using BH with $N = 32$, $K_1 = 4$, $I_1 = 2$, and $K_2 = 1$.

The HINs described in [5] are special cases of the networks described here, i.e., all I_i are equal to 1. In addition, some one-level networks can be constructed hierarchically. For example, an ordinary binary hypercube network of size N is a two-level network with $I_1 = N/K_1$ and $K_2 = 1$, where hypercube connection is used at both levels.

In the following analysis, all hierarchical networks are restricted to two levels. This is because the analysis for the two-level networks is relatively simple and the results can be extended to the networks with more levels. Also it is pointed out in [5] [6] that two is a pragmatic choice for the number of levels in the hierarchy. For a two-level network, we assume that K_2 is always 1, i.e., only one cluster for each group of interface nodes.

3 Performance and cost-effectiveness measures

We now analyze the performance and cost-effectiveness of the hierarchical networks. We first give some definitions and make some assumptions. Let

N — the total number of nodes in a hierarchical network;

$L^{(i)}$ — the total number of links at level i;

$L_c^{(i)}$ — the number of links in a cluster at level i;

L — the total number of links in a hierarchical network;

K_1 — the number of clusters at level 1;

I_1 — the number of interface nodes selected from each cluster at level 1;

$Dm^{(i)}$ — the diameter of a cluster at level i;

Dm — the diameter of a hierarchical network;

$AD^{(i)}$ — the average internode distance of a cluster at level i;

AD — the average internode distance of a hierarchical network;

$TD_{max}^{(i)}$ — the highest traffic density over links in a cluster at level i;

TD_{max} — the highest traffic density over links in a hierarchical network;

λ — the message generation rate at each node;

$\lambda_{link,max}^{(i)}$ — the message arrival rate at the level i links which cause the longest delay at this level;

$\mu^{(i)}$ — the message processing rate of each link at level i;

$W_{max}^{(i)}$ — the longest average delay at level i links;

W_{max} — the longest average delay at links in a hierarchical network;

p — the probability that the source and destination nodes of a message are in the same level 1 cluster. (For comparison, we adopt a similar message distribution model used in [5].) Thus, $(1-p)$ is the probability that the source and destination are in different clusters. The larger the value of p for a given cluster size, the stronger the locality of communication. It is assumed that both intracluster and intercluster communications are uniformly random, i.e., a source node sends an intracluster message to each other node in its cluster with equal probability and a source node sends an intercluster message to each node in other cluster with equal probability. Note that the case of a node sending messages to itself is excluded.

The four important performance measures — diameter, average internode distance, traffic density over links, and queueing delay with contention are derived below. We also give the cost-effectiveness measures and briefly show the fault tolerance capability of the hierarchical networks.

3.1 Diameter

The *diameter* of a network is the maximum internode distance between any two nodes. For a two-level hierarchical network, the diameter is

$$\begin{cases} Dm \le 2Dm^{(1)} + Dm^{(2)}; & \text{if } 1 \le I_1 < N/K_1 \\ \\ Dm = Dm^{(1)} + Dm^{(2)}; & \text{if } I_1 = N/K_1 \end{cases} \tag{1}$$

The formula above is derived based on the following facts:

i) $I_1 = 1$: If a cluster at level 1 is constructed by a symmetric network (the network looks identical when viewed from any of its vertices) such as hypercube and ring, the $Dm^{(1)}$ is the distance between the interface node and the node farthest away from it. Considering the two clusters for which their interface nodes at level 2 are farthest away from each other, we can easily find that $2Dm^{(1)} + Dm^{(2)}$ is the diameter of the hierarchical network. For an asymmetric network at level 1 such as a binary tree, the $Dm^{(1)}$ may be greater than the distance from the interface node to any other node in the cluster. In this case, $Dm \le 2Dm^{(1)} + Dm^{(2)}$.

ii) $2 \le I_1 < N/K_1$: Usually $Dm \le 2Dm^{(1)} + Dm^{(2)}$ because more interface nodes give more alternative paths between any two nodes. However, for some type of networks such as completely connected network used at level 1, the distance between two non-interface nodes in the two clusters which are farthest away from each other is always $2Dm^{(1)} + Dm^{(2)}$. From i) and ii), we have the inequality above.

iii) $I_1 = N/K_1$: In this case, every node is an interface node. A message from a source node in cluster i to its destination node in cluster j, $i \neq j$, can go through a path which does not include any link in cluster i. Thus, the diameter is at most $Dm^{(1)} + Dm^{(2)}$. On the other hand, if cluster i is farthest away from cluster j and the position of the destination node in cluster j corresponds to that of the node in cluster i which is farthest away from the source node, the distance between the source and destination is at least $Dm^{(1)} + Dm^{(2)}$. The equation above is thus proved.

3.2 Average internode distance

Like diameter, *average internode distance* is a fundamental property of a topology. Average internode distance is the expected number of link traversals a "typical" message needs to reach its destination. It is a better indicator of message delay than the diameter [17]. Average internode distance depends on the message distribution which describes the probability of message exchanges among different nodes.

A hierarchical network is usually not symmetric even if the networks used to construct clusters at each level are symmetric. As a result, the average internode distance from different source nodes to all other nodes can be different. For example, the average internode distance from an interface node to all other nodes will be shorter than that from a non-interface node. However, if we take the average of the average internode distances over all nodes, the average internode distance of a two-level hierarchical network can be computed as follows:

$$\begin{cases} AD \le p \cdot AD^{(1)} + (1-p)(2AD^{(1)} + AD^{(2)}); & \text{if } 1 \le I_1 < N/K_1 \\ AD = p \cdot AD^{(1)} + (1-p)(AD^{(1)} + AD^{(2)}); & \text{if } I_1 = N/K_1 \end{cases} \tag{2}$$

The derivation of the formula is similar to that of the diameter.

3.3 Traffic density

Traffic density over links is another important performance measure which reflects link utilization. The analysis of traffic density is important, especially for asymmetric networks, because this measure can indicate potential communication bottlenecks. So, it may be a better performance measure than diameter or average internode distance for asymmetric networks. Low traffic density is preferable. Traffic density is measured in terms of the average number of messages per link per unit time, given that each node issues one random message per unit time (Here "random" means that the destination distribution of the messages is uniform).

Since a hierarchical network may be asymmetric, the traffic density over each link in a cluster can vary. Also the traffic density over a link at level 1 may be different from that over a link at level 2. Here the analysis is concentrated on the links with the highest traffic density, because they are potentially the bottlenecks and they determine the worst case in communication delay. For simplicity, it is assumed that the networks used to construct clusters at each level are symmetric. Note that traffic density is related to the traffic distribution pattern and the routing algorithm employed for the network.

The highest traffic density over the links at level 1: It is easy to see that the links directly connecting interface nodes would be the links with the highest traffic density (From now on, we only consider the traffic density over these links). The traffic density, $TD^{(1)}_{max}$, over these links consists of three parts:

$TD^{(1)}_{local}$ — Traffic density generated by intracluster communications;

$TD^{(1)}_{out}$ — Traffic density generated by outgoing messages to other clusters;

$TD^{(1)}_{in}$ — Traffic density generated by incoming messages from other clusters.

When the message distribution within a cluster is uniform, i.e., when every node in the cluster generates a random message per unit time and every message behaves statistically identically, $TD^{(1)}_{local}$ can be computed using $AD^{(1)}$:

$$TD^{(1)}_{local} = \frac{pN}{K_1} \times \frac{AD^{(1)}}{L^{(1)}_c} \tag{3}$$

which means that the total number of traversals made by all the pN/K_1 messages are shared equally by $L^{(1)}_c$ links in the cluster. Thus, $TD^{(1)}_{local}$ over every link in the cluster is the same.

To calculate the traffic density generated by intercluster messages, it is necessary to specify a routing strategy first. A natural and simple routing strategy is to divide a cluster into I_1 disjoint subclusters of equal size (e.g., divide a cube into subcubes), each containing an interface node. When a node wants to send a message to another cluster, it first sends the message to the interface node in its subcluster. In this way, the interface node in a subcluster is responsible for sending out the messages generated by all the nodes in the subcluster. For the incoming messages from other clusters, however, the interface node has to forward them to all the nodes in the whole cluster. Since each cluster is symmetric, the amount of traffic through each interface node is the same.

Based on this routing strategy, the $TD^{(1)}_{out}$ over a link connecting an interface node is

$$TD^{(1)}_{out,l} = (1-p)(\frac{N}{K_1 I_1} - 1)q_l, \quad 1 \leq I_1 \leq \frac{N}{K_1} \tag{4}$$

where l represents the l^{th} link connecting the interface node and q_l is a fraction that gives the percentage of outgoing messages through the l^{th} link over all the outgoing messages generated by the subcluster (except the interface node). The range of the values of l depends on the network topology, but it is at least 1. The value of q_l depends on both network topology and routing algorithm. If the l links equally share the outgoing traffic, all values of q_l are identical. Note that when $I_1 = N/K_1$, $TD^{(1)}_{out,l} = 0$ because every node is an interface node and no outgoing message needs to go through the intracluster links.

The $TD^{(1)}_{in}$ over the l^{th} link connecting an interface node is

$$\begin{cases} \frac{1-p}{I_1}(\frac{N}{K_1}-1)q_l' \le TD^{(1)}_{in,l} \le (1-p)(\frac{N}{K_1}-1)q_l' \; ; & \text{if } 1 \le I_1 < N/K_1 \\ TD^{(1)}_{in,l} = (1-p)(\frac{N}{K_1}-1) \times \frac{AD^{(1)}}{L_c^{(1)}} \; ; & \text{if } I_1 = N/K_1 \end{cases} \tag{5}$$

The q_l' is a fraction that gives the percentage of incoming messages through the l^{th} link over all the incoming messages via the interface node to the cluster (except the interface node). Note that q_l is just for a subcluster, while q_l' is for the whole cluster. The $TD^{(1)}_{in,l}$ above is derived as follows:

i) When $I_1 = 1$, all the incoming messages to this cluster must go through the interface node. There are total of $N - \frac{N}{K_1}$ nodes in other clusters and the probability of each node sending a message per unit time to the nodes in this cluster (except the interface node) is $(1-p)(\frac{N}{K_1}-1)/(N-\frac{N}{K_1})$. Thus, the $TD^{(1)}_{in}$ over the l^{th} link is $TD^{(1)}_{in,l} = (1-p)(\frac{N}{K_1}-1)q_l'$.

ii) When $2 \le I_1 < N/K_1$, the average number of incoming messages via an interface node, say A, is $\frac{1-p}{I_1}(\frac{N}{K_1}-1)$. However, the incoming traffic via other interface nodes of this cluster may also go through the links connecting to A, i.e., the paths from different interface nodes to all other nodes overlap. Calculation of the extra incoming traffic depends on the type of network and the routing algorithm. So, in general $TD^{(1)}_{in,l} \ge \frac{1-p}{I_1}(\frac{N}{K_1}-1)q_l'$ and also $TD^{(1)}_{in,l} \le (1-p)(\frac{N}{K_1}-1)q_l'$ which is the worst case of $I_1 = 1$.

iii) When $I_1 = N/K_1$, every node receives the same amount of incoming messages. Thus, $AD^{(1)}$ and $L_c^{(1)}$ can be used to calculate $TD^{(1)}_{in,l}$.

To find out the highest traffic density over the links at level 1, we must consider the possibility that the largest $TD^{(1)}_{out}$ and the largest $TD^{(1)}_{in}$ are not over the same link. Therefore, we have

$$TD^{(1)}_{max} = TD^{(1)}_{local} + \max_l(TD^{(1)}_{out,l} + TD^{(1)}_{in,l}) \; . \tag{6}$$

Note that for a given network, a lower bound of $TD^{(1)}_{max}$ is the $TD^{(1)}_{max}$ for $I_1 = N/K_1$, i.e., at that time all links at level 1 equally share the overall traffic, which is the best that any optimal routing algorithm can do.

The highest traffic density over the links at level 2: The message distribution on the links at level 2 is uniform because each level 2 network is assumed to be symmetric. Each node sends $\frac{(1-p)N}{I_1 K_1}$ messages per unit time and there are total of K_1 nodes in each cluster. Thus, we have

$$TD_{max}^{(2)} = \frac{(1-p)N}{I_1} \times \frac{AD^{(2)}}{L_c^{(2)}} . \tag{7}$$

The highest traffic density in a hierarchical network: Considering all links at both levels, we have the highest traffic density in a two-level hierarchical network, that is,

$$TD_{max} = \max(TD_{max}^{(1)}, TD_{max}^{(2)}) . \tag{8}$$

From Eqs. (4), (5), and (7), we can see that increasing the value of I_1 reduces both $TD_{max}^{(1)}$ and $TD_{max}^{(2)}$. Therefore, it is better than replication of the level 2 links [5] which reduces only $TD_{max}^{(2)}$.

3.4 Queueing analysis with contention

Queueing analysis with contention is a popular method of performance analysis which can provide a deeper insight into the behavior of a system. Similar to the analysis of traffic density, here we are interested in those links causing the longest average message delay (i.e., average queueing time + average service time), W_{max}. Note that our goal here is to obtain the "order of magnitude" evaluation and comparison of different networks rather than to accurately predict their performance under some particular workload. Therefore, to simplify the analysis, we use simple and well-known queueing models ($M/M/1$ and $M/M/r$) [11] [13] based on the following assumptions:

1) Packet-switching transmission is used. A message may consist of many packets. It is assumed that a single packet can be transferred between two nodes in unit time.

2) Each node generates messages independent of other nodes, at rate λ, and the intermessage times are distributed exponentially.

3) Each link is bidirectional. It delivers a message in either direction at a time, but messages coming from both directions are considered as the traffic over the link. Thus, each link is modeled as a queueing center.

4) Each level i link processes the messages at rate $\mu^{(i)}$ and the message service times are also distributed exponentially.

5) There is infinite buffer capacity, i.e., no message is dropped due to a full buffer. This assumption has been adopted by many previous analyses of networks [5] [15], because it makes the derivation of closed-form expressions easy and still provides a reasonable approximation to a real system, especially for the "order of magnitude" evaluation of the relative performance.

6) The routing strategy is the same as we used for the analysis of traffic density, i.e., when a node needs to send a message to another cluster, it first sends the message to the interface node in its subcluster.

The longest delay at level 1: Each level 1 link is modeled as an $M/M/1$ queueing center. It is easy to see that the $TD^{(1)}_{max}$ derived before can be directly used to derive the message arrival rate at the links which cause the longest delay at level 1 (The difference is that for traffic density we assume that on the average each node issues *one* message per unit time, while now we assume that each node issues λ messages per unit time). Thus, we have

$$\lambda^{(1)}_{link,max} = \lambda \cdot TD^{(1)}_{max} .$$

The longest delay at level 1 links is

$$W^{(1)}_{max} = \frac{1}{\mu^{(1)} - \lambda^{(1)}_{link,max}} = \frac{1}{\mu^{(1)} - \lambda \cdot TD^{(1)}_{max}} . \tag{9}$$

The longest delay at level 2: Each level 2 link is also modeled as an $M/M/1$ queueing center. Using $TD^{(2)}_{max}$ to compute the message arrival rate at level 2 links, we have the following result:

$$\lambda^{(2)}_{link,max} = \lambda \cdot TD^{(2)}_{max} ,$$

$$W^{(2)}_{max,mul} = \frac{1}{\mu^{(2)} - \lambda^{(2)}_{link,max}} = \frac{1}{\mu^{(2)} - \lambda \cdot TD^{(2)}_{max}} , \tag{10}$$

where the subscript "*mul*" is for the approach of choosing multiple interface nodes (we will use "*rep*" for replicating level 2 links).

For comparison, we also consider the delay for the approach of replicating level 2 links in [5]. Each group of r-replicated links can be modeled as an $M/M/r$ queueing center. $TD^{(2)}_{max}$ can still be used to compute $\lambda^{(2)}_{link,max}$, but we should always let $I_1 = 1$ when we compute $TD^{(2)}_{max}$ (It is also the same for computing $TD^{(1)}_{max}$ and $W^{(1)}_{max}$). Using the formulas for $M/M/r$ model [2], we have

$$\lambda^{(2)}_{link,max} = \lambda \cdot TD^{(2)}_{max} ,$$

$$u = \frac{\lambda^{(2)}_{link,max}}{\mu^{(2)}} = \frac{\lambda \cdot TD^{(2)}_{max}}{\mu^{(2)}} , \qquad \rho = \frac{\lambda^{(2)}_{link,max}}{\mu^{(2)} \cdot r} = \frac{u}{r} ,$$

$$C(r,u) = \frac{\frac{u^r}{r!}}{\frac{u^r}{r!} + (1-\rho)\sum_{n=0}^{r-1} \frac{u^n}{n!}} ,$$

$$W^{(2)}_{max,rep} = \frac{1}{\mu^{(2)}}\left[\frac{C(r,u)}{r(1-\rho)} + 1\right] , \tag{11}$$

where u and ρ are the traffic intensity and server utilization, respectively, and $C(r,u)$ is *Erlang's C formula*. In general, for the same traffic, an $M/M/r$ queueing system (receiving all messages) will cause shorter delay than r $M/M/1$ queueing systems (each receiving $1/r$ of total messages). However, since replicating level 2 links cannot reduce $TD^{(1)}_{max}$ and $W^{(1)}_{max}$ on level 1 links which can often be the bottleneck of a network, this approach may result in a longer delay in a network than choosing multiple interface nodes.

The longest delay in a network: The longest delay on the links in a network is the maximum of $W^{(1)}_{max}$ and $W^{(2)}_{max}$, that is,

$$W_{max} = \begin{cases} \max(W^{(1)}_{max},\ W^{(2)}_{max,mul})\ ; & \text{for multiple interface nodes} \\ \\ \max(W^{(1)}_{max},\ W^{(2)}_{max,rep})\ ; & \text{for replicated level 2 links} \end{cases} \tag{12}$$

3.5 Cost-effectiveness measures

Like the analyses in [5] and other articles, the cost of a hierarchical network is defined as the total number of links used, because one of our goals is to minimize the link cost of a network. The total number of links in a two-level hierarchical network is

$$L = L^{(1)} + L^{(2)} = K_1 L^{(1)}_c + I_1 L^{(2)}_c , \tag{13}$$

where $L^{(i)}_c$ is the number of links in each cluster at level i and $L^{(i)}$ is the total number of links at level i. Note that replication of links at level 2, say r-replicated, leads to the same L as that for $I_1 = r$.

In general, trying to minimize AD, TD_{max}, or W_{max} results in an increase in L, and vice versa. Thus, we adopt the products of L and AD, L and TD_{max}, and/or L and W_{max} as cost-effectiveness measures. Which one is more critical depends on applications and design considerations for a network. We will use all of the measures in the following analysis. A smaller value of $L \times AD$, $L \times TD_{max}$, or $L \times W_{max}$ is better.

3.6 Fault tolerance capability

A critical problem to interconnection networks of large size is fault tolerance. Since hierarchical networks are mainly for large systems, the fault tolerance capability of the networks must be considered. A common criterion used to measure the fault tolerance capability of interconnection networks is *full access*, i.e., the capability of a network that provides a connection from any of its input sources to any of its output destinations. Under the criterion of full access, a network is assumed to be faulty if there is any source-destination pair that cannot be connected because of faulty components in the network. A network is said to be *k-fault tolerant*, if it can still provide a connection for any source-destination pair in the presence of any instance of up to k faults in the network. The basic idea for fault-tolerance is to provide multiple paths for a source-destination pair so that alternate paths could be used in case of faults in a path. A faulty component can be a node or a link. Since a node-fault is usually more severe than a link-fault, we consider only node-faults here.

In a hierarchical network, the interface nodes are critical because they are the "bridges" connecting clusters. If there is only one interface node in each cluster, a failure of any interface node will disconnect the nodes in its cluster to all the nodes in other clusters. So, this kind of network cannot tolerate any node-fault, and is obviously not appropriate for large systems. By choosing I_1 interface nodes in each cluster, we can construct a network which may tolerate multiple (up to $I_1 - 1$) node faults, depending on its reconfiguration rule and routing algorithm. This means that a network constructed using our approach will be more reliable than that in [5]. Considering that the probability of multiple faults within a cluster is much smaller than that of a single fault, we can easily find that the reliability of a network is enhanced very quickly as I_1 increases.

4 Case studies of hierarchical networks

In this section, we analyze and compare some examples of hierarchical networks, based on the measures given in the last section, to find out how different structures (topologies) affect performance and cost-effectiveness of the hierarchical networks. The networks we choose are Binary Hypercube/Binary Hypercube (BH/BH) and Complete Connection/Binary Hypercube (CC/BH). For comparison, the ordinary binary hypercube (BH) is used as a reference network.

4.1 BH/BH networks

Let $N = 2^M$ be the total number of nodes in a BH/BH and $\frac{N}{K_1} = 2^m$ be the number of nodes in each cluster at level 1. So, there are $K_1 = 2^{M-m}$ clusters at level 1. I_1 interface nodes are selected from each cluster, where $1 \leq I_1 \leq 2^m$ and I_1 is assumed to be a power of 2. Since we consider two-level networks, K_2 is assumed to be 1. When $I_1 > 1$, we divide each cluster into I_1 subcubes and each subcube contains an interface node. Based on these assumptions, we have the following results:

1. The number of links:

$$L^{(1)} = K_1 L_c^{(1)} = m2^{M-1} ,$$

$$L^{(2)} = I_1 L_c^{(2)} = I_1(M-m)2^{M-m-1} ,$$

$$L_{BH/BH} = L^{(1)} + L^{(2)} = 2^{M-1}(\frac{I_1(M-m)}{2^m} + m) .$$

2. Diameter:

$$Dm_{BH/BH} = (m - \log I_1) + (M-m) + m = M + m - \log I_1 .$$

There are two special cases:

i) If $I_1 = 1$, $Dm_{BH/BH} = M + m$ which is the diameter of the BH/BH networks given in [5].

ii) If $I_1 = 2^m$, $Dm_{BH/BH} = M$ which is the diameter of the ordinary binary hypercube (BH) networks.

3. Average internode distance:

$$\begin{aligned} AD_{BH/BH} &\approx \frac{pm}{2} + (1-p)(\frac{m - \log I_1}{2} + \frac{M-m}{2} + \frac{m}{2}) \\ &= \frac{m}{2} + \frac{(1-p)(M - \log I_1)}{2} . \end{aligned}$$

4. Traffic density: We first consider the highest traffic density over the links at level 1, that is,

$$TD_{max}^{(1)} = TD_{local}^{(1)} + \max_l(TD_{out,l}^{(1)} + TD_{in,l}^{(1)}) ,$$

Since each cluster is symmetric, we have

$$TD_{local}^{(1)} = \frac{pN}{K_1 L_c^{(1)}} \times AD^{(1)} = \frac{p2^m}{m2^{m-1}} \times \frac{m2^{m-1}}{2^m - 1} \approx p .$$

To compute $TD_{out,l}^{(1)}$ and $TD_{in,l}^{(1)}$, we need to specify routing algorithms. Here we consider two different routing algorithms for comparison:

1) The routing algorithm which evenly distributes the incoming (outgoing) messages over all links connecting to the interface node in the cluster (subcluster): There are $m - \log I_1$ links involved in sharing the outgoing traffic and m links involved in sharing the incoming traffic, respectively. Thus, $q_l = 1/(m - \log I_1)$ and $q_l' = 1/m$, and

$$TD^{(1)}_{out,even} = \begin{cases} (1-p)(\frac{2^m}{I_1} - 1)\frac{1}{m - \log I_1}\,; & 1 \le I_1 < 2^m \\ 0\,; & I_1 = 2^m \end{cases}$$

$$\begin{cases} \max(1-p,\ \frac{(1-p)(2^m-1)}{I_1 m}) \le TD^{(1)}_{in,even} \le \frac{(1-p)(2^m-1)}{m}\,; & 1 \le I_1 < 2^m \\ TD^{(1)}_{in,even} = 1-p\,; & I_1 = 2^m \end{cases}$$

Note that $TD^{(1)}_{in,even}$ cannot be less than $1-p$ because now $TD^{(1)}_{in,even} = \max_l(TD^{(1)}_{in,l})$ and it should not be less than that for $I_1 = 2^m$. Then, for the routing algorithm, we have

$$TD^{(1)}_{max,even} = TD^{(1)}_{local} + TD^{(1)}_{out,even} + TD^{(1)}_{in,even}\,.$$

Two special cases are

i) $TD^{(1)}_{max,even} = p + \frac{2(1-p)(2^m-1)}{m}$ if $I_1 = 1$, and

ii) $TD^{(1)}_{max,even} = 1$ if $I_1 = 2^m$, which is equal to that for BH.

2) A fixed routing algorithm which is the simplest and most common routing algorithm used for hypercube systems: The routing code is computed as the bitwise Exclusive-OR of the source and destination addresses. The routing code is scanned from the most significant bit to the least significant bit. By tracing the routing algorithm, we can find that when $1 \le I_1 < 2^m$, more than half of the total outgoing messages from a subcluster ($\frac{2^{m-1}}{I_1}$ out of $\frac{2^m}{I_1} - 1$) go through the link connecting the interface node and the node whose address differs only in the least significant bit (e.g., the link between nodes 0000 and 0001 in Fig. 2 (a)). The next link (0000 — 0010) shares almost a quarter of the total outgoing traffic ($\frac{2^{m-1}}{2I_1}$ out of $\frac{2^m}{I_1} - 1$), and so on. Thus, if we define the l^{th} link as the link that connects two nodes whose addresses differ only in the l^{th} bit position (starting with the least significant bit as bit 0), we have

$$TD^{(1)}_{out,l} = \begin{cases} (1-p)\frac{2^{m-l-1}}{I_1}\,; & 1 \le I_1 < 2^m,\ 0 \le l \le m-1 \\ 0\,; & I_1 = 2^m \end{cases}$$

Moreover, we also find a similar situation for the incoming messages on these links but in reverse order. More than a half of total incoming messages to a cluster ($\frac{2^{m-1}}{I_1}$ out of $\frac{2^m}{I_1} - 1$) go through the link connecting the interface node and the node whose address differs only in the most significant bit (e.g., the link between nodes 0000 and 1000 in Fig. 2 (b)). The next link shares almost a quarter, and so on. If I_1 interface nodes are selected appropriately (e.g., for $2^m = 16$, we select 0000 and 1111 if $I_1 = 2$ (see Fig. 2 (b) and (c)) or 0000, 0101, 1010, and 1111 if $I_1 = 4$), we can also have

$$\begin{cases} \frac{(1-p)2^l}{I_1} \le TD^{(1)}_{in,l} \le (1-p)2^l \; ; & 1 \le I_1 < 2^m, 0 \le l \le m-1 \\ TD^{(1)}_{in,l} = 1 - p \; ; & I_1 = 2^m \end{cases}$$

It is obvious that $TD^{(1)}_{out,max}$ and $TD^{(1)}_{in,max}$ are not on the same link. By combining $TD^{(1)}_{out,l}$ and $TD^{(1)}_{in,l}$, we can find that

$$\begin{cases} \frac{(1-p)2^{m-1}}{I_1} + (1-p) \le \max_l(TD^{(1)}_{out,l} + TD^{(1)}_{in,l}) \le (1-p)(2^{m-1}+1); & 1 \le I_1 < 2^m \\ \max_l(TD^{(1)}_{out,l} + TD^{(1)}_{in,l}) = 1 - p; & I_1 = 2^m \end{cases}$$

which shows that $\max_l(TD^{(1)}_{out,l} + TD^{(1)}_{in,l})$ occurs on link 0. Here we have already included the incoming traffic via other interface nodes and through link 0, which is approximately $\frac{(1-p)(I_1-1)}{I_1}$. Thus, the highest traffic density over the links at level 1 for fixed routing algorithm is

$$TD^{(1)}_{max,fix} = TD^{(1)}_{local} + \max_l(TD^{(1)}_{out,l} + TD^{(1)}_{in,l}) \, .$$

Two special cases are

i) $TD^{(1)}_{max,fix} = p + (1-p)(2^{m-1}+1)$ if $I_1 = 1$, and

ii) $TD^{(1)}_{max,fix} = 1$ if $I_1 = 2^m$. Note that $TD^{(1)}_{max}$ is always 1 (the lower bound) when $I_1 = 2^m$, no matter what routing algorithm is used. This is true because at that time, the BH/BH is a BH and overall traffic within a cluster is perfectly balanced.

The traffic density over the links at level 2 can be derived as follows:

$$TD^{(2)}_{max,BH/BH} = \frac{(1-p)N}{I_1} \times \frac{AD^{(2)}}{L^{(2)}_c} = \frac{1-p}{I_1} \times \frac{2^M}{2^{M-m}-1} \, .$$

When $I_1 = 1$ and $I_1 = 2^m$, $TD^{(2)}_{max,BH/BH} \approx (1-p)2^m$ and $TD^{(2)}_{max,BH/BH} \approx 1 - p$, respectively. Finally, we have

$$TD_{max,BH/BH} = \max(TD^{(1)}_{max,BH/BH}, TD^{(2)}_{max,BH/BH}) \, ,$$

where $TD^{(1)}_{max,BH/BH}$ can be either $TD^{(1)}_{max,even}$ or $TD^{(1)}_{max,fix}$, depending on which routing algorithm is used.

Fig. 5 shows $TD_{max,BH/BH}$ for the two routing algorithms. We can see that the two algorithms yield comparable results, i.e., the fixed routing algorithm does not result in a significantly higher $TD_{max,BH/BH}$. This is because both algorithms can balance the intercluster traffic on the related links. To balance the overall traffic (i.e., both intercluster and intracluster traffic), we need to consider some other routing algorithms which can balance the link utilization within clusters, as described in [4]. However, these algorithms incur more overhead and make control complicated.

5. The longest average delay: The longest average delay in a BH/BH can be easily computed using $TD_{max,BH/BH}$ and the formulas derived in the last section. Since the two routing algorithms lead to comparable TD_{max}, we use $TD^{(1)}_{max,BH/BH} = TD^{(1)}_{max,even}$ to compute $W^{(1)}_{max,BH/BH}$. For $W^{(2)}_{max,BH/BH}$, we consider two cases based on two different approaches for comparison: one chooses multiple interface nodes (denoted as $W^{(2)}_{BH/BH,mul}$) and the other replicates level 2 links (denoted as $W^{(2)}_{BH/BH,rep}$).

1) The longest delay at level 1: Each link is modeled as an $M/M/1$ queueing center.

$$\lambda^{(1)}_{link,max} = \lambda \cdot TD^{(1)}_{max,BH/BH} ,$$

$$W^{(1)}_{max,BH/BH} = \frac{1}{\mu^{(1)} - \lambda \cdot TD^{(1)}_{max,BH/BH}} .$$

2) The longest delay at level 2:

i) Multiple interface nodes: Each link is modeled as an $M/M/1$ queueing center.

$$\lambda^{(2)}_{link,max} = \lambda \cdot TD^{(2)}_{max,BH/BH} ,$$

$$W^{(2)}_{BH/BH,mul} = \frac{1}{\mu^{(2)} - \lambda \cdot TD^{(2)}_{max,BH/BH}} .$$

ii) Replicated links ($I_1 = 1$): Each group of r-replicated links is modeled as an $M/M/r$ queueing center.

$$u = \frac{\lambda \cdot TD^{(2)}_{max,BH/BH}}{\mu^{(2)}} , \qquad \rho = \frac{u}{r} ,$$

$$C(r,u) = \frac{\frac{u^r}{r!}}{\frac{u^r}{r!} + (1-\rho)\sum_{n=0}^{r-1} \frac{u^n}{n!}} ,$$

C-2

$$W^{(2)}_{BH/BH,rep} = \frac{1}{\mu^{(2)}}\left[\frac{C(r,u)}{r(1-\rho)} + 1\right] .$$

3) The longest delay in a network:

$$W_{max,BH/BH} = \begin{cases} \max(W^{(1)}_{max,BH/BH}, W^{(2)}_{BH/BH,mul}) ; & \text{if choosing multiple interface nodes} \\ \max(W^{(1)}_{max,BH/BH}, W^{(2)}_{BH/BH,rep}) ; & \text{if replicating level 2 links} \end{cases}$$

Note that when replicating level 2 links, the values of $TD^{(1)}_{max}$ and $W^{(1)}_{max,BH/BH}$ should be computed with $I_1 = 1$. Fig. 6 shows $W_{max,BH/BH}$ for the two different approaches (for replicating level 2 links, I_1 means the number of replication) under $p = 0.5$ and $p = 0.8$, respectively. Since for given sizes and workload, the level 1 links of the BH/BH networks with replicated level 2 links saturate when $p < 0.7$, Fig. 6 (a) includes the results only for the BH/BH networks which are allowed to have multiple interface nodes. Also when $I_1 \leq 2$ (with $p = 0.5$) and $I_1 = 1$ (with $p = 0.8$), all of these networks saturate under the given workload. Therefore, Fig. 6 does not show the results for these cases.

4.2 CC/BH networks

Similar to BH/BH networks, let $N = 2^M$ be the total number of nodes in a CC/BH and $\frac{N}{K_1} = 2^m$ be the number of nodes in each cluster at level 1. There are $K_1 = 2^{M-m}$ clusters at level 1. I_1 interface nodes are selected from each cluster, where $1 \leq I_1 \leq 2^m$. When $I_1 > 1$, any I_1 nodes can be selected as interface nodes because each cluster is a clique. Based on these assumptions, we have the following results:

1. The number of links:

$$L^{(1)} = K_1 L^{(1)}_c = 2^{M-1}(2^m - 1) ,$$

$$L^{(2)} = I_1 L^{(2)}_c = I_1(M-m)2^{M-m-1} ,$$

$$L_{CC/BH} = L^{(1)} + L^{(2)} = 2^{M-1}\left(\frac{I_1(M-m)}{2^m} + 2^m - 1\right) .$$

2. Diameter:

$$Dm_{CC/BH} = \begin{cases} M - m + 2 ; & 1 \leq I_1 < 2^m \\ M - m + 1 ; & I_1 = 2^m \end{cases}$$

3. Average internode distance:

$$AD_{CC/BH} \approx \begin{cases} p + (1-p)(\frac{M-m}{2} + 2) ; & 1 \leq I_1 < 2^m \\ p + (1-p)(\frac{M-m}{2} + 1) ; & I_1 = 2^m \end{cases}$$

4. **Traffic density:** The highest traffic density over the links at level 1 is:

$$TD^{(1)}_{max} = TD^{(1)}_{local} + \max_l (TD^{(1)}_{out,l} + TD^{(1)}_{in,l}) ,$$

where
$$TD^{(1)}_{local} = \frac{pN}{K_1 L^{(1)}_c} \times AD^{(1)} = \frac{p2^m}{2^{m-1}(2^m - 1)} \times 1 = \frac{2p}{2^m - 1} .$$

The $\max_l (TD^{(1)}_{out,l} + TD^{(1)}_{in,l})$ can be computed as follows:

1) $TD^{(1)}_{out,l}$: Since each cluster is a clique, a link connecting an interface node shares either $1/(\frac{2^m}{I_1} - 1)$ of total outgoing traffic (i.e., the traffic generated by a node in the same subcluster) or nothing (i.e., the traffic generated by a node not in the subcluster). Thus,

$$TD^{(1)}_{out,l} = \begin{cases} 1-p ; & 1 \leq I_1 < 2^m \text{ and link } l \text{ connecting two nodes both in the subcluster} \\ 0 ; & 1 \leq I_1 < 2^m \text{ and link } l \text{ connecting any node not in the subcluster} \\ 0 ; & I_1 = 2^m \end{cases}$$

2) $TD^{(1)}_{in,l}$: A link connecting an interface node shares $1/(2^m - 1)$ of total incoming traffic via the interface node. This makes the amount of incoming traffic double on a link connecting two interface nodes. So, we have

$$TD^{(1)}_{in,l} = \begin{cases} 1-p ; & I_1 = 1 \\ \frac{2(1-p)}{I_1} ; & 2 \leq I_1 \leq 2^m \text{ and link } l \text{ connecting two interface nodes} \\ \frac{(1-p)}{I_1} ; & 2 \leq I_1 \leq 2^m \text{ and link } l \text{ connecting any non-interface node} \end{cases}$$

3) Since $TD^{(1)}_{out}$ over the link connecting two interface nodes is 0, we can see that $\max_l (TD^{(1)}_{out,l} +$

$TD^{(1)}_{in,l}$) is not on this kind of links unless $I_1 = 2^m$. As a result, we have

$$\max_l(TD^{(1)}_{out,l} + TD^{(1)}_{in,l}) = \begin{cases} (1-p)(1+\frac{1}{I_1})\,; & 1 \le I_1 < 2^m \\ \frac{1-p}{2^m-1}\,; & I_1 = 2^m \end{cases}$$

From the derivation above, we have

$$TD^{(1)}_{max,CC/BH} = \begin{cases} \frac{2p}{2^m-1} + (1-p)(1+\frac{1}{I_1}); & 1 \le I_1 < 2^m \\ \frac{2p}{2^m-1} + \frac{1-p}{2^m-1}\,; & I_1 = 2^m \end{cases}$$

The traffic density over the links at level 2 is the same as that of BH/BH networks, that is,

$$TD^{(2)}_{max,CC/BH} = \frac{(1-p)N}{I_1} \times \frac{AD^{(2)}}{L^{(2)}_c} = \frac{1-p}{I_1} \times \frac{2^M}{2^{M-m}-1}\,.$$

Finally, we have

$$TD_{max,CC/BH} = \max(TD^{(1)}_{max,CC/BH}, TD^{(2)}_{max,CC/BH})\,.$$

5. The longest average delay: The longest average delay in a CC/BH can be computed as follows:

1) The longest delay at level 1: Each link is modeled as an $M/M/1$ queueing center.

$$\lambda^{(1)}_{link,max} = \lambda \cdot TD^{(1)}_{max,CC/BH}\,,$$

$$W^{(1)}_{max,CC/BH} = \frac{1}{\mu^{(1)} - \lambda \cdot TD^{(1)}_{max,CC/BH}}\,.$$

2) The longest delay at level 2: For $W^{(2)}_{max,CC/BH}$, we also consider two different cases, i.e., $W^{(2)}_{CC/BH,mul}$ and $W^{(2)}_{CC/BH,rep}$.

i) Multiple interface nodes: Each link is modeled as an $M/M/1$ queueing center.

$$\lambda^{(2)}_{link,max} = \lambda \cdot TD^{(2)}_{max,CC/BH}\,,$$

$$W^{(2)}_{CC/BH,mul} = \frac{1}{\mu^{(2)} - \lambda \cdot TD^{(2)}_{max,CC/BH}}\,.$$

ii) Replicated links ($I_1 = 1$): Each group of r-replicated links is modeled as an $M/M/r$ queueing center.

$$u = \frac{\lambda \cdot TD^{(2)}_{max,CC/BH}}{\mu^{(2)}}, \quad \rho = \frac{u}{r},$$

$$C(r,u) = \frac{\frac{u^r}{r!}}{\frac{u^r}{r!} + (1-\rho)\sum_{n=0}^{r-1}\frac{u^n}{n!}},$$

$$W^{(2)}_{CC/BH,rep} = \frac{1}{\mu^{(2)}}\left[\frac{C(r,u)}{r(1-\rho)} + 1\right].$$

3) The longest delay in a network:

$$W_{max,CC/BH} = \begin{cases} \max(W^{(1)}_{max,CC/BH}, W^{(2)}_{CC/BH,mul}); & \text{if choosing multiple interface nodes} \\ \max(W^{(1)}_{max,CC/BH}, W^{(2)}_{CC/BH,rep}); & \text{if replicating level 2 links} \end{cases}$$

$W_{max,CC/BH}$ is shown in Fig. 6 for $p = 0.5$ and $p = 0.8$, respectively. Note that for given parameter values (sizes and workload), the level 2 links of the CC/BH networks saturate when I_1 is too small ($I_1 \leq 2$ if $p = 0.5$ or $I_1 = 1$ if $p = 0.8$). Therefore, Fig. 6 gives the results only for the CC/BH networks which have the appropriate number of interface nodes or replication of links.

4.3 Comparison and analysis

The purpose of comparison here is to see how different structures of hierarchical networks (e.g., the same topology at both levels or different topologies at each level) and different design approaches affect performance and cost-effectiveness. The influence of setting design parameters (i.e., the size of clusters and the number of interface nodes) will be analyzed in detail in the next section.

Figs. 3-9 show Dm, AD, TD_{max}, W_{max}, $L \times AD$, $L \times TD_{max}$, and $L \times W_{max}$, respectively, for some examples of BH/BH, BH, and CC/BH with respect to I_1 (either the number of interface nodes or the number of replication of level 2 links), where I_1 is a power of 2. The size of networks, N, is 1024 and the size of each cluster at level 1 is 16. The values of p are 0.5 and 0.8, respectively. It is assumed that $\lambda = 1$ and $\mu^{(1)} = \mu^{(2)} = 3$. Note that a BH is equivalent to a BH/BH with $I_1 = 16$. From these examples, we can find the following:

1) If I_1 is small, the TD_{max} and/or W_{max} may stay on the links at level 2. (A '*' on the plots in Fig. 5 or Fig. 6 means that the value is from $TD^{(2)}_{max}$ or $W^{(2)}_{max}$.) This is the situation mentioned

in [5] (I_1 is always 1). With the increase in I_1, the TD_{max} and W_{max} of BH/BH networks move to the links at level 1, which implies that the traffic density and message delay over the links at level 2 are reduced faster than that at level 1. For the given CC/BH networks, the TD_{max} is on the level 2 links all the time, because the degree of connection at level 1 is much higher than that at level 2.

2) The lower bound of TD_{max} in BH/BH is 1 which comes from the lower bound of $TD^{(1)}_{max}$, while that in CC/BH can be lower. Similar thing happens to W_{max}. They indicate that it would be necessary to use a topology with higher degree of connection to construct the clusters at level 1, if we want to reduce TD_{max} and W_{max} in BH/BH further.

3) For a BH/BH network (actually also for other values of λ, $\mu^{(i)}$, and p), choosing multiple interface nodes always leads to a shorter delay than replicating its level 2 links, because the delay at level 1 is usually the longest delay in the network and replicating level 2 links cannot reduce the highest traffic at level 1.

4) If I_1 is large enough, a CC/BH network with multiple interface nodes may also achieve a shorter delay than that with replicated level 2 links, because the latter suffers the longer delay at level 1 links. Considering the fact that CC networks have the highest degree of connection among all network topologies but they can still result in the longest delay occurring at level 1 if each cluster has a single interface node, we may conclude that in general, choosing multiple interface nodes is necessary for balancing the traffic over all links in a network and thus is a better design approach than replicating level 2 links.

5) For both BH/BH and CC/BH, a very small value of I_1 is not a good choice or may even be impossible because of saturation. Although it leads to the smallest $L \times AD$ (actually just a little bit smaller), it results in much larger TD_{max} and $L \times TD_{max}$. It may also cause larger W_{max} and $L \times W_{max}$. When p is large, $I_1 = N/K_1$ for BH/BH (= BH) is not good either because of large $L \times AD$, $L \times TD_{max}$, and $L \times W_{max}$.

6) When I_1 is large, CC/BH networks lead to smaller TD_{max}, W_{max}, and $L \times TD_{max}$ (if p is also large) but result in larger $L \times W_{max}$. CC/BH networks also lead to a significant reduction of Dm and AD (about 27% – 42% for Dm and 25% – 40% for AD). However, the $L \times AD$ of CC/BH is relatively large. Thus, there is a trade-off between performance and cost-effectiveness in design of networks. If performance is considered as the main issue for a network, the degree of connection of clusters at level 1 may have to be higher than that at level 2.

5 Design of cost-effective hierarchical networks

In this section, we consider how to design a cost-effective network by choosing appropriate design parameters. This problem has been studied by several researchers [1] [6]. For example, in [6] the authors considered the optimum cluster size and optimum number of levels in the hypercube-based hierarchical networks, using the average internode distance as the performance measure and the number of links as the cost measure. Here we are considering more general cases that are not restricted to some specific topology. Traffic density and average message delay are also used as performance measures so that the analysis is more realistic. The cost-effectiveness measures to be used in the following analysis are thus $L \times AD$, $L \times TD_{max}$, and $L \times W_{max}$.

In order to do the analysis, it is necessary for us to define the problem more precisely. Recall the motivation for hierarchical networks: Exploit locality in communication to reduce the number of links. Thus, if locality exists, for a given non-hierarchical network (*reference network*), we can construct a corresponding hierarchical network in which the number of links at the higher level is reduced so that the hierarchical network could be more cost-effective than its counterpart. The reduction of links at the higher level can be done using a topology with lower degree of connection or selecting fewer interface nodes or both. On the other hand, we can construct a hierarchical network using a topology with higher degree of connection at the lower level while keeping the same topology as that of the reference network at the higher level. Since this method can lead to a significant reduction of TD_{max}, W_{max}, and AD, the hierarchical network can be still more cost-effective than its counterpart in spite of its larger number of links, if the size of clusters at the lower level is small enough. Of course, the second method can be used only for the relatively small networks, otherwise the total number of links may become prohibitively large.

Based on the two methods above, we know that the basic design parameters for a hierarchical network are the topology at each level, the number of interface nodes in each cluster, and the size of clusters. The ideal approach is to consider all of these parameters at the same time, but it is very difficult (even to just consider two of them). We have already seen the impact of different topologies in the last section. Therefore, in the following we only consider how to choose the number of interface nodes and the size of clusters, i.e., we assume that the topologies at both levels are given.

Let L_R, AD_R, TD_R, and W_R be the L, AD, TD_{max}, and W_{max} of the given reference network (i.e., the non-hierarchical network), respectively. Similarly, let L_H, AD_H, TD_H, and W_H represent those of a corresponding two-level hierarchical network. We define the problem as follows:

Problem:

Given N, p, λ, $\mu^{(1)}$, $\mu^{(2)}$, and the topologies of a two-level hierarchical network and its reference network, find the size of clusters at level 1 (N/K_1) and the number of interface nodes (I_1), such that

1) $L_H \times AD_H$, $L_H \times TD_H$, and/or $L_H \times W_H$ are minimized.

2) $\frac{\min(L_H \times AD_H)}{L_R \times AD_R} \leq 1$ and/or $\frac{\min(L_H \times TD_H)}{L_R \times TD_R} \leq 1$ and/or $\frac{\min(L_H \times W_H)}{L_R \times W_R} \leq 1$. □

Some explanations are needed for the above definition:

i) The first condition is to ensure that the resulting values of two parameters are optimal. The second condition is to ensure that the resulting hierarchical network is more cost-effective than its reference network.

ii) p is usually a function of N/K_1, i.e., $p = f(N/K_1)$. Thus, here "given p" means that $f(N/K_1)$ is known.

iii) The reference network has the same topology as that of level 1 clusters or that of level 2 clusters or both. For example, a BH can be a reference network for a CC/BH or a BH/BH.

iv) In general, it is very difficult to minimize all of $L_H \times AD_H$, $L_H \times TD_H$, and $L_H \times W_H$, because they require different values of I_1 and N/K_1. Based on the requirements of an application, either one or two of them are chosen as the main measure, or a trade-off must be made. Similarly, the three inequalities are hard to be satisfied at the same time. When only some of the inequalities can be satisfied, we should ensure that the left side of the other inequalities is less than or equal to a small constant, so that we could still have a cost-effective network.

Solving the problem is not straightforward because of multiple variables in the inequalities and the dependence between them. Also, p depends on N/K_1 and the computation of N/K_1 needs p. Another thing we should mention is that for a hierarchical network, $TD_{max} = \max(TD_{max}^{(1)}, TD_{max}^{(2)})$ requires computation of the second inequality separately for $TD_H = TD_{max}^{(1)}$ and $TD_H = TD_{max}^{(2)}$ (i.e., we have to assume $TD_H = TD_{max}^{(1)}$ or $TD_H = TD_{max}^{(2)}$ each time), and each resulting pair of N/K_1 and I_1 must be consistent with the preassigned TD_H (i.e., the pair must lead to $TD_{max} = TD_H$ in the network). The same thing is for computation of the third inequality because of $W_{max} = \max(W_{max}^{(1)}, W_{max}^{(2)})$. For the first inequality (AD), we need to consider whether $I_1 = N/K_1$ or not. In the following, we propose an algorithm for the problem.

Algorithm:

Step 1: For each of preassigned AD_H, TD_H, and/or W_H, solve

$$\frac{L_H \times AD_H}{L_R \times AD_R} \leq 1 \ \text{and/or} \ \frac{L_H \times TD_H}{L_R \times TD_R} \leq 1 \ \text{and/or} \ \frac{L_H \times W_H}{L_R \times W_R} \leq 1 ,$$

respectively, to find N/K_1 in terms of N, p, λ, $\mu^{(1)}$, $\mu^{(2)}$, and I_1.

Step 2: By assigning each possible value to I_1 and then solving the inequality involving N/K_1 and $p = P(N/K_1)$, find out all possible pairs of N/K_1 and I_1 which are valid for the preassigned AD_H, TD_H, and/or W_H.

Step 3: Compute $L_H \times AD_H$, $L_H \times TD_H$, and/or $L_H \times W_H$ for each valid pair of N/K_1 and I_1. Find the pair which minimizes $L_H \times AD_H$, $L_H \times TD_H$ and/or $L_H \times W_H$. □

For this algorithm, we should note that:

i) It may not find the optimal solution (i.e., the optimal pair of N/K_1 and I_1) if for some reason (e.g., difficulty of solving an inequality) not all possible valid pairs can be found. However, the algorithm will give the best pair from all available pairs, and it guarantees that any solution (optimal or near optimal if it exists) will lead to a hierarchical network which is more cost-effective than or equivalent to its reference network.

ii) The algorithm tries to fix variables one by one, i.e., it solves for a variable at a time. This is because we want to avoid dealing with multiple variables at a single step. It also tries to fix N/K_1 first because the size of clusters at level 1 is usually small, so that values of I_1 could be limited in a small range ($\leq N/K_1$).

We now analyze the algorithm in detail:

Step 1: Find N/K_1:

The difficulty of this step is that the inequalities may not be solved easily. However, by simplifying these inequalities, we may solve them directly or numerically. For example, let us consider the case that the hierarchical network is a BH/BH and its reference network is a BH. We choose $L \times W_H$ as the main cost-effectiveness measure and assume $TD_H = TD^{(2)}_{max}$ to compute TD_H and W_H. From the last section, we know that

$$TD_H \approx \frac{1-p}{I_1} \times \frac{N}{K_1} , \quad W_H = \frac{1}{\mu^{(2)} - \lambda \cdot TD_H} ,$$

$$TD_R \approx 1\ , \quad W_R = \frac{1}{\mu^{(2)} - \lambda \cdot TD_R}\ ,$$

and for BH/BH versus BH, we always have $L_H \le L_R$. So,

$$\frac{L_H \times W_H}{L_R \times W_R} \le \frac{W_H}{W_R} = \frac{\mu^{(2)} - \lambda}{\mu^{(2)} - [\frac{\lambda(1-p)}{I_1} \times \frac{N}{K_1}]}\ .$$

Since it must be $\mu^{(2)} > \lambda^{(2)}_{link,max}$ for a stable system, $\frac{L_H \times W_H}{L_R \times W_R} \le 1$ can be satisfied if $\frac{1-p}{I_1} \times \frac{N}{K_1} \le 1$, that is,

$$\frac{L_H \times W_H}{L_R \times W_R} \le 1 \ \Rightarrow\ \frac{N}{K_1} \le \frac{I_1}{1-p}\ .$$

Then, we need to check whether N/K_1 in this range is also good for $L_H \times TD_H$ and $L_H \times AD_H$. If $\frac{N}{K_1} \le \frac{I_1}{1-p}$, we have

$$\frac{L_H \times TD_H}{L_R \times TD_R} \le \frac{TD_H}{TD_R} = \frac{1-p}{I_1} \times \frac{N}{K_1} \le 1\ .$$

For $L_H \times AD_H$ of BH/BH, we can find that it increases as I_1 increases (see the last section). Since a BH is a special case of BH/BH with the maximum value of I_1 (i.e., N/K_1), $\frac{L_H \times AD_H}{L_R \times AD_R} \le 1$ can hold. Therefore, N/K_1 in this range is acceptable, but it still needs to be validated at the next step to consist with the preassigned TD_H.

Step 2: Determine valid pairs of N/K_1 and I_1:

The number of choices of I_1 is at most N/K_1. Sometimes we want I_1 to be a power of 2, which will produce at most $\log(N/K_1) + 1$ choices. After assigning a possible value to I_1, we can solve the inequality involving p and N/K_1 and obtain a pair of N/K_1 and I_1. Since $p = P(N/K_1)$, we may need to solve the inequality implicitly or numerically. Then, we check whether the resulting pairs are valid, i.e., whether the pairs are consistent with the preassigned AD_H, TD_H, or W_H. For example, if we use $TD^{(2)}_{max}$ to compute N/K_1 at Step 1, then the pairs obtained at Step 2 must lead to $TD_H = TD^{(2)}_{max}$.

Step 3: Find the optimal pair of N/K_1 and I_1:

This step is straightforward: Use all valid pairs to compute all possible $L_H \times AD_H$, $L_H \times TD_H$, and/or $L_H \times W_H$ and then choose the pair leading to the minimum. Besides computing these values, we may see the trends of $L_H \times AD_H$, $L_H \times TD_H$, and $L_H \times W_H$ by direct analyzing related formulas. Here we show how I_1 affects $L \times AD$ and $L \times TD_{max}$.

1) From Eq. (13), $L = K_1 L_c^{(1)} + I_1 L_c^{(2)}$, we can find that $L_c^{(2)}$, which is a function of K_1, will be quite large because K_1 is usually large. Thus, L will increase rapidly as I_1 increases. At the same

time, however, AD (see Eq. (2)) is reduced only a little, i.e., at most $(1-p)AD^{(1)}$ over the whole AD. Therefore, if $L \times AD$ is considered as a main cost-effectiveness measure, small I_1 (1 or 2) is preferred.

2) From Eqs. (3), (4), (5), and (7), we can find that $TD^{(1)}_{out,l}$, $TD^{(1)}_{in,l}$, and $TD^{(2)}_{max}$ are almost inversely proportional to I_1, while $TD^{(1)}_{local}$ is independent of I_1. Two situations will occur:

i) When I_1 is small or the topology of clusters at level 1 has high degree of connection, usually $TD_{max} = TD^{(2)}_{max}$. At that time, increasing I_1 will definitely reduce TD_{max} and $L \times TD_{max}$, which may also lead to the reduction of W_{max} and $L \times W_{max}$. This can be seen from Eqs. (7) and (13). For example, if I_1 is doubled, a half of $TD^{(2)}_{max}$ will be reduced, but L will not be doubled because the item $K_1 L^{(1)}_c$ of L does not increase with I_1. Thus, we can increase I_1 in this situation.

ii) As $TD^{(2)}_{max}$ decreases, $TD^{(1)}_{max}$ may become TD_{max} at some point because of $TD^{(1)}_{local}$ unchanged. After that, increasing I_1 only reduces $TD^{(1)}_{out,l}$ and $TD^{(1)}_{in,l}$. $L \times TD_{max}$ may still decrease for a while but it may eventually increase, if p is large (i.e., $TD^{(1)}_{local}$ is high). In this situation, we can choose the I_1 which yields a value close to the turning point.

6 Concluding remarks

A class of general hierarchical interconnection networks for message-passing architectures has been presented. The proposed hierarchical networks may have any number of interface nodes in each cluster. It has been found that increasing the number of interface nodes in each cluster is better than replicating links, because the former can considerably reduce intracluster traffic density as well as intercluster traffic density and still use the same number of links as the latter. In addition, it enhances the fault tolerance capability of the networks.

The proposed networks with two levels have been evaluated in terms of the performance measures — diameter, average internode distance, traffic density over links, and queueing delay with contention. By examining several typical networks, we have shown that different structures could significantly affect their performance and cost-effectiveness. We have also shown how design of a cost-effective network relies on choosing appropriate design parameters, such as the size of clusters and the number of interface nodes, and how different cost-effectiveness measures require different values of these parameters. An algorithm has been developed for choosing the optimal design parameters.

Hierarchical networks are generally asymmetric, which results in some heavy traffic links that may degrade performance and reliability. Our analysis of traffic distributions shows that for a two-level network, congestion can take place at either level, depending on values of design parameters. Therefore, a good approach to the design of an efficient hierarchical network must ensure the balance of traffic over all levels of the network.

References

[1] S. G. Abraham and E. S. Davidson, "A communication model for optimizing hierarchical multiprocessor systems," In *1986 Int'l Conf. Parallel Processing*, pp. 467–474, 1986.

[2] A. O. Allen, *Probability, Statistics, and Queueing Theory with Computer Science Applications*, New York: Academic Press, 1978.

[3] D. Carlson, "The mesh with a global mesh: A flexible, high-speed organization for parallel computation," In *1st Int'l Conf. Supercomput. Syst.*, pp. 618–627, 1985.

[4] S. P. Dandamudi, "A performance comparison of routing algorithms for hierarchical hypercube multicomputer networks," In *1990 Int'l Conf. Parallel Processing*, pp. I-281–I-285, Aug. 1990.

[5] S. P. Dandamudi and D. L. Eager, "Hierarchical interconnection networks for multicomputer systems," *IEEE Trans. Computers*, vol. C-39, pp. 786–797, June 1990.

[6] S. P. Dandamudi and D. L. Eager, "On hypercube-based hierarchical interconnection network design," *J. Parallel and Distributed Computing*, vol. 12, pp. 283–289, 1991.

[7] T. Y. Feng, "A survey of interconnection networks," *IEEE Computer Mag.*, vol. 14, pp. 12–27, 1981.

[8] K. Ghose and K. R. Desai, "The design and evaluation of the hierarchical cubic network," In *1990 Int'l Conf. Parallel Processing*, pp. I-355–I-362, Aug. 1990.

[9] W. D. Hillis, *The Connection Machine*, Cambridge, MA: MIT Press, 1985.

[10] K. Hwang and J. Ghosh, "Hypernet architectures for parallel processing," In *1987 Int'l Conf. Parallel Processing*, pp. 810–819, Aug. 1987.

[11] L. Kleinrock, *Queueing Systems:*, vol. 1, New York: Wiley and Sons, 1975.

[12] D. J. Kuck, E. D. Davidson, D. H. Lawrie, and A. H. Sameh, "Parallel supercomputing today and the cedar approach," *Science*, vol. 231, pp. 967–974, Feb. 1986.

[13] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik, *Computer System Performance*, Englewood Cliffs, NJ: Prentice-Hall, 1984.

[14] S. A. Mabbs and K. E. Forward, "Optimising the communication architecture of a hierarchical parallel processor," In *1990 Int'l Conf. Parallel Processing*, pp. I-516–I-520, Aug. 1990.

[15] K. Padmanabhan, "Efficient architectures for data access in a shared memory hierarchy," *J. Parallel and Distributed Computing*, vol. 11, pp. 314–327, 1991.

[16] D. A. Reed and R. M. Fujimoto, *Multicomputer Networks: Message-Based parallel Processing*, Cambridge, MA: MIT Press, 1987.

[17] D. A. Reed and D. C. Grunwald, "The performance of multicomputer interconnection networks," *IEEE Computer*, vol. 20, pp. 63–73, June 1987.

[18] H. J. Siegel, *Interconnection Networks for Large-Scale Parallel Processing*, 2nd edition, New York: McGraw-Hill, 1990.

[19] R. J. Swan, S. H. Fuller, and D. Siewiorek, "Cm* — a modular, multi-microprocessor," In *The National Computer Conf.*, pp. 637–644, 1977.

[20] S. W. Wu and M. T. Liu, "A cluster structure as an interconnection network for large multicomputer systems," *IEEE Trans. Computers*, vol. C-30, pp. 254–264, 1981.

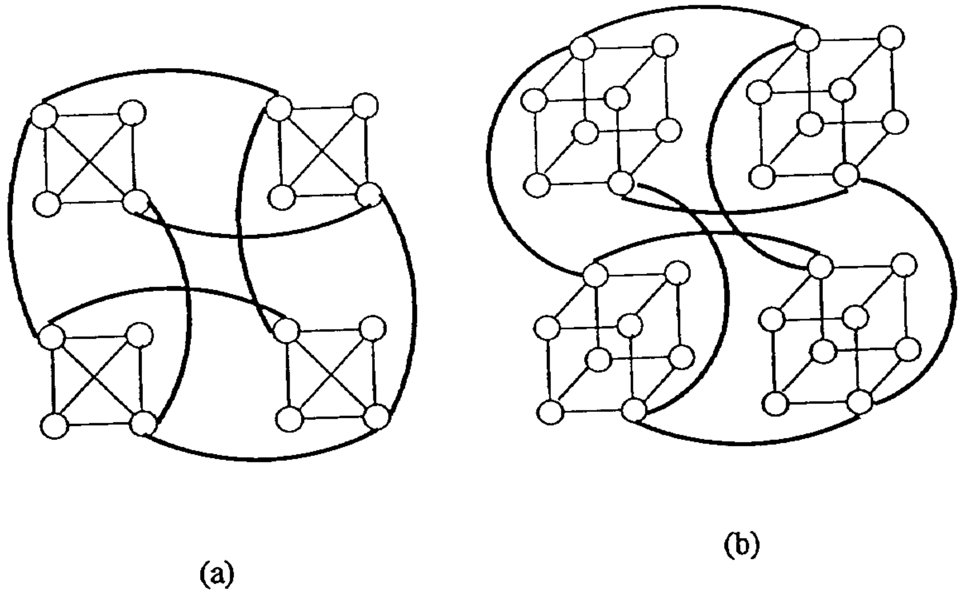


Fig. 1. (a) A two-level CC/BH network with $N = 16$, $K_1 = 4$, $I_1 = 2$, and $K_2 = 1$. (b) A two-level BH/BH network with $N = 32$, $K_1 = 4$, $I_1 = 2$, and $K_2 = 1$. (The links at level 2 are darkened.)

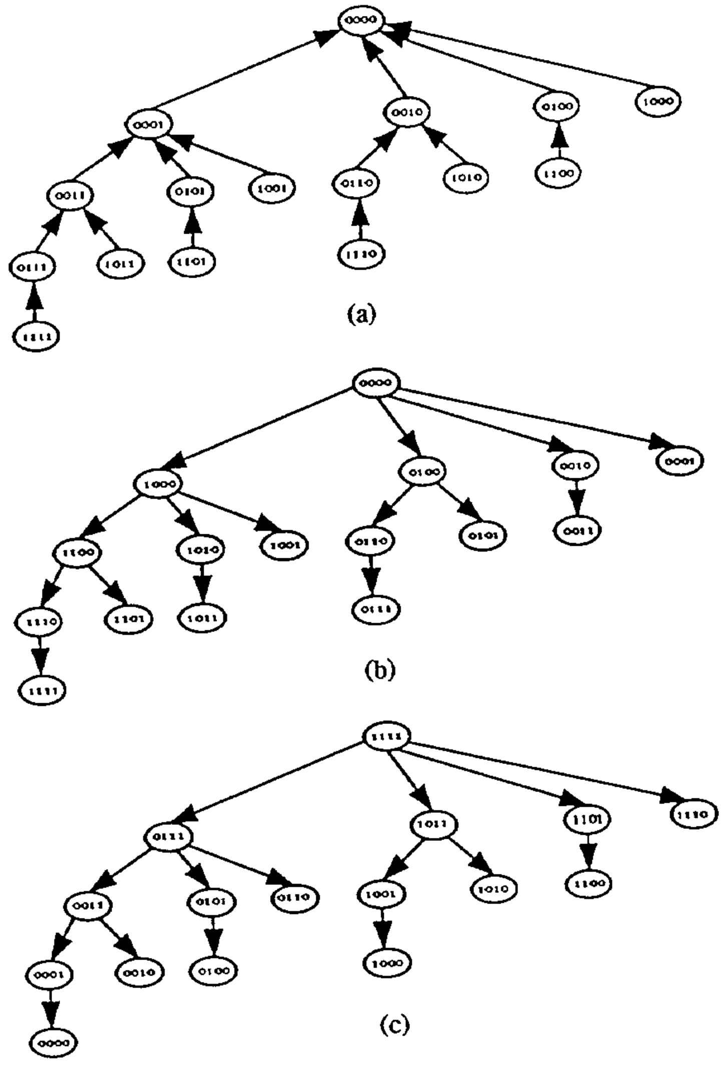


Fig. 2. (a) Outgoing messages via interface node (0000) in a subcluster (BH) with 16 nodes. (b) & (c) Incoming messages via interface nodes (0000) and (1111), respectively, in a cluster (BH) with 16 nodes.

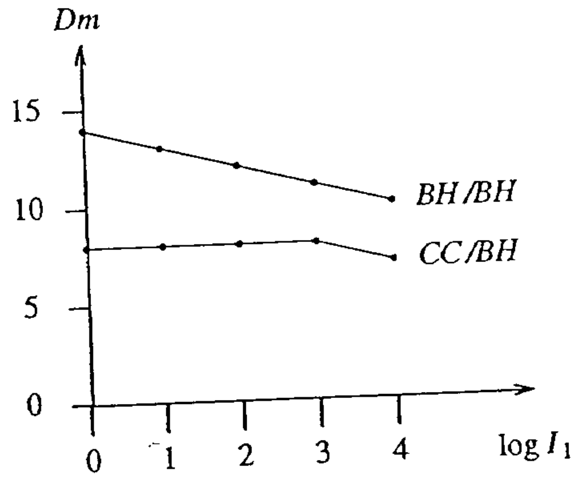


Fig. 3. Diameter (Dm) versus I_1, with N=1024 and N/K_1=16.

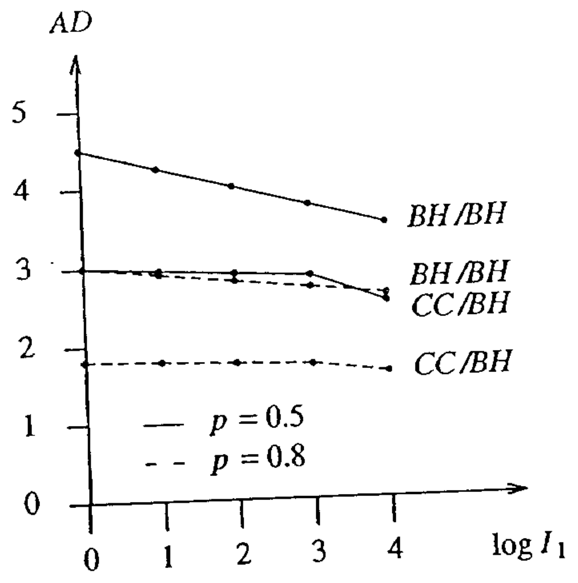
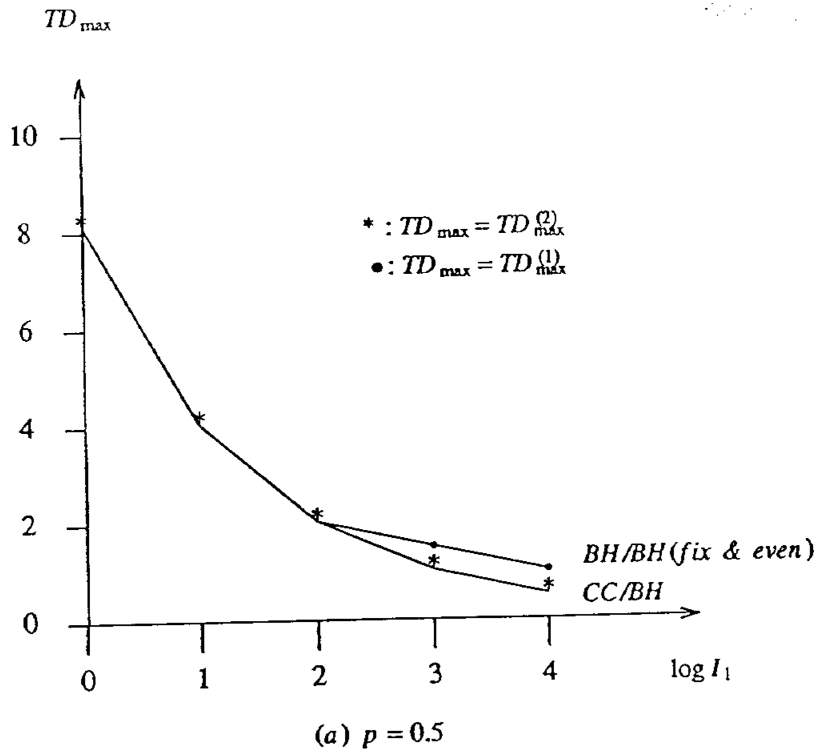


Fig. 4. Average internode distance (AD) versus I_1 under $p = 0.5$ and $p = 0.8$, respectively, with N=1024 and N/K_1=16.



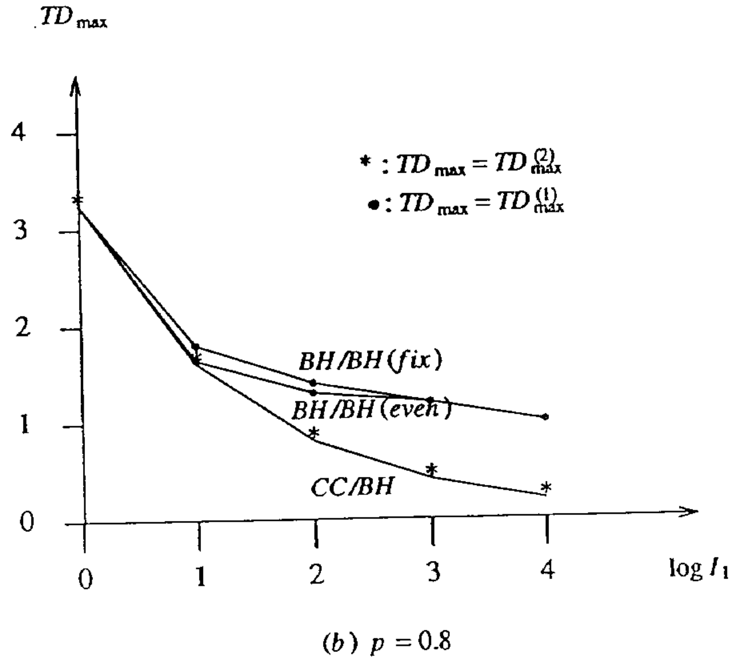
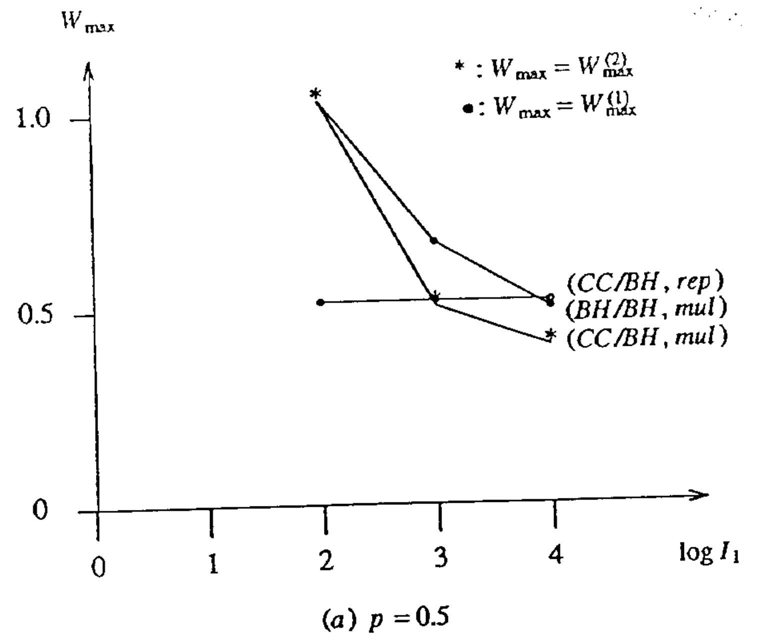


Fig. 5. The highest traffic density ($TD_{\max}$) versus I_1 under (a) $p = 0.5$ and (b) $p = 0.8$, respectively, with N=1024 and N/K_1=16.



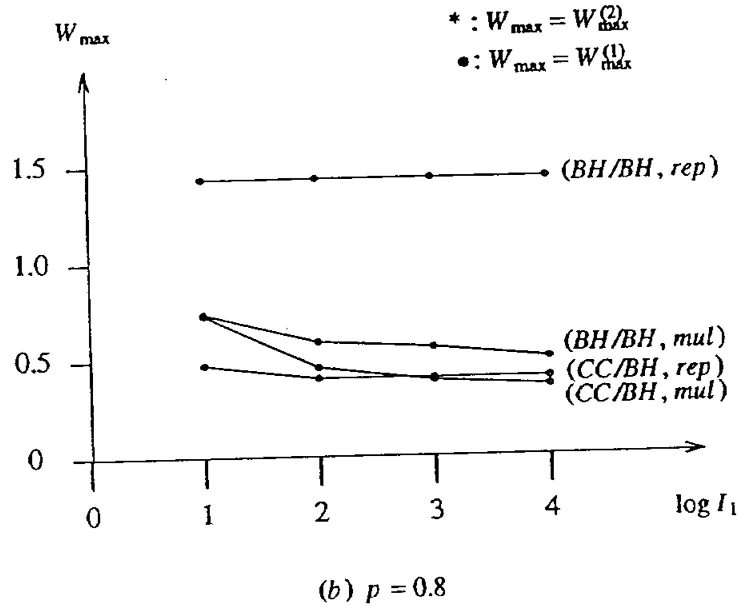


Fig. 6. The longest queueing delay ($W_{\max}$) versus I_1 for (a) $p = 0.5$ and (b) $p = 0.8$, respectively, with $N = 1024$, $N/K_1 = 16$, $\lambda = 1$, and $\mu^{(1)} = \mu^{(2)} = 3$.

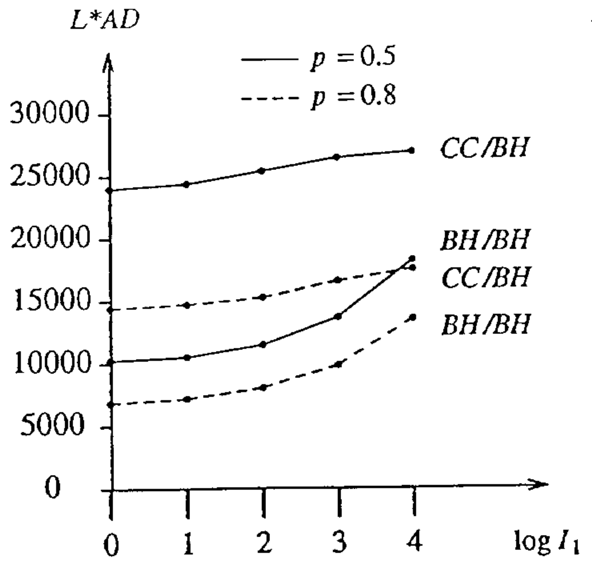


Fig. 7. $L*AD$ versus I_1 under $p = 0.5$ and $p = 0.8$, respectively, with N=1024 and N/K_1=16.

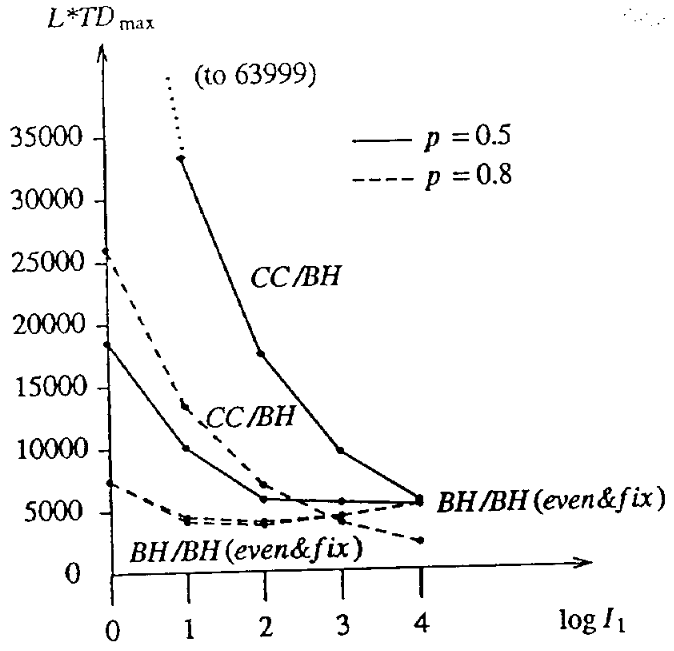


Fig. 8. $L^*TD_{\max}$ versus I_1 under $p = 0.5$ and $p = 0.8$, respectively, with N=1024 and N/K_1=16.

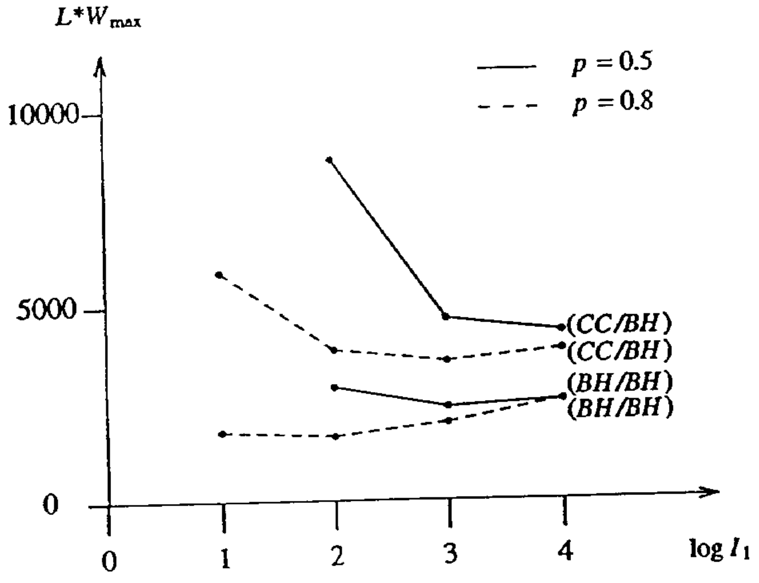


Fig. 9. L^*W_{max} versus I_1 under $p = 0.5$ and $p = 0.8$, respectively, with N=1024, N/K_1=16, λ=1, and $\mu^{(1)} = \mu^{(2)} = 3$.