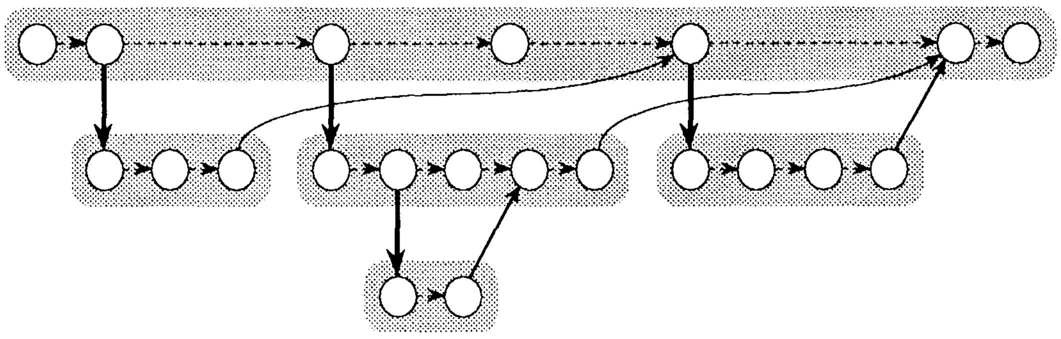
MIT/LCS/TR-546

PROCEEDINGS OF THE 1992 MIT STUDENT WORKSHOP ON VLSI AND PARALLEL SYSTEMS

Edited by
Charles E. Leiserson

August 1992

Proceedings of the 1992 MIT Student Workshop on VLSI and Parallel Systems



Edited by Charles E. Leiserson

July 21, 1992

The papers in this volume were submitted to the *1992 MIT Student Workshop on VLSI and Parallel Systems.* The workshop was organized by the VLSI and Parallel Systems Group at MIT to promote an interchange of ideas among the various research activities at MIT in VLSI and parallel systems. It was held on July 21, 1992 at the MIT Endicott House in Dedham, Massachusetts. Of the 54 papers in this proceedings, 16 were chosen for presentation at the workshop. These papers are marked with an asterisk.

Proceedings of the 1992 MIT Student Workshop on VLSI and Parallel Systems

MIT Endicott House
July 21, 1992

Contents

Micron-Scale Display Technology

Phillip Alvelda[1]
NE43-810
alvelda@ai.mit.edu

I. Introduction

The primary objective of this current research project, is to design, fabricate, and test a micron-scale liquid crystal-based virtual display. The intended application for this device is a small hand-held or eyeglass-mounted display for systems such as a portable "pen-mounted oscilloscope" (tricorder?) or heads-up type eyeglasses. Note that the design requirements for this system are quite different from those of similar spatial light modulators fabricated for optical computing applications where focal lengths are typically much longer. As such, several additional psychophysical effects were considered when optimizing the display design for visual inspection.

In spite of several complications due to the addition of a liquid crystal surface, the low-mass and static power dissipation characteristics of CMOS processes are shown to be quite usable as an active electronic back-plane diplay driver. The final version of this prototype single-chip display is expected to have a resolution of approximately 1000 x 1000 pixels on a 1 centimeter die.

Several preliminary designs for VLSI display sub-circuits will be presented, and the architecture of the first-generation prototype awaiting fabrication through MOSIS will be discussed.

II. Background

A simple display can be fabricated by depositing an appropriate liquid crystal on the top of an active CMOS backplane, and then sealing it with a cover glass whose inside surface has been coated with a transparent conductor such as indium-tin-oxide. In equilibrium, the liquid crystal sandwiched between the CMOS die and the coverglass assumes a particular orientational order which when illuminated, is opaque. When a voltage is applied by the CMOS circuitry to a pixel pad underneath the liquid crystal, the electric field re-aligns the LC molecules in an orientation which is transparent. Since the Aluminum METAL2 used in the pixel pads is a good diffuse reflector, simply shining a light on an array of appropriately addressed pixels produces an image.

Unfortunately since each pixel can range in size from 10 to 50 microns across, it produces a very tiny image that is difficult to see without magnification. A quality microscope can provide a nice image, but is rather inconvenient to lug about attached to one's eyeglasses. On-the-other-hand, a simpler, more light-weight optical system introduces considerable distortion.

Several companies have begun compensating for this distortion in "virtual-reality" displays by pre-processing the image data in real-time to pre-warp the image data before it is "distorted" by a smaller optical system. Obviously, a real-time image processing computer is not yet typically light-weight and would severely limit the portability of such a system. The solution demonstrated in this project, is a VLSI pixel array which is designed to approximate the inverse of the optical

[1]This work is supervised by Tom Knight for DARPA.

distortion function of a typical small lens system without additional processing simply by appropriate layout and scaling of the pixel elements.

III. The Pixel Array

For the preliminary device design, 324 pixels were positioned in a precise 18 x 18 array tailored to exactly account for the distortions introduced in the LEEP optical system (See Figure). Ray tracing experiments through the actual optics resulted in a Distortion function which was monotonically increasing with radius from the optical axis. Therefore, there exists an inverse function which can negate this nonlinear distortion. A third-order polynomial fit to the distortion function data provided an approximation to the inverse function:

$$D^{-1}(r_i) = r_i + \kappa r_i^3, \quad \kappa = 0.32$$

This radially symmetric function would then be used to position and scale the individual pixels of the display. Since the array is very regular, albeit of peculiar and specific form, a silicon compiler was used to implement a scalable architecture, where multi-size chips can be auto "instanced and routed" with the late-binding specification of only a few parameters such as ARRAY_WIDTH, PACKING_DENSITY, etc. And while the L language from mentor has some of these features, it does not have any built-in trigonometric functions or floating-point math functions (i.e. exponentials, logs, sin, cos, etc...). This Auto-Chip generator will also be presented.

Other issues which will also be discussed :

- Polygon generation for transformed pixel arrays
- Pixel Jitter and Anti-Aliasing
- Light-sensitivity and shielding
- Failure modes and redundancy
- Device Interface

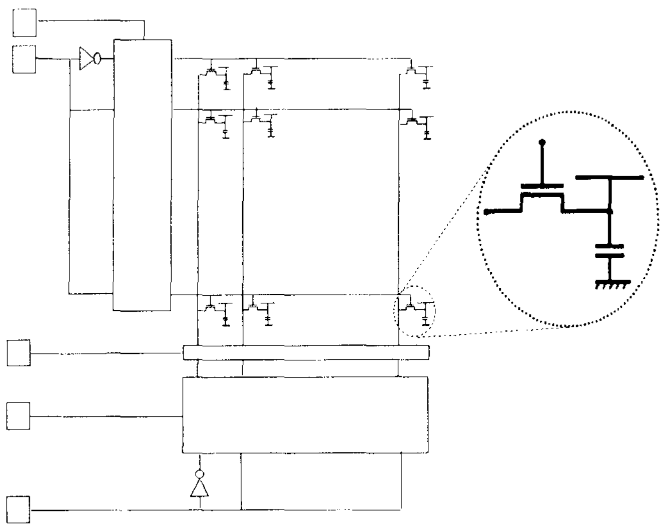


Figure 1.0 Array Schematic.

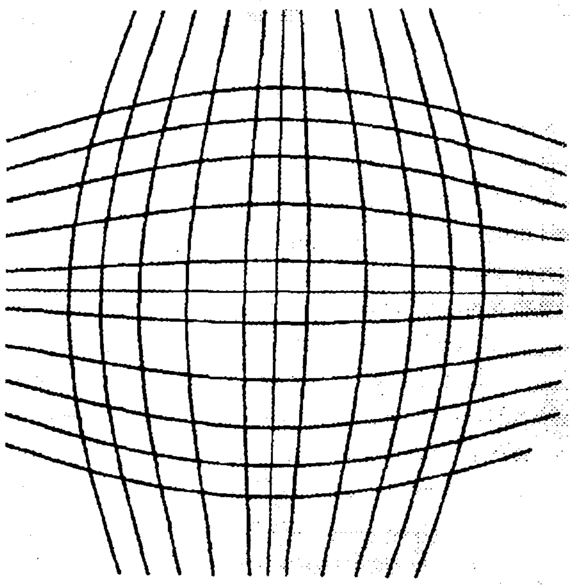


Figure 2.0 LEEP Distorion Function.

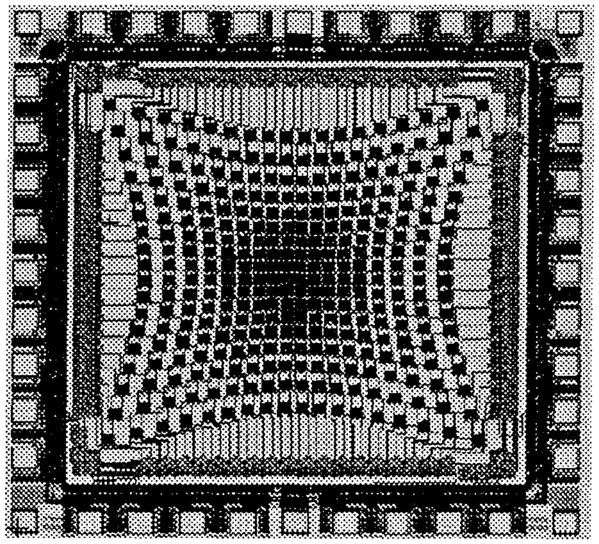


Figure 3.0 Preliminary Chip Layout

References:

[1] Burns, D., **Microcircuit Analysis Techniques using Field effect Liquid Crystals**, Proc. SPIE Volume 1256, 1987.

[2] Channin, D., **Liquid Crystal Techniques for Observing Integrated Circuit Operation**, IEEE Transactions on Electron Devices, October, 1974.

Optimization of Loops for Dynamic Dataflow Machines

Boon Seong Ang [1]
NE43-205, hahaha@abp.lcs.mit.edu

Using the Id language, Monsoon has successfully demonstrated the concept of dynamic dataflow execution. Programs that are difficult to parallelize explicitly show considerable parallelism when compiled [3] and run on Monsoon. However, run time statistics show that considerable overhead is incurred compared to Von Neuman code executing on uniprocessors. Loops, in particular, are expensive.

We pursued several compilation techniques for loops that produce Monsoon code using fewer dynamic instructions than previous methods. We first focused on producing good code for sequential loops, which usually form the innermost loops and have the biggest impact on run time. We found two new compilation schema with significantly lower overhead. We also implemented strip mining of k-bounded loops, which weakens data-dependence and thus increases parallelism; and the lifting of allocation of reusable storage, including frames and certain heap objects, out of loops. We are currently collecting run time statistics and comparing these with C/Fortran code running on MIPS R3000.

In a *sequential loop*, only one iteration of the loop executes at any one time. From an implementation point of view, the loop requires only one frame, and passing values from one iteration to the next need not occur across the network. We thus expect the code for sequential loop to be much more efficient than *k-bounded*[2] loops.

Implementing sequential loops is tricky. As the *tags* of tokens from different iterations destined for the same instruction are the *same*, tokens from different iterations could be confused. The compiler must ensure that this *cannot* happen by adding artificial data-dependence to the dataflow

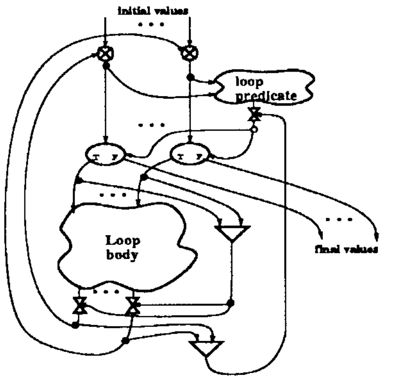


Figure 1: Culler's Sequential Loop Schema

graphs. Culler[2] gave one such schema that uses two barriers. This is shown in Figure 1.

This schema, however, is very expensive. Consider this simple loop that computes the sumation of 1 to n:

```
{for i <- 1 to n do
   next s = s + i;
 finally s};
```

Under Culler's schema, each iteration takes 22 tokens [3].

With the new schema shown in Figure 2, we can ensure no confusion occurs. *Self gating*, as shown in the figure, is unnecessary for a nextified variable[4] that is *strict in itself*[5] , a situation encountered often in real code. This can be determined by the compiler. Under the new schema, our summation example takes only 12 tokens per iteration.

The implementation of sequential loops on Monsoon can be further improved by observing that mechanisms that support fine grain parallelism

[1] Supervisor: Arvind. Funding is provided in part by the Advanced Research Projects Agency of the Department of Defence under Office of Naval Research contract N00014-89-J-1988

[2] See [2] for a definition of sequential and k-bounded loop.

[3] Each token takes one cycle.

[4] A nextified variable is a variable that is updated each iteration.

[5] A nexified variable is strict in itself if computation of its *next* value always requires its current value.

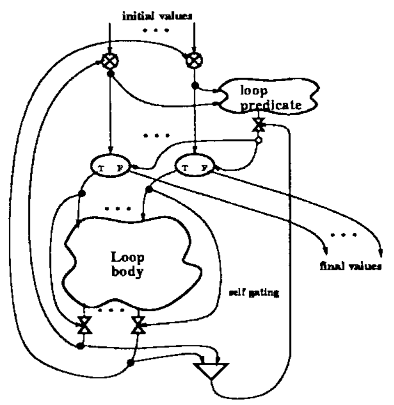


Figure 2: New Sequential Loop Schema

and synchronization, such as switching at loop iteration boundaries, and synchronization with nextified variable are not needed in sequential loops. Instead, we synchronize at the iteration level. The values of nextified variable are passed via memory locations instead of tokens. We call this *Frame Based Nexified Variable* optimization. With this schema, the summation example takes only 7 tokens per iteration.

Strip mining breaks the data dependence between iterations of a FOR loop when all the updates to nexified variables are of some simple form, such as incrementing or decrementing a nexified variable by a constant amount each iteration. This can be extended to include operations that are both commutative and associative, and whose resulting values are not used within the loop but merely returned from it.[6]

Strip mining of a k-bounded FOR loop converts it into a pair of doubly nestled loops. The outer loop executes k iterations in parallel while the inner loop, which does the original work, is sequential.

Strip mining has two big advantages:

- It allows us to use the sequential loop schema, which is much cheaper then the k-bounded loop schema, while offering parallel execution of the loop.
- It breaks the coupling between the k frames of a loop, allowing each to proceed independently at full speed. Previously, if one of the k frames resided on a PE that was busy executing other work, the entire loop stalled. With strip mining, once the k sequential loops are started, a frame on a busy PE will not affect the execution on the other $(k-1)$ frames.

[6] Overflow, underflow and precision in the case of floating point operations may cause some problems.

Allocation of reusable storage can be lifted out of loops. Certain resources allocated within a loop are only needed over one loop iteration, for instance, frames for procedure invocation and loops nestled within the outer loop. Instead of allocating and deallocating once per iteration, we can do so just once for each *frame* of the loop. Once frame allocation is lifted out, some other initialization code, such as storage of *loop constants*[7] can also be lifted out.

The same idea can be applied to heap objects with life-times of a constant number of loop iterations. Such examples, which abound in Id code doing iterative numerical computation, can easily be detected at compile time. As before, we can recycle those heap objects once they become "garbage".

We expect the optimizations outlined here to produce efficient Monsoon code from loops written in Id. So far, results are encouraging. [1] contains a detail report of this work.

References

[1] B. S. Ang. *Optimization of Loops for Dynamic Dataflow Machines.* MS thesis, MIT, Cambridge MA, Under preparation.

[2] D. E. Culler. *Managing Parallelism and Resources in Scientific Datflow Programs.* PhD thesis, MIT, Cambridge MA, Jun 1989.

[3] J. E. Hicks. *Id Compiler Back End for ETS and Monsoon.* CSG Memo 310, MIT Lab. for Comp. Sci., Cambridge MA, Jun 1990.

[7] A loop-constant is a variable that is computed outside the loop, and remains constant over the entire loop.

Virtual Memory for Data-Parallel Computing

Lars E. Bader
leb@theory.lcs.mit.edu

Thomas H. Cormen
thc@theory.lcs.mit.edu

MIT Laboratory for Computer Science
Cambridge, MA 02139

Some applications that are well-suited to data-parallel computing, such as large finite-element problems, must sometimes process more data than will fit in the RAM of even the largest parallel computer. Consequently, the data typically resides on a disk array and is brought into RAM as needed.

To support such applications, we have designed a virtual-memory system for a data-parallel machine. Our system manages the disk I/O efficiently and removes the burden of planning disk I/O from the application programmer. In this report, we focus on the issues of data layout, page-replacement policy, and permutation routing.

Our virtual-memory model lays out vectors across processors and across a set of disks organized into blocks of records. The parallel machine has P processors, D disks, and B records per disk block. A track consists of a set of blocks at the same location on each disk, and it contains BD records. Vectors are organized by track on disk and by track image in RAM. Each parallel I/O operation transfers B records between each disk and RAM, with BD/P records transferred per processor.

Significant data-parallel operations fall into three general categories—elementwise operations, scans, and permutations—but the performance of only scans is affected by how vectors are laid out. As shown in Figure 1, vectors can be laid out in row-major, column-major, or blocked fashions.

Our system uses blocked layout because it is a good compromise in limiting the number of scans across all the processors and the number of disk I/Os. Letting N be the number of records in a vector, S be the time required to scan once across the P physical processors, A be the time to perform an arithmetic operation, and IO be the disk I/O time, the scan times for the different vector-layout methods are the following:

- With row-major layout,

$$T_{\text{row}} = \frac{N}{P} S + 2\left(\frac{N}{P} - 1\right) A + 2\frac{N}{BD} IO .$$

- With column-major layout,

$$T_{\text{col}} = S + \left(2\frac{N}{P} - 2\right) A + 3\frac{N}{BD} IO .$$

- With blocked layout,

$$T_{\text{blocked}} = \frac{N}{BD} S + \left(2\frac{N}{P} + \frac{N}{BD} - 2\right) A + 2\frac{N}{BD} IO .$$

Scans for blocked layout require only one physical scan per disk track and hence are faster than for row-major layout. Specifically, $T_{\text{blocked}} \le T_{\text{row}}$ when $A \le S$ and $N \ge BD \ge 2P$, which holds for vectors larger than a track since $BD \gg P$ and $S \gg A$. With blocked layout, scans read each track only once, rather than twice, as is the case with column-major layout. Hence, scans are faster with blocked layout. Specifically, $T_{\text{blocked}} \le T_{\text{col}}$ when $S \le IO - A$ and $N \ge BD$, which holds for vectors larger than a track because $IO \gg S \gg A$.

The paging system manages vectors based on their size, with tracks treated as pages. When vectors fit in RAM, we want the system to be roughly as fast as a non-VM system. If we were to treat all tracks equally under LRU replacement, then accesses of large vectors could result in all small vectors being paged out of RAM. In fact, an LRU scheme may be pointless for very large vectors, such as those that exceed the RAM size. Observe that the first track accessed for such a vector may be paged out by the time the last track is accessed, and thus accessing each track yields a page fault. Our system partitions RAM into two halves, with separate LRU replacement. One half holds only tracks of large vectors (more than one track), and the other half holds only tracks containing small vectors (at most one track). With this scheme, accesses to large and small vectors do not interfere with each other's paging behavior. The paging system may be made yet more efficient by distinguishing between large vectors with size less than the large-vector RAM, for which LRU replacement may reduce disk references, and those that are larger than the large-vector RAM, for which LRU replacement does not improve paging performance.

Our paging system is optimal when vectors are accessed in a stack-like fashion. Performance for large

This research was supervised by Professor Charles E. Leiserson and was supported in part by the Defense Advanced Research Projects Agency under Grant N00014-91-J-1698.

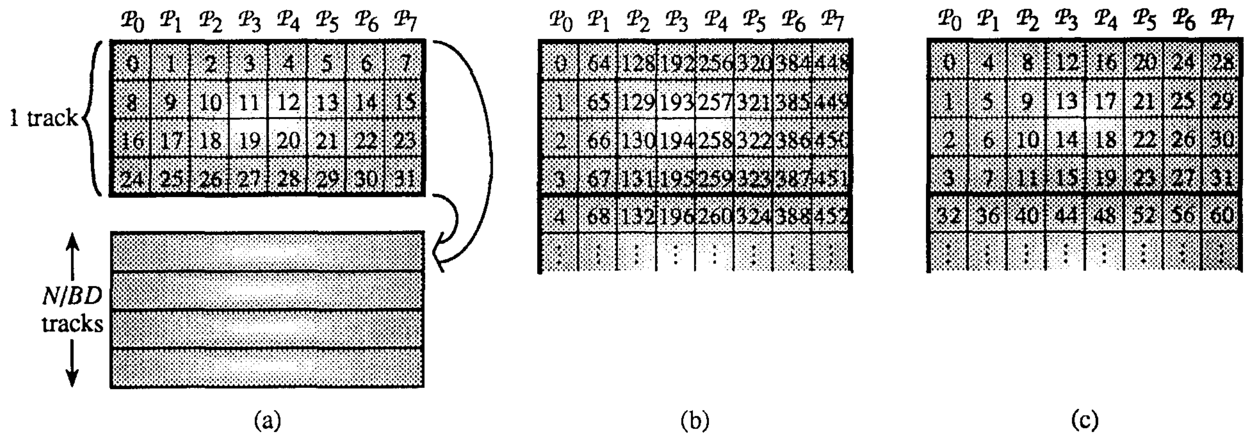


Figure 1: Three different ways to lay out vectors on a disk array, indicating the mapping of vector elements to processors $\mathcal{P}_0, \mathcal{P}_1, \ldots, \mathcal{P}_7$. Each 4×8 rectangle delimits a track. (a) Row-major layout. If a vector spans more than one track, it occupies a contiguous set of N/BD tracks, as shown. (b) Column-major layout, shown for 64 elements per processor. (c) Blocked layout. It can be viewed as a transpose within each track of the row-major layout, so that the ordering is column-major within each track.

vectors and small stack-based vectors is competitive. For those data-parallel languages that tend to have stack-like accesses, good performance results. In addition, our method is competitive if locations of small vectors are fixed on pages. A system with the ability to relocate small vectors, however, may outperform our paging method.

Our system supports permutation operations, providing special routines for some types of permutations that can be performed faster than general permutations. RAM is cleared to provide work space for large permutations, which are expensive. For general permutations, we sort N target addresses using external radix sort, which uses $\Theta\left(\frac{N}{BD}\frac{\lg N}{\lg(M/BD)}\right)$ parallel I/Os. The optimal sorting and permutation bound is $\Theta\left(\frac{N}{BD}\frac{\lg(N/B)}{\lg(M/B)}\right)$ parallel I/Os, achieved by the more complicated algorithms of Vitter and Shriver and also Nodine and Vitter. Some classes of permutations that can be done faster than general ones are monotonic routes, mesh communication, bit-permute/complement (BPC) permutations (which include matrix-transpose, bit-reversal, vector-reversal, hypercube, and matrix-reblocking permutations), and bit-matrix-multiply/complement (BMMC) permutations (which include Gray-code permutations). Monotonic routes require only one read per source track and one write per destination track. Storing the mesh so that each track holds a submesh, the elements on each track in a mesh permutation are destined for either the same track or one other, requiring $\Theta(N/BD)$ parallel I/O's. We have implemented Cormen's algorithm [1] for BPC permutations, and we have found that it provides a significant speedup over the external radix-sort method. Mesh and BMMC permutations are not yet implemented specially.

There are three ways to incorporate special permutations into the virtual-memory system. First, we can treat them as general permutations. Second, we can provide linguistic constructs for them, avoiding the overhead of generating target addresses and enabling direct calls to special code. For example, the source language we use includes a pack instruction, which performs a type of monotonic route. Third, we can detect them at run time and call special code. Currently, we detect BPC permutations at run time by forming a candidate bit permutation, using $\lceil(\lg(N/B)+1)/D\rceil$ parallel I/Os, and then verifying that it describes the given permutation, using at most N/BD I/Os.

Our system uses the compiler for the source language NESL and the interpreter for the stack-based intermediate language VCODE developed by Blelloch et al. Our virtual-memory system is a complete implementation of CVL, which is the machine interface for VCODE. Rather than implement it on a real data-parallel machine, we have chosen for convenience to simulate it on workstations, which allows us to generate machine-operation statistics that might be difficult to determine on a real machine. The simulator is written in about 7500 lines of C.

References

[1] Thomas H. Cormen. Fast permuting in disk arrays. In *Proceedings of the 1992 Brown/MIT Conference on Advanced Research in VLSI and Parallel Systems*, pages 58–76, 1992. Conference version is an extended abstract; full paper to appear in *Journal of Parallel and Distributed Computing*.

The Optimal Synthesis of VLSI Array Architectures From Algorithmic Descriptions

Donald G. Baltus[1]

The capabilities of VLSI technology now allow many algorithms to be realized monolithically using application-specific array architectures. While early work in this area involved finding and describing architectures to solve specific problems, more recent research has been directed towards the development of systematic methodologies for synthesizing array architectures from high-level algorithmic descriptions[2]. This paper describes the array synthesis system DESCARTES[3] which is being developed by the author at MIT. As compared with other work in this area, DESCARTES is applicable to a wider class of algorithms and is the only system of its kind that incorporates systematic and exhaustive architectural exploration into the synthesis process.

The input to DESCARTES is essentially a set of affine recurrence equations. Many important algorithms including those in the areas of digital signal processing, graph theory, and matrix computation can be described using such recurrences. The target implementation space is a 1- or 2-dimensional mesh of application-specific processing elements. Neither the processors nor their interconnections need be uniform throughout the array. For a given input description, DESCARTES generates a set of possible RTL-level implementations. The synthesis process includes an architectural exploration phase which guarantees that all legal designs which optimize a user-specified temporal objective function are generated. A simple matrix-vector product example will be used throughout the paper and is described below. The associated dependency graph (DG) is shown in Figure 1.

$$1 \leq i \leq 3, \qquad x(i) = \sum_{j=1}^{3} a(i,j) * b(j)$$

Since algorithms are mapped onto a class of architectures with well defined structural and interconnection characteristics, important implementation-level costs can be accurately predicted at the architectural level. More specifically, the structured nature of the target architecture allows spatial relationships between computations to be seen at the architectural level. This information in turn allows communication costs to be accurately modeled and incorporated into the design exploration process.

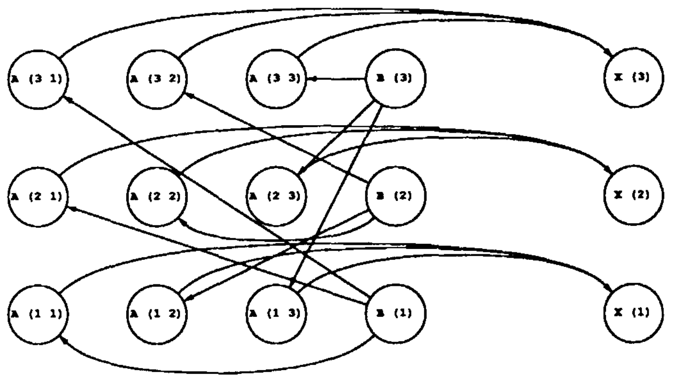


Figure 1: The Data Dependency Graph

While the ability to accurately predict implementation-level characteristics is essential for effective performance-directed synthesis, the availability of these predictors complicates the tasks of architectural exploration and synthesis. If communication costs are ignored, the problems of scheduling and of allocation can be solved in isolation. If communication delay is modeled as a function of relative spatial locations, however, the scheduling and allocation tasks become more closely linked and must be solved together.

The problem of combined scheduling and allocation is approached by casting the problem of architectural exploration as the problem of exploring different embeddings of the nodes of the data dependency graph into a space-time lattice. For each node, the location in one dimension designates the time at which the computation takes place while the location in the other dimensions designates the spatial location where the computation will be performed[4].

Clearly not all embeddings in the space-time lattice are valid. More specifically, the nodes must be embedded such that causality and communication delay constraints are satisfied, such that processor functionality constraints are satisfied, and finally such that bandwidth and I/O constraints are met.

Architectural exploration is thus reduced to the problem of exploring different embeddings that satisfy the constraints outlined above. Each embedding represents a different architecture. If each node of the dependency graph can be embedded independently, however, the problem

[1]MIT Rm. 36-881 (don@rle-vlsi.mit.edu). This research is supervised by Prof. Jonathan Allen and is funded by Analog Devices and IBM.

[2]A survey of different techniques can be found in [1] while information on more recent work can be found in [2] and [3].

[3]**DESCARTES** stands for Design Environment for Systematic Cross-Level **AR**chi**T**ectural **E**xploration and **S**ynthesis.

[4]While the space-time lattice is typically 2- or 3-dimensional, techniques have also been developed for mapping higher-dimensional lattices onto 1- or 2-dimensional target array architectures.

of architectural exploration becomes computationally intractable. The number of possible embeddings is exponentially related to the problem size and there is no guarantee that any regularity that existed in the original algorithm specification will be preserved as that algorithm is mapped into the implementation space.

This problem is solved by grouping *similar* nodes and moving the members of each such group together during architectural exploration. A group of similar nodes corresponds to a subset of the dependency hierarchy whose elements are identical in all aspects except index location. Since similar nodes necessarily correspond to references of the same array variable, a set of spatial relationships is inherently defined among the members of each group.

Architectural exploration involves mapping each such group into the space-time lattice in a way that ensures that the spatial relationships established within each group are maintained. Affine transformations are used to perform this mapping[5]. The use of these transforms provides a constrained moved set for the architectural exploration, ensures that desired spatial relationships are preserved, and finally guarantees that the schedule associated with each group will be an affine function of spatial location. The node groupings for the example and one set of affine mappings is shown in Figure 2.

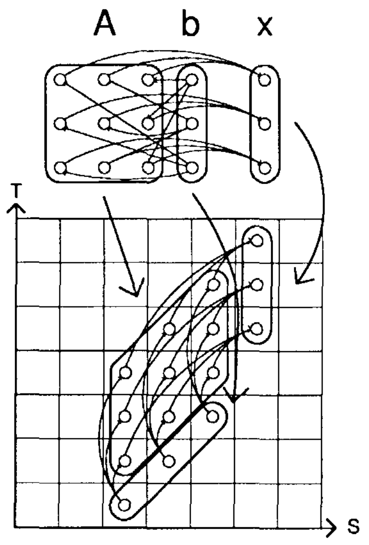


Figure 2: Similar Node Groupings and One Affine Embedding in Space-Time

Efficient architectural exploration is achieved by separating the scheduling and allocation aspects of the search. The different characteristics of scheduling and allocation make a partitioned search much more efficient than a naive exploration of different embeddings. While the effect is the same as a combined exploration, the search is structured in a way that allows the scheduling and allocation phases to be performed largely independently.

[5]An affine transform is a vector function of the form $f(x) = Ax + b$ where A is an $S \times R$ matrix and b is a constant vector. Since in this context the function must map $\mathbf{Z}^R \rightarrow \mathbf{Z}^S$, A is restricted to be an integer matrix and b is restricted to be in $\mathbf{Z}^S$.

The embedding constraints outlined above are first translated into constraints on the scheduling problem. The scheduling problem is then efficiently solved as an integer linear programming (ILP) problem. A user-defined delay function serves as the cost to be optimized. Finally, for each optimal schedule, consistent embeddings in the space-time lattice are explored. Since very few optimal schedules are typically found, full exploration of the space-time lattice need only be performed a limited number of times. The combined search efficiently and exhaustively explores an important subset of all possible affine embeddings and returns all designs which optimize the user-defined objective function. For the matrix-vector example, a set of 4 (symmetric) optimal schedules is found after exploring less than 100 partial designs. Each such schedule has 2 associated consistent embeddings. Corresponding RTL implementations (obtained after localization and projection of the embedded DGs) are shown in Figure 3.

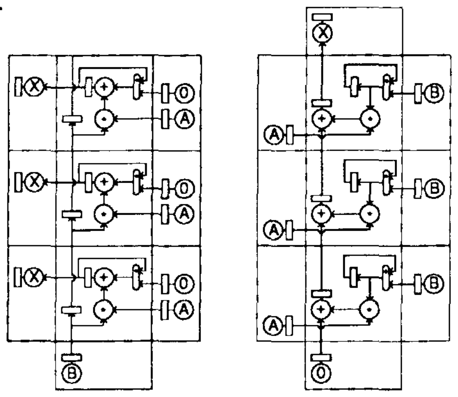


Figure 3: Two RTL-Level Implementations[6]

DESCARTES is implemented in Common Lisp and is operational through the embedding phase. Final mapping to RTL has not yet been implemented. The program has derived architectures for image and signal processing as well as non-numeric applications, and it efficiently explores design spaces in excess of 10^{12} points.

References

[1] J.A.B. Fortes, K.S. Fu, and B.W. Wah. Systematic Design Approaches for Algorithmically Specified Systolic Arrays. In V.M. Milutinovic, editor, *Computer Architecture: Concepts and Systems*, pages 454–494. North-Holland, New York, New York, 1988.

[2] S.V. Rajopadhye and R.M. Fujimoto. Synthesizing Systolic Arrays From Recurrence Equations. *Parallel Computing*, 14, No. 2:163–189, June 1990.

[3] Y. Yaacoby and P.R. Capello. Converting Affine Recurrence Equations to Quasi-Uniform Recurrence Equations. In *VLSI Algorithms and Architectures - Proceedings of the 3rd Aegean Workshop on Computing*, pages 319–328, 1988.

[6]The implementation on the right corresponds to the embedding in Figure 2.

Managing Storage for Multithreaded Computations

Robert Blumofe
MIT Laboratory for Computer Science
Cambridge, MA 02139
rdb@theory.lcs.mit.edu

In a multithreaded computation, the dynamic scheduling of threads dramatically impacts both the running time and memory usage. The effect of dynamic thread scheduling is most apparent in systems that support nonstrict semantics such as dataflow and futures, because threads execute based on the availability of data. The effect is also apparent when threads can stall due to the long latency of remote loads or loads from shared memory. In light of this dynamic behavior, a scheduler trying to execute a computation quickly often tries to expose as much parallelism as possible by keeping as many threads active as possible. Unfortunately, each active thread makes a claim on memory for an activation record, and therefore aggressively trying to expose parallelism may place excessive demands on memory capacity. We consider the problem of dynamically scheduling threads to expose sufficient parallelism for optimal speedup while not exposing more parallelism than the memory system can handle.

To formalize our goals, we consider the time and space used to execute a multithreaded computation with one processor. We assume the processor executes a single instruction at each time step, so the execution takes time T_1 equal to the total number of instructions executed. In considering the space usage, we only count the memory used for stack-based storage, and we assume each activation record takes unit space. Therefore, the execution takes space S_1 equal to the maximum stack depth. With p processors, Brent's Theorem guarantees that any scheduling policy that uses processors in a greedy fashion (never idling a processor unnecessarily) executes in time $T_p \leq T_1/p + T_\infty$, where T_∞ is the running time in an execution with an infinite number of processors. This bound is within a factor of two of optimal, since both T_1/p and T_∞ are lower bounds on the running time. We consider a p-processor execution with T_p proportional to $T_1/p + T_\infty$ to be time efficient. What about space? What should we consider to be space efficient? When a processor executes a particular thread, it requires a context equivalent to the contents of the stack at the time that thread was executed in the single-processor execution. Therefore, we allow each processor to use as much memory as the single processor used, and we consider a p-processor execution with space usage S_p proportional to $S_1 p$ to be space efficient.

We would like a scheduling algorithm that can execute any multithreaded computation efficiently in both time and space, but our first result states that no such algorithm exists. In particular, for any number p of processors, there exists a multithreaded computation such that any schedule with $T_p \leq \alpha(T_1/p + T_\infty)$ and $S_p \leq \beta(S_1 p)$ must have $\alpha\beta = \Omega(\sqrt{T_1}/p)$. In other words, there exists a multithreaded computation for which any schedule giving efficient space usage must give poor speedup and any schedule giving efficient speedup must require excessive space usage.

In light of this lower bound result, we consider scheduling algorithms for special cases of multithreaded computations. We give efficient scheduling algorithms for computations having a strict semantics for procedure invocation, and we use this result to give a technique for handling nonstrict procedure invocation. This latter result requires some care, since when space is bounded, invoking

This research was supervised by Professor Charles E. Leiserson and was supported in part by the Defense Advanced Research Projects Agency under contract N00014-91-J-1698 and by a National Science Foundation Graduate Fellowship.

procedures before their parameters have been computed can actually result in running times that are slower than when procedures always wait for their parameters.

We derive these results from our model of a multithreaded computation as a directed acyclic graph in which each node represents a unit length task and each edge represents an ordering between two tasks; see Figure 1. The *size* of the computation (the number of tasks) is T_1. Any execution of the computation must observe the ordering imposed by the edges: for any pair of tasks u and v, if there is a path from u to v, then task u must execute before task v. We define the *computation depth* as the length of the longest path in the computation, that is T_∞.

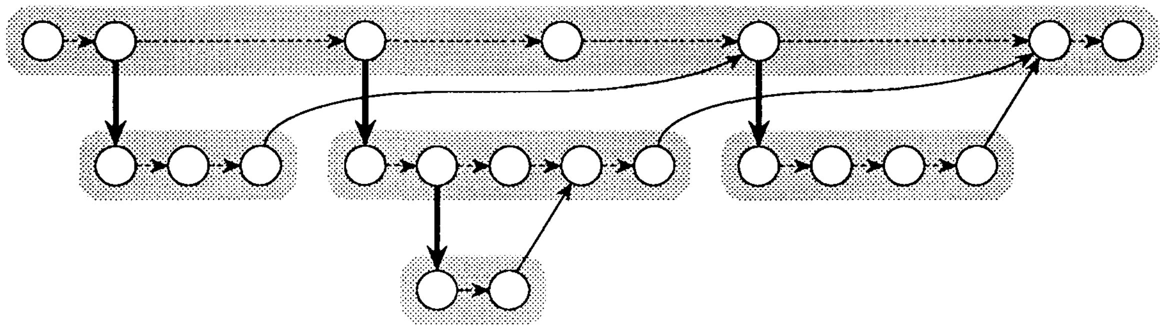


Figure 1: A multithreaded computation. Continue edges are dashed, spawn edges are thick, and data edges are thin. The threads are shaded. This computation has size $T_1 = 21$, tree depth $S_1 = 3$, and computation depth $T_\infty = 12$.

We partition the edges of the computation into three types. The *continue* edges impose the intra-thread ordering — these are the dashed edges in Figure 1 that form each thread (shown shaded) into a linear order. The *spawn* edges represent the invocation of a thread by some task in another thread. Collapsing each thread into a single node and connecting the threads by the spawn edges produces a rooted tree of arbitrary degree called the *invocation tree*. We define the *tree depth* of the computation as the depth S_1 of the invocation tree. Lastly, the *data* edges enforce the ordering required by producer/consumer relationships. If task u computes a value x and task v uses x, then a data edge from u to v ensures that task v executes after task u.

Our scheduling algorithms are all based on depth-first priority, where the depth of a task is the depth of its thread in the invocation tree. With a global depth-first priority queue, the resulting synchronous algorithm efficiently schedules multithreaded computations having strict procedure-invocation semantics. For this global depth-first algorithm we show $T_p = O(T_1/p + T_\infty)$ and $S_p = O(S_1 p)$. By incorporating a randomized load-balancing technique, we replace the global queue with p local queues — one per processor — to produce a semisynchronous algorithm. For strict computations, we show that this local depth-first algorithm has a guaranteed space bound of $S_p = O(S_1 p \lg p)$ and a high-probability time bound of $T_p = O(T_1/p + T_\infty \lg p + (\lg T_1)(\lg p))$. This time bound for the semisynchronous algorithm gives linear speedup (with a small constant factor) whenever two conditions hold: The computation is reasonably large compared to the number of processors ($T_1/\lg T_1 = \Omega(p \lg p)$), and the average available parallelism has at least $\lg p$ slack ($T_1/T_\infty = \Omega(p \lg p)$). For these reasonably large and parallel multithreaded computations, if the memory capacity scales sufficiently to incorporate $\lg p$ slack, the local depth-first algorithm almost surely executes with linear speedup.

References

[1] Robert Blumofe and Charles E. Leiserson, "Managing Storage for Multithreaded Computations," in preparation.

Clay-1: A Distributed Bit–Parallel Computer

Mike Bolotski
misha@ai.mit.edu

MIT AI Lab
703 – 545 Technology Square, Cambridge, MA 02139

Introduction

We're working on a massively parallel SIMD computer for early vision tasks. We call this VLSI-based very fine-grained malleable architecture the *Clay* machine. The central idea of the architecture is to maximize the number of bits transformed in each cycle. To do this we place as many processors on a chip as possible. We believe we can produce a chip which can transform 3200 bits per cycle. In contrast, a CM-2 processor chip operates on 16 bits, and a conventional RISC processor on at most 64 bits at once.

The Clay architecture consists of one-bit processors, connected in a mesh-with-bypass network. The individual components are mostly conventional, and the architecture could easily be misinterpreted as yet another bit-serial mesh machine. The distinctions are subtle, but result in a significant performance advantage over bit-serial systems.

The key observation is that typical bit-serial processor elements (PEs) dedicate much more silicon area to memory than to processing. The idea of the Clay architecture is to replace a single PE with a large memory by several of PEs with smaller memories, and allocate a PE to each bit of a word. As a result, the entire data word can be transformed in parallel, increasing the performance by a factor equal to the word length, with only a small increase in area. Consider a PE with 512 bits of memory and a minimal one-bit ALU whose area equals that of 32 memory bits. A group of 16 smaller PEs with 32 bits of memory each, requires only twice the area of the original PE, while delivering as much as 16 times the performance.

We call this data organization *distributed bit-parallel* (DBP), as each data word is distributed among a group of PEs, and is operated on in parallel. Such a group operating on a single word is called a *cluster*. Cluster organization is purely a software construct, and the PEs can be grouped to manipulate data of various word sizes.

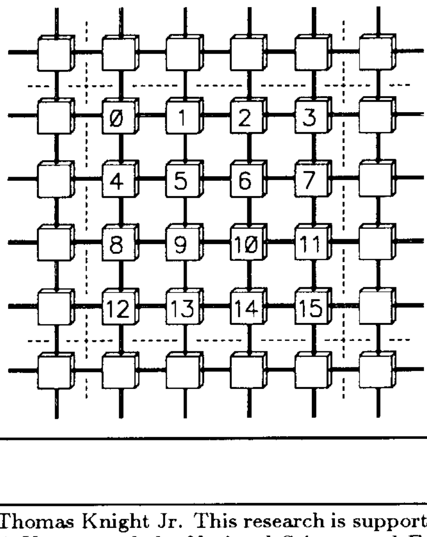
Architecture

The core of the architecture is the small, massively replicated processing element. Each PE contains 64 bits of dual-ported memory arranged in two banks, two 3-input ALUs, and 4 NEWS communication registers. Each ALU can access two bits from its associated memory bank and one from the other bank. In effect, this the first "super-scalar" bit-serial processor.

The PEs are connected in a circuit–switched mesh-with-bypass network. The bypass enhancement to the mesh allows the powerful capabilities of binary tree embedding and one-to-many broadcast, and only minor circuitry is required for its implementation. A slight delay is still incurred at each bypassed PE due to this circuitry. As a result, communication time is still proportional to distance but the constant is sufficiently small so that local communication occurs in unit time.

Arithmetic Algorithms

This subsection describes some important arithmetic operations: **`shift`**, **`accumulate`**, **`compare`**, **`add`**, and **`multiply`**. The first two use only nearest–neighbor connections; **`compare`** uses the bypass to broadcast; **`add`** uses the bypass to embed trees.



Acknowledgments: This work is conducted under the direction of Professor Thomas Knight Jr. This research is supported in part by the Defense Advanced Research Projects Agency under contract N00014-87-K-0825 and the National Science and Engineering Research Council of Canada.

A cluster is organized in row-major order, as shown above. Every fourth PE in a row is connected to the PE one row below and four columns to the left. As long as the horizontal dimension of the cluster is four, logically adjacent bits are also electrically adjacent. Throughout all these algorithms, the cluster can be considered to be organized as a line.

Shift. The simplest DBP arithmetic operation is the shift. Each processor simply replaces its bit with a neighbor's. In the case of logical shifts, the MSB or LSB must be cleared; for arithmetic shifts the MSB must be retained, which requires an extra cycle. Since the clusters are rectangular, shifts by a multiple of the row size are fast since the bits need only move vertically. Thus, a shift by 9 can be accomplished by two vertical shifts by 4, followed by a conventional horizontal shift by 1.

Accumulate. The sum of a sequence of n numbers can be evaluated in $\Theta(n)$ cycles with the carry-save adder (CSA) technique, which computes the sum and carry bits separately for each addition. The computation is purely local, and therefore independent of word size. Many computations which are usually thought of as additions can be reformulated as accumulations. As a result, the speed of operations such as region summing, counter updating, or multiplication is improved.

Compare. The comparison algorithm takes advantage of the bypass capabilities of the network. It is based on the observation that the most significant differing bit (MSDB) between two words determines which word is greater. Thus, $A > B$ when the MSDB is 1. The algorithm is straightforward: all PEs with identical bits bypass themselves and then send the value of their bit of A up to the MSB. If the MSB receives a 0, then A is smaller then B, otherwise A is greater.

Add. The idea of the well-known logarithmic-time algorithm is to "look-ahead" at the carry by computing the eventual carry into each one-bit adder. It turns out that the carry c_k into the adder computing bit k can be expressed as $c_{k+1} = g_k + p_k c_k$, where g_k is the *generate* bit and p_k is the *propagate* bit. The computation of p and g can be expressed in parallel prefix form, and the carry can thus be computed in logarithmic time.

Communication Algorithms

Inter-cluster operations on the Clay architecture are much faster than on conventional bit-serial grid arrays. The bypass mechanism allows logarithmic-time algorithms by embedding a regular binary tree in the mesh of clusters. These algorithms operate very efficiently since data is communicated over the wide data bus formed by a cluster.

Routing Operation The well-known bitonic sort algorithm operates efficiently on the architecture, and can be used as the building block for a routing primitive. A sort can implement routing by using the destination cluster ID as the sort key. A full bitonic sort of records with a 16-bit key and a 16-bit datum on a 64K processor machine requires approximately 30,500 cycles. For technologically and financially feasible machine sizes, the algorithm can operate faster on the Clay mesh than on a bit-serial hypercube architecture due to the wider data path and faster local operations.

Scan Operations The parallel prefix operator can be implemented efficiently on any network that allows embedding of binary trees. Since the mesh-with-bypass architecture provides a fast broadcast capability, it can implement the Ladner-Fischer parallel prefix algorithm. Unlike tree-based scan algorithms, which perform an up-sweep to propagate information from the leaves to the internal nodes, and then a down-sweep to send global information down to the leaves, the LF algorithm performs only a forward sweep. As a result it is both faster, and consumes less memory storage than tree-based scans.

Conclusion

Based on a preliminary processor design, we estimate that 1600 PEs can fit on a single IC implemented in a 1.0 micron technology. Since each PE is very simple, requiring approximately 8 gate delays per cycle, clock rates of 125 MHz should be easily attainable. A summary of expected performance of a single IC on various computations is shown below.

Operation	8-bit MOPS	16-bit MOPS	32-bit MOPS
Add	1670	700	360
Shift	12500	6250	3125
Accumulate	8300	4150	2075
Move	8300	3125	1040
Compare	3600	1390	480
Multiply			32

References

[1] M. Bolotski. Distributed Bit-Parallel Architecture and Algorithms for Early Vision. Master's thesis, University of British Columbia, Aug 1990.

[2] Power Efficient Computation with Distributed Bit-Parallelism. DARPA Proposal, BAA92-03, Embedded MicroSystems

Migration in Distributed-Memory Multiprocessors

Eric A. Brewer*
Parallel Software Group
brewer@lcs.mit.edu

Process migration has been used to improve load balancing, overall performance, and availability [Dou90]. The vast majority of work in this area involves the migration of Unix processes in distributed systems. This paper examines the problems of thread and object migration in homogeneous multiprocessors.

After comparing migration in distributed systems and multiprocessors, we discuss the issues involved in migrating idle objects, followed by the additional problems for migrating objects with active threads. We then introduce a novel invariant that reduces the impact of migration on normal-case (non-migrating) code. We also present two novel mechanisms for frame migration, discuss their performance, and summarize our current status.

1 Migration in Multiprocessors

The primary goal of a migration mechanism is minimal impact on the performance of objects that do not migrate. We expect migration to be rare but helpful; migration is useful primarily for active long-lived objects that would provide less competition for resources if moved elsewhere. For example, if two such objects share a processor it is worthwhile to move one of them to an idle processor.

A fundamental concept is the notion of *location transparency*, which is the property that the same code can run on any processor. Code that uses addresses specific to the current processor is not location transparent. Its addresses must be translated into corresponding addresses on the new host.

With the exception of Emerald [JLHB88], all of the previous work on migration assumes either a global address space with caching, hardware support for virtual memory, or both. We can make neither of these assumptions, both of which lead to solutions that provide location transparency for free. We also can not assume the existence of hardware tags or full/empty bits, although we will exploit them when available.

2 Migrating Idle Objects

For objects without active threads, there are two concerns, migrating the data and ensuring that all references to the object remain valid. Moving the data is quite simple; the only complication is translating any local addresses.

The primary problem involves references to the object. References that are local addresses become invalid when the object moves, thus all such addresses must be located and updated. We use a combination of residence checks and globally unique *object identifiers* (OIDs). Thus, a local invocation consists of a residence check followed by a dereference of the local address. If the object is not resident, the caller computes the object's new home from its OID and performs a remote invocation. The techniques presented so far are well established and have been implemented in Emerald [JLHB88]. Unlike Emerald, we will (essentially) eliminate the residence check on architectures with tag bits by causing a trap on access to objects that are no longer local.

3 Migrating Active Objects

The addition of active threads severely complicates migration; in particular a method must be able handle the migration of its object in the middle of the code, not just at the beginning or end.

The last item we borrow from Emerald is the *Object-Frame Invariant:* an object and a frame for a method of that object must be in the same address space. This invariant allows the code of the method to access the object using pointers and without using residence checks. Without this invariant, the normal-case performance would deteriorate severely.

A corollary of this invariant is that methods cannot directly access the slots of another object.[1] Without this corollary, if a method had pointers to three objects and one of them changed address spaces, then the frame and the other two objects would also have to move. This could lead to a severe domino effect. Thus, a method directly accesses the slots of "self", but indirectly accesses the slots of other objects so that it need not be in the same address space. Relaxing this invariant is an area of future research; e.g., an object can directly access another if it "pins down" the object during the accesses.

4 Migration Points

We introduce a novel invariant: An object cannot migrate unless all of its threads are at *migration points.* A migration point is simply a point in the code at which the object is allowed to migrate. This invariant improves performance because code can assume that its object will not migrate *between* migration points, so it can use pointers arbitrarily in these regions. Such pointers become invalid *across* migration points if the object migrates.

*Eric Brewer is supported by an Office of Naval Research Fellowship, and is a member of Bill Weihl's Parallel Software Group. Additional support provided by the National Science Foundation, grant CCR-8716884; by DARPA, Contract N00014-89-J-1988; and by an equipment grant from Digital Equipment Corporation.

[1] In some languages, methods can access the slots of objects of the same type (in addition to "self").

An implication of this invariant is that migration may be delayed. To prevent unbounded delay, we ensure that migration points are relatively frequent. For example, all synchronization points are migration points so that if a method gets blocked due to synchronization, its object is free to migrate.

In general, crossing a migration point requires verifying that the object has not migrated, and migrating the frame if it has moved. The challenging part of migration points is reducing the cost of crossing one when the object does not migrate.

The first approach to this problem is to hide the verification of no movement in the first pointer dereference after the migration point. Thus, if the object has not migrated the dereference behaves normally, while if the pointer is invalid a trap occurs that updates the state of the method to reflect the new home of the object.

The trap code is fairly complicated. First, it must deduce the intended object and the offending frame. Next, it uses a hash table to locate the *fix-up block* for that migration point. A fix-up block is a piece of code that knows the layout of the frame and the register usage at the point of the trap. The first thing the fix-up block does is move the frame to the new host. It then adjusts the values of saved registers and pointers in the frame that contain local addresses to contain the correct addresses for the new host. Finally the fix-up block jumps to the instruction that caused the trap, which causes the method to restart from the failed dereference. Since all of the local addresses have been updated the code continues without a hitch.

The fix-up block is generated automatically by the compiler. The current plan is to generated a specialized fix-up block for each migration point, although it may be possible to generate one fix-up block per method.

5 Frame Patching

On some architectures, it is not possible to force a trap on the next access to a location. For such architectures, we propose an alternative mechanism called *frame patching*. The first point to note is that when an object (locally) decides to migrate, all other frames on the processor are suspended, since only one frame runs at a time. Each frame has a saved program counter, called its *return instruction pointer* or `rip`.

The basic idea behind frame patching, shown in Figure 1, is to update the `rip` of all of an object's frames so that they point to a fix-up block. Since each frame must be at a migration point for the object to migrate, there is a valid fix-up block for every such `rip`. Thus, when the frame becomes active it will execute the fix-up block before continuing with the rest of the method. As before the fix-up block ensures that all local pointers have the correct value for the new host.

Finally, we have developed algorithms that perform frame patching lazily, thus allowing the cost to be spread out over time and amortizing some of the cost across multiple migrations.

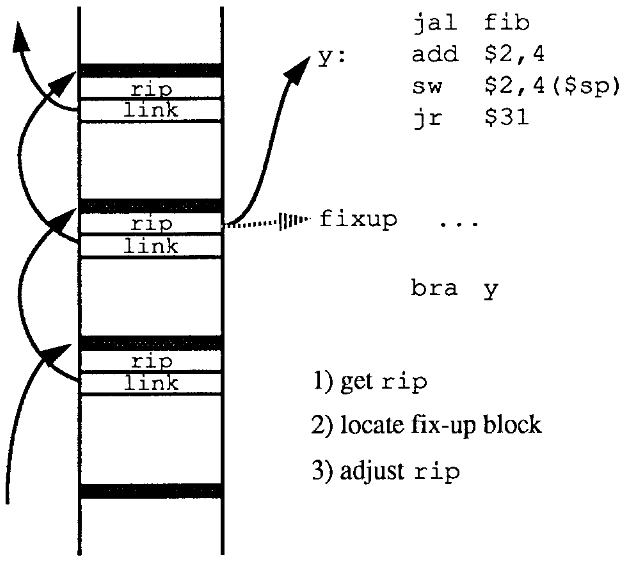


Figure 1: This figure depicts a portion of a stack with the `rip` for the middle activation record being updated to point to the corrsponding fix-up block.

6 Conclusion

We are currently generating the code required to simulate migration points, fix-up blocks and frame patching. The PROTEUS simulator will allow us to determine the effectiveness of these options both in terms of the cost of migration and the impact on non-migrating objects. Open questions that we hope to resolve include how long an object should live before it is considered for migration, how frequent migration decisions should be, and what is the right information on which to base those decisions.

This work examines migration in distributed-memory multiprocessors and addresses different problems than those of Unix-based migration. The contribution of this work includes the development of migration points, fix-up blocks, and frame patching.

The invariants we define, combined with low-cost mechanisms for the common case that objects do not migrate, should lead to a run-time system that combines the resource-balancing benefits of migration with the high performance of direct access to the fields within an object.

Acknowledgements: Thanks to Bill Weihl, Anthony Joseph and Carl Waldspurger.

[Dou90] F. Douglis. *Transparent Process Migration in the Sprite Operating System.* PhD thesis, University of California At Berkeley, Technical Report UCB/CSD 90/598, September 1990.

[JLHB88] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computers*, 6(1):109–133, February 1988.

Improving the Performance of Cache Memories Without Increasing Cache Size or Associativity

Nicholas Carter (npcarter@ai.mit.edu), NE43-611

June 24, 1992

Studies have shown that increasing the size and/or associativity of traditional cache designs becomes less effective as the size and associativity of the cache increase. In particular, once a cache has reached 32-128K of memory in size, and four-way set-associativity, the decrease in average memory access time gained by increasing the size or associativity of such a cache becomes extremely small.

This abstract describes four proposals for improving the performance of a cache memory without increasing its size or associativity[1] In order to determine the effectiveness of implementing these proposals, a trace-driven simulator was implemented which took instruction traces of program execution on a RISC System/6000 computer as inputs and predicted the number of cycles that would be required to execute the program that generated the trace if it were run on an actual machine. The SPEC benchmarks were selected as the source of the traces to be simulated. Approximately 100,000,000 instructions were traced from each of the ten SPEC benchmarks, resulting in approximately 1 billion instructions being simulated to test each of the methods.

The first method studied was a prefetching strategy that made use of the expected behavior of the load with update and store with update instructions that are implemented in the POWER architectures. These instructions calculate the address for a memory reference by adding an offset to the contents of a register, and then store the result of that computation back in the register, making them very useful in cases where a succession of equally spaced memory locations are to be read. Based on this predicted use of these instructions, it was theorized that implementing a system to prefetch the data that would be required if a given instruction of this type were executed a second time after fetching the data required for the first execution would improve performance. Unfortunately, simulations predicted that this modification would result in a performance degradation of about 4%, presumably because much of the prefetched data was removed from the cache before being used.

The second proposal involved allowing the cache to interrupt the process of bringing a cache line into memory to satisfy a miss which occurred while the line was being fetched. Since the RISC System/6000 fetches the datum required to satisfy a cache miss first from the main memory, this results in not fetching data that will probably be used in order to fetch data that is definitely needed more quickly. This resulted in a 2.9% improvement in performance overall, although performance

[1]This work was carried out at IBM's Yorktown and Hawthorne research sites, while I was an intern at IBM as part of the VI-A program. Funding was provided by the salary paid me by IBM and other personal resources. Tom Knight supervised this research.

was reduced by 3.6% on the "dnasa7" benchmark.

Another idea involved the use of a "load history table" to store data about how often cache lines are used before being replaced in the cache. Whenever a load or store instruction is executed, the load history table is checked to determine if that instruction has been executed before, and whether or not the cache line referenced by that instruction the last time it was executed was used enough to merit bringing the entire line into the cache. The use of a load history table was found to increase the performance of the machine by slightly more than 4%, depending on the size of the history table, and to improve the performance of the machine on all of the tests that were run.

The final proposal that was examined was the use of a "victim cache", as proposed by Jouppi[2]. A victim cache is a small buffer of cache lines that is used to store cache lines that have been thrown out of the main cache, so that they may be accessed more quickly than if they were returned to the main memory. This reduces the performance impact of reducing the associativity of the main cache. The simulations that were run assumed that the 64K, four-way set-associative cache that is currently implemented in the RISC System/6000 was replaced with a direct-mapped cache containing the same amount of memory and a victim cache of varying size, with a one-cycle penalty being incurred when a needed line was contained in the victim cache. Going from a set-associative to a direct-mapped cache reduced the performance of the machine by approximately 8%. Adding a victim cache resulted in a machine with performance .5%-1.1% better than that of the original machine, making the use of direct-mapped caches attractive because of the shorter access times that are possible with such a cache.

Increasing the size or associativity of the cache in a machine like the RISC System/6000 produces only very minor improvements in performance. (.2% performance improvement from doubling cache size, and .05% improvement from doubling associativity) All of the proposals that were found to improve the performance of the machine resulted in greater performance improvements than traditional methods of improving cache performance, and should require less hardware investment, suggesting that methods such as these should be considered in order to improve the performance of future architectures.

[2]Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th Annual Symposium on Computer Architecture*, pages 364-372, 1990

Multi-threaded Compilation of Id Programs

Yonald Chery[1]
MIT Computation Structures Group
NE43-202
yonald@whopper.lcs.mit.edu

I am currently implementing compiler analysis techniques [5] [6] for compiling Id, a non-strict functional language [2], which is expected to dramatically improve code performance for multi-threaded machines such as Monsoon [4] and *T [1].

Instead of values communicated via *tokens* flowing along arcs between instructions, the multithreaded model views computational progress as the run-time scheduling of interacting, multiple instruction sequences called *threads*, each of which is identified and scheduled at compile-time [3]. These threads make use of the context frame to pass data between instructions as is done in von-Neumann machines. Communicating values across threads sharing a context is also done using frame locations. The creation and initial execution of threads is accomplished through the use of `fork` and `join` instructions, which start new threads and synchronize the completion of threads, respectively. Such a computational model efficiently executes programs which yield long threads with little required synchronization.

In contrast, Id makes finding long threads difficult due to non-strictness. Consider the following example from [5] in figure 1. The figure shows the definition and two different invocation of procedure `f` (each in a "letrec" block of mutually recursive bindings).

```
                               { a = f yy 3;     { a = f 3 yy;
def f x y =                     yy = tail a;      yy = head a;
  cons (x + 2) (y + 3);         ...  }            ...  }

          (a)                      (b)               (c)
```

Figure 1: Use of non-strictness in Id programs

Depending on the invocation, the multiplication can either precede the addition (as in figure 1b) or vice-versa (as in figure 1c). As a result, both sub-expressions must be evaluated in separate threads and scheduled at run-time.

The compiler analysis techniques implemented provide improved methods for grouping instruction into larger *partitions* (code DAG's which ultimately become compiled as threads through standard code generation techniques), reducing the control flow dependencies between partitions, and performing global analysis that can be extended for doing inter-procedural analysis.

[1]Research supported by GEM Masters Fellowship and through ONR Grant No. N00014-89-J-1988 and supervised by Professor Arvind.

Local partitioning consists of repeated, alternating passes of grouping instructions within a dataflow program graph[2] *basic block* according to their input and output dependencies until no further progress can be made. Global analysis consists of propagating information across basic block boundaries to improve the partitioning of a caller or callee block. Local partitioning and global analysis are alternated until no further improvements can be made.

Once partitioned, dataflow arcs between partitions are converted to control flow arcs by inserting `frame-store` and `frame-fetch` instructions in the producing and consuming partitions, respectively. Redundant control flow arcs in the partitioned graph correspond either to fanout trees in the original dataflow program graph or from multiple values being communicated between two partitions. Removing these redundant arcs reduces the amount of `fork`ing and `join`ing performed at run-time.

Current work involves developing a new compiler intermediate representation to better expose control flow and support global analysis. Preliminary results using test cases shows these techniques to be successful in identifying large partitions. This research is intended for eventual use in the Id Compiler for *T, a multi-threaded hybrid von-Neumann/dataflow machine currently being developed by CSG and researchers at Motorola Cambridge Research Center.

References

[1] R. Nikhil, Arvind, G. M. Papadopoulos. *T: A Multithreaded Massively Parallel Architecture. In *Proceedings of 19th Annual International Symposium on Computer Architecture*, pp. 156 - 167, May 1992.

[2] R. Nikhil. Reference Manual for Id 90, Computation Structures Group Memo 284-1, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge MA, September 1990.

[3] G. M. Papadopoulos, K. R. Traub. Multithreading: A Revisionist View of Dataflow Architectures. In *Proceedings of 18th Annual International Symposium on Computer Architecture*, pp. 342-351, IEEE, May 1991.

[4] G. M. Papadopoulos. Implementation of a General Purpose Dataflow Multiprocessor, Technical Report LCS/TR-432, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge MA, August 1988.

[5] K. R. Traub. Multithreaded Code Generation for Dataflow Architectures from Non-Strict Programs. In *Functional Programming Languages and Computer Architecture '91*, Volume 523 of *Lecture Notes in Computer Science*, pp. 73-101, Springer-Verlag, August 1991.

[6] K. R. Traub, D. E. Culler, K. E. Schauser. Global Analysis for Partitioning Non-Strict Programs into Sequential Threads, In *Proceedings of ACM Conference on LISP and Functional Programming*, June 1992.

[2]Dataflow program graphs serve as the compiler's intermediate representation.

Frame Memory Management for the Monsoon Processor[1]

Derek Chiou[2]

Multiprocessor architectures require some sophisticated, yet fast memory management primitives to support procedure calls in high level languages. These primitives serve two intertwined purposes – allocation of activation area, called a *context* or a *frame*, for an iteration of a procedure, and, since a pointer to a frame contains a processor number, distribution of work across the nodes of the multiprocessor. We describe an activation frame memory management system, its implementation, and results.

1 Activation frames on Monsoon

Allocating and deallocating frames for parallel computing is not as easy for sequential computing. Since a procedure may have multiple outstanding calls to other procedures, we wind up with a *tree* of activation frames. Parallel frame allocation is a mix between stack and heap allocation on a sequential computer. A full implementation of a parallel frame manager would essentially be a heap manager, including all of the associated complexity. We simplify the problem by restricting the number of different sizes a frame can be to allow the frame allocator to run much more quickly. This constraint greatly simplifies the code of our frame manager, thus increasing its speed.

Monsoon[2] takes the position that a single frame must exist on a single processor. Thus, work is divided between processors at a procedural level. The frame allocator will, therefore, partition work across processors by how it handles requests for frames. We consider load distribution/balancing as an important part of frame allocation.

2 Frame managers for Monsoon

Over the course of the past year, we have examined many different frame managers. We have two currently being researched. One has been heavily optimized by code duplication to avoid unnecessary run-time evaluation. The other is far more complicated since it implements a deferred coalescing buddy-system. All frame managers are written in MONASM[3], our assembly language, for optimal speed.

Both frame managers use approximately the same algorithm, quick-fit, for frame allocation. When a frame is requested, the correct quick-list is checked for a frame. If there are no frames on the correct quick-list, the algorithm attempts to allocate a frame from the tail of the frame memory. If there is not sufficient memory in the tail to allocate a frame of the desired size, the algorithm looks for a frame larger than the desired size. If a frame of a larger size is not found, behavior between the two frame managers differs. The first frame manager, $rts_{inlined}$, will return an error and halt the machine while the other frame manager, $rts_{coalesce}$, will attempt to coalesce the frames. Both frame managers do remote frame management, that is, the processor on which a frame resides manages that frame. If processor P_i desires a frame from P_j, P_i must send a request for a frame to P_j.

[1]Supervisor: Prof. Gregory M. Papadopoulos. Funding is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-89-J-1988.

[2]Office: NE43-203. Email: derek@abp.lcs.mit.edu

Program	1pe	2pe	4pe	8pe
GAMTEB	1	1.96	3.86	7.44
PARAFFINS	1	1.95	3.52	5.52
SIMPLE	1	1.87	3.48	6.21
MATRIX-MULTIPLY	1	1.99	3.87	7.23

Table 1: Speedup on Monsoon Hardware

3 Load distribution and parallelism

As noted earlier, load distribution is an important part of frame allocation. Load distribution is the partitioning of work to different processors and is a significant problem for parallel machines. The thinking in our group has been that dataflow exposes so much parallelism that a simple and approximately random load distribution scheme, though it might be somewhat unbalanced, should be able to keep all processors reasonably busy.

On this thinking, and after experimenting with a few other options, we arrived at a simple round-robin scheme to distribute work across all the processors. Every processor has its own set of round-robin counters and each frame size has its own round-robin counter. The load balancing scheme works well with all of the benchmark programs we have run so far.

4 Current Results and Future Work

We have been running an early version of $rts_{inlined}$ on Monsoon hardware and simulators for several months. It is very robust and easy to use. Its speedup performance is shown in Table 1. GAMTEB and MATRIX-MULTIPLY speed up very nicely. SIMPLE seems to lock up the machine on anything but modest loop bounds, strangling the amount of parallelism we can exploit and destroying our speedup. We are looking into this problem. Currently, we feel that there is a lack of parallelism in PARAFFINS, limiting its speedup. $rts_{inlined}$ is about 30% faster than its earlier version. Frame management overhead is completely dependant on the program being run. It ranges from virtually nothing for MATRIX-MULTIPLY to around 13% for SIMPLE and GAMTEB and about 30% for PARAFFINS.

We believe that we have achieved the objective of building an efficient frame manager for Monsoon. Future work includes evaluating the performance of the frame manager for larger machine configurations and exploring further modifications to the frame manager for additional speed.

References

[1] M. J. Beckerle. Internal design for mint. Technical report, Motorola, Inc., Cambridge MA, Oct 1990.

[2] D. E. Culler and G. M. Papadopoulos. The explicit token store. *Journal Of Parallel and Distributed Computing*, 10(4):289–308, 1990.

[3] K. R. Traub. Monasm reference manual. Technical Report MCRC-TR-5, Motorola, Inc., Cambridge MA, Apr 1990.

Reconfiguration of Multipath MIN Architectures

Fred Chong
MIT Artificial Intelligence Laboratory
ftchong@ai.mit.edu

As the number of components in large-scale multiprocessors becomes large, the fault tolerance of such machines becomes increasingly important. We examine methods of reconfiguring a multiprocessor which has suffered faults in its interconnection network. We concentrate upon the decision of which processing nodes should be used and which nodes should be shut down. Due to their high fault-tolerance, we focus upon architectures which use *multipath* multistage interconnection networks (multipath MINs). Multipath networks have multiple paths between any input and any output. The routers used to construct these networks are characterized by *radix* and *dilation*. The radix of a router refers to the number of logical directions the router switches to. The dilation of a router refers to the number of redundant channels in each of these channels.

An important multipath MIN is the randomly-wired multibutterfly, shown in Figure 1. Multibutterflies have been shown, in theory, to possess substantial fault tolerance and performance [Upf89]. Leighton and Maggs [LM92] used a *fault-propagation* reconfiguration algorithm to prove that no matter how an adversary chooses k routers to fail, there will be at least $N - O(k)$ inputs and $N - O(k)$ outputs between which permutations can be routed in $O(\log N)$ router cycles, for an $N \times N$ network.

However, these asymptotic results do not guarantee that fault-propagation is a practical algorithm for reconfiguration. In fact, the algorithm initially appeared too conservative to use in practice. Fault-propagation centers upon the following recursive definition of a *blocked* router: a router is blocked if it is faulty or if any of its logical directions leads to only blocked routers. This definition of blocking is propagated stage-by-stage backward from the last stage to the first stage of the network. Any processing node which is connected to only blocked routers is shutdown in the reconfiguration. Fault-propagation is conservative because it discounts the utility of blocked routers. A blocked router is not necessarily useless. It may still have many usable channels.

We compared fault-propagation to a non-conservative *multi-hop* algorithm. The multi-hop algorithm shuts down processing nodes only when absolutely necessary — when a node has no surviving input or output connections to the network. To allow communication between all nodes, a multi-hop system must allow messages to be routed through intermediate destinations. Figure 2 shows our simulation results for both reconfiguration strategies. Our results show that the the conservative fault-propagation criterion produces the best performance. Synchronization requirements of applications make it critical to eliminate nodes with poor network connections.

Further details of our work are available in [CK92]. We examine another class of multipath networks, the *maximal-fanout* networks. We present an $\Omega(n^{\frac{1}{4}})$ lower time bound for a worst-case permutation these networks. We further show how a randomized approach avoids this worst case. We show empirically that maximal-fanout networks perform just as well as randomly-wired multibutterflies.

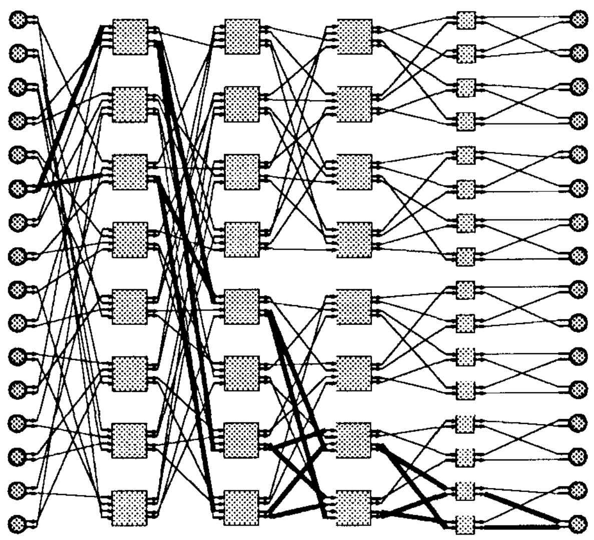
References

[CK92] Frederic T. Chong and Thomas F. Knight, Jr. Design and performance of multipath MIN architectures. In *Symposium on Parallel Architectures and Algorithms*, San Diego, California, June 1992. ACM. To appear.

[LM92] Tom Leighton and Bruce Maggs. Fast algorithms for routing around faults in multibutterflies and randomly-wired splitter networks. *IEEE Transactions on Computing*, 41(5):1–10, May 1992.

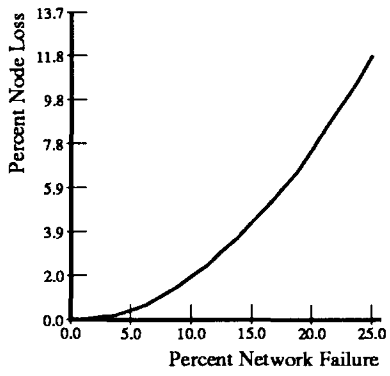
[Upf89] E. Upfal. An $O(\log N)$ deterministic packet routing scheme. In *21st Annual ACM Symposium on Theory of Computing*, pages 241–250. ACM, May 1989.

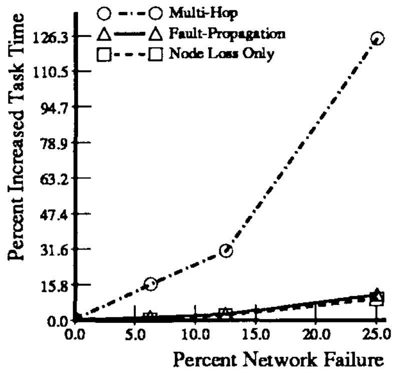
Acknowledgments: This work is conducted under the direction of Thomas F. Knight, Jr. This research is supported by an Office of Naval Research Graduate Fellowship and the Defense Advanced Research Projects Agency under contract N00014-87-K-0825.



A randomly-wired four-stage multibutterfly connecting 16 endpoints. Each component in the first three stages is a radix-2, dilation-2 router. To prevent any unique critical paths between endpoints, the last stage is composed of radix-2, dilation-1 routers. The multiple paths between a selected pair of endpoints are shown in bold.

Figure 1: Randomly-Wired Multibutterfly





Left: The number of processor nodes lost after reconfiguration is plotted against the number of uniformly distributed router failures for randomly-wired multibutterflies. At 25 percent network failure, only about 10 percent of the nodes are lost.
Right: The average time to route a particular task is plotted against the number router failures. The ***node loss only*** curve is a reference line which plots the performance degradation due solely to the reduced number of processors. The next two curves show the performance of the multi-hop and fault-propagation algorithms on multibutterfly-based systems. While the multi-hop system degrades significantly, the fault-propagation system suffers very little additional performance degradation due to loss in network bandwidth.

Figure 2: Node Loss and Performance under Network Failure

M.I.T.
TRANSIT
PROJECT

Scan-Based Testability for Fault-Tolerant Architectures (Abstract)

André DeHon
(andre@ai.mit.edu)
NE43-791, x3-5868

MIT AI Lab
545 Technology Square, Cambridge, MA 02139
May 8, 1992

With the standardization of Test Access Ports (TAPs) and boundary-scan techniques in IEEE-1149.1-1990 [Com90], vendors are beginning to make components with scan-based TAPs readily available. Nonetheless, the facilities offered by TAP interfaces such as the IEEE-1149 standard are not well-suited for fault-tolerant system architectures. The singular and serial nature of the scan path exposes a critical single point of failure in the testability system. Architects are forced to either use a few, long serial scan chains or use many short scan chains. The former allows a fault in a scan path to affect a large number of components while the latter requires significant wiring for the control of many scan paths. Furthermore, standard TAPs provide no facilities for bringing small portions of the system into test-mode while leaving the remainder of the system in normal operation. In fault-tolerant architectures where the system can function without all components on-line, these all-or-nothing testing modes can be inconvenient.

We have developed three simple additions to standard scan practices which allow scan techniques to be utilized effectively in a fault-tolerant setting. The basic techniques introduced are:

1. Multi-TAP scan architecture – each component is given multiple Test Access Ports allowing the component to be accessed from any of several scan paths. Figure 1 shows the basic scan architecture for a dual-TAP component.
2. Port-by-port selection – each *channel* on a component can be independently disabled.
3. Partial-external-scan – each *channel* can be scanned in boundary-test mode independently of the operation of other channels on the same component.

When combine these additions provide a scan architecture which is well adapted for a large class of fault-tolerant systems. In particular, the additions allow:

1. Minimized impact of scan path faults on system diagnosability
2. Minimally intrusive in-operation fault-diagnosis
3. In-operation reconfiguration for:
   - fault-masking
   - repair

These additions and the capabilities they provide are developed in [DeH92].

Acknowledgments: This work is conducted under the direction of Principal Research Scientist Thomas Knight Jr. This research is supported in part by the Defense Advanced Research Projects Agency under contract N00014-91-J-1698. This material is based upon work supported under a National Science Foundation Graduate Fellowship. Any opinions, findings, conclusions or recommendations expressed in this publication are those of the author and do not necessarily reflect the views of the National Science Foundation.

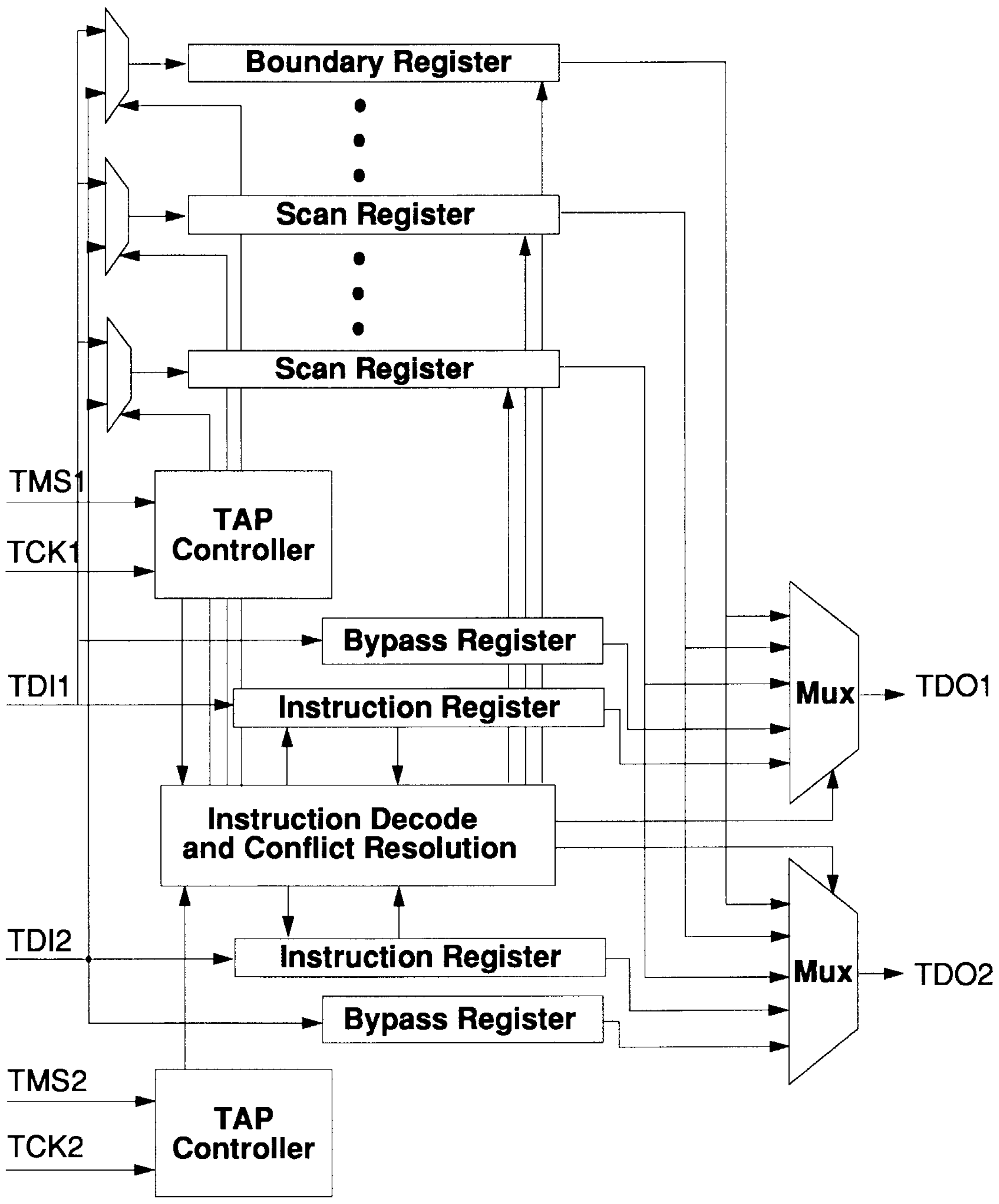


Figure 1: Scan Architecture for Dual-TAP Component

References

[Com90] IEEE Standards Committee. *IEEE Standard Test Access Port and Boundary-Scan Architecture.* IEEE, 345 East 47th Street, New York, NY 10017-2394, July 1990. IEEE Std 1149.1-1990.

[DeH92] André DeHon. Scan-based testability for fault-tolerant architectures. Transit Note 60, MIT Artificial Intelligence Laboratory, January 1992.

Serializing Parallel Programs (Abstract)

Michael D. Ernst[1]

Programmers would like to be able to write a single program for both parallel and serial computers. Historically, the focus has been on parallelizing serial code. In this paper, we argue that the reverse—serializing parallel code—is both more natural and more efficient. We introduce and evaluate three methods for serializing parallel code—unrolling, loop common expression elimination, and finite differencing—and compare them to parallelization. All three methods are based on a form of common subexpression elimination across loop boundaries.

An algorithm's fastest implementation depends crucially on the target architecture. Fast serial algorithms resist parallelization or vectorization because of dependences between loops which were introduced to reduce the total work done by the program. Parallelization requires detection and removal of these dependences in order to permit loop iterations to execute concurrently. The maximally parallel implementation of an algorithm achieves the greatest speedup on a parallel machine with sufficiently many processors, but it may be inefficient when run on a machine with too few processors. Work may be repeated on several processors in order to avoid dependences and permit the processors to operate independently. Efficient serialization requires the elimination of this redundant computation.

When the parallelism in the problem exceeds that available in the hardware, the best implementation is a hybrid parallel/serial one which breaks the problem into pieces, then uses a fast serial algorithm to evaluate each of the pieces. The programmer should not be burdened with this task, since he usually cannot know how many processors his program will be run on. Furthermore, specifying two different algorithms is error-prone, particularly when the programmer must direct their interaction. The transformations described in this paper take care of those details.

We choose to serialize parallel code rather than the reverse because we can obtain better performance on parallel computers and, often, on serial ones as well. An equally important motivation is that explicitly parallel programs written in a high-level fashion (for instance, in the data-parallel paradigm) tend to be easier to write, read, and debug than serial ones, and are much simpler than arbitrary parallel programs.

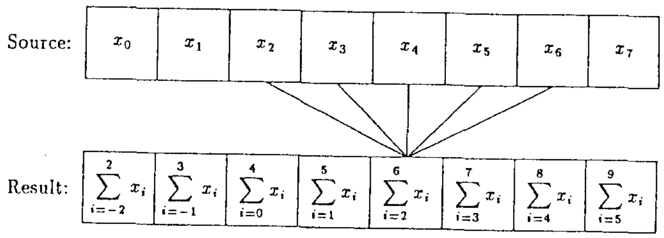

Source:
x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7
Result:
$\sum_{i=-2}^{2} x_i$ $\sum_{i=-1}^{3} x_i$ $\sum_{i=0}^{4} x_i$ $\sum_{i=1}^{5} x_i$ $\sum_{i=2}^{6} x_i$ $\sum_{i=3}^{7} x_i$ $\sum_{i=4}^{8} x_i$ $\sum_{i=5}^{9} x_i$

Figure 1: The five-element example, a Jacobi-like window sum operation.

```
for i = 2 to n-3
  newx[i] = (x[i-2] + x[i-1] + x[i]
            + x[i+1] + x[i+2]) / 5
```

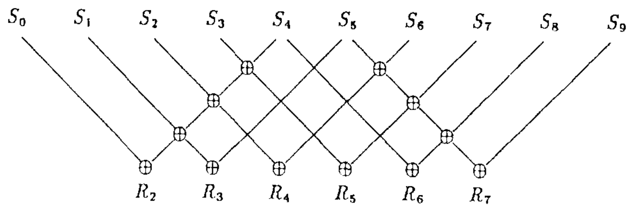
Figure 2: Elementwise (data-parallel) implementation of the window sum. Each result array element can be computed in parallel.

A one-dimensional Jacobi-like relaxation problem will be used as an example to demonstrate the three methods. The input is a vector of numbers, and the goal is to compute, for each i, the average of the five nearest values (those with indices $i-2$ to $i+2$). This computation is graphically depicted in figure 1, in which a line between a source and result array element indicates that the result array element depends on the source array element.

The first method is to unroll the loop and perform ordinary common subexpression elimination; figure 3 shows that 6 result elements can be computed with 12 additions, for an average cost of 2 operations per element. Finding a good unrolling is difficult in its own right: unrolling by one element less or more would increase the cost per element to 2.2 or 2.3, respectively.

[1]Author's address: MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139. Email: mernst@theory.lcs.mit.edu.

This research was supervised by Charles Leiserson and supported by a National Defense and Science Graduate Fellowship and by Defense Advanced Research Project Agency contract N00014-91-J-1698.



S_0 S_1 S_2 S_3 S_4 S_5 S_6 S_7 S_8 S_9
R_2 R_3 R_4 R_5 R_6 R_7

Figure 3: The five-element sum after unrolling to expose six iterations and finding the optimal set of common subexpressions.

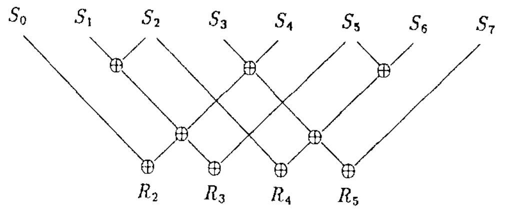


Figure 4: When loop common expressions are taken into account, computations performed by previous iterations help the current iteration. When computing the next set of four results, the sum $S_5 + S_6$ can be reused in the same place that $S_1 + S_2$ occupy in this diagram, resulting in a savings of 1 operation per 4 elements.

The best unrolling can be quite large, resulting in code size explosion. Another difficulty is that finding the optimal set of common subexpressions is NP-complete [1].

The second method is to use loop common expressions, which are expressions that can be used by more than one loop iteration. Iteration i arranges its computations so as to help iteration $i+1$, possibly resulting in slightly increased costs for iteration i, relative to ordering its computations in the greediest way. Any extra cost is offset by the fact that iteration $i-1$ has done the same thing, relieving iteration i of some work it would otherwise have to do. See figure 4 for an example. This method performs well and may require fewer temporaries and less interprocessor communication than the other methods, but its per-result operation count is usually slightly higher.

The third method is an extension of the method of finite differencing [2, 3]. The difference between the values computed by two loop iterations is added to a previous result to produce a new one. Such a strategy is worthwhile when, given the final result for iteration i, it is easier to undo some work and then compute result $i+1$ than to compute it from the intermediate results that were used in computing result i. (This method is never worthwhile in straight-line code.) See 5 for an example of such code; figure 6 shows the computation performed. This transformation can only be

```
runningsum = x[0] + x[1] + x[2] + x[3];
for (i = 2; i <= n-3; i++)
  { runningsum = runningsum + x[i+2];
    newx[i] = runningsum / 5;
    runningsum = runningsum - x[i-2]; }
```

Figure 5: Running sum implementation of the five-element sum.

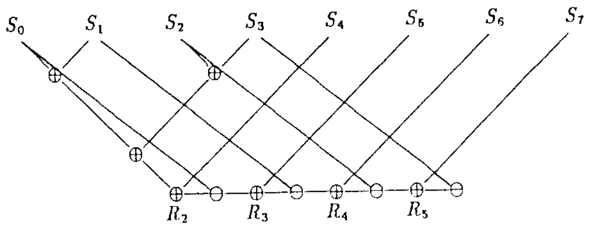


Figure 6: Diagram of the operations performed by the running sum implementation for each result element. Unlike the other two methods, no unrolling is required and after the initialization code, it requires 2 operations per element.

performed if the operation (addition, in our example) is commutative and has an inverse; all three methods require associativity. This method can suffer from numerical instability if the operation does not have an exact inverse—few computer implementations of arithmetic operators do—but it requires very little unrolling and its operation count is quite low.

Two of these methods have been implemented and show great promise; an implementation of the third is underway. Each is best in a particular set of circumstances, and speedups ranging to ten times have been observed for certain problems. For typical programs, the speed improvement is more modest, but still noticeable.

References

[1] A. V. Aho, S. C. Johnson, and J. D. Ullman. Code generation for expressions with common subexpressions. *Journal of the ACM*, 24(1):146–160, January 1977.

[2] J. Earley. High level iterators and a method for automatically designing data structure representation. *Computer Languages*, 1(4):321–342, 1975.

[3] Robert Paige. Transformational programming: applications to algorithms and systems. Technical Report DCS-TR-118, Rutgers University Department of Computer Science, New Brunswick, New Jersey, September 1982.

A Systems Language Compiler for the J-Machine [1]

D. Brennan Gaunce
Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139
bren@ai.mit.edu

Introduction

This abstract describes the J compiler, a systems language compiler for the J-Machine. The J-Machine is a fine-grain concurrent computer comprised of up to 65536 36-bit Message Driven Processors (MDP) which communicate through a low latency network [1]. The J language is modeled after the C programming language with additional syntax to provide support for accessing primitives in the underlying architecture. The main goal for the J compiler is to provide a systems programming language for the J-Machine by supporting access to the underlying hardware. The language should allow efficient communication without imposing a high runtime overhead. A secondary goal is compatibility with the C language; in the future, the compiler should run most C programs on a single processor with little or no modification.

Language Enhancements

The J language provides many enhancements to the C language to allow full accessibility to the underlying hardware. The most prominent feature of the language is the function declaration.

A function may be declared as a *subroutine*, *fault handler*, or *handler*. A *subroutine* is a function which is executed immediately on the same processor as its caller. This is useful for executing procedure calls without suspending the current thread. A *fault handler* allows the programmer to set up a function which will be executed when a fault occurs (e.g. a type fault). *Handlers* are functions which execute on a different node. Handlers may be invoked either *synchronously* or *asynchronously*. A synchronous function invocation suspends the current thread until a reply message is received from the callee; an asynchronous invocation continues executing the current thread concurrently with the callee. If a thread tries to use the result of an asynchronous call then a fault occurs, suspending the thread until the result is received. Function declarations themselves may be specified as synchronous or asynchronous to allow certain optimizations to be performed. For example, a function declared as asynchronous void need not reply to the caller.

Other enhancements in the language include tags to support tagged words, multiple return values, segment descriptors, and priority function invocations.

Implementation

Compilation consists of parsing, type checking, code generation, and optimization. Parsing and type checking are similar to existing compilation techniques. The output of the parser is a syntax tree. The type checker decorates the tree with types and similar annotations. Code generation transforms the syntax tree into a *complex* intermediate code similar to *complex J* described in [3]. This intermediate language resembles the MDP instruction format, except that a symbolic register set is used, and operands are not as restricted. Code generation is followed by a series of optimizations, including copy propagation, constant folding, dead code elimination, and peephole optimization. Next, registers are allocated and spill code introduced. The complex intermediate language is then transformed to a *simple* intermediate language using legal MDP operands. Finally, several MDP-specific transformations are performed, such as send folding and long branch calculation. Following optimization, MDP assembly code is emitted.

Rules

The rules language allows the J compiler to be maintained easily without writing any source code. Each rule maps a pattern of instructions on the left hand side to another set of instructions on the right hand side. A

[1] William Dally both supervised and contributed to this work. The research described in this paper was supported in part by the Defense Advanced Research Projects Agency under contracts N00014-88K-0738 and N00014-87K-0825 and in part by a National Science Foundation Presidential Young Investigator Award, grant MIP-8657531, with matching funds from General Electric Corporation and IBM Corporation.

set of instructions only matches if each instruction in the left hand side matches instructions inside a basic block. Each instruction pattern contains patterns specifying both an operator and its operands. Operators model MDP instruction opcodes. Operator patterns may be literal, matching only a particular operator, or an identifier, preceded by a question mark, which match any operator. Examples of operator patterns are `ADD` and `?op`. Similarly, operand patterns may be literal or identifiers. Operand identifier patterns may also contain a predicate, represented by a letter preceding the operand, which specifies that the pattern can match only certain operands. For example, `R?op1` will only match register operands. Other examples of operand patterns are `A?op3`, `?reg`, and `32`. Finally, *escape clauses* back to the compiler allow more general rules to be formed.

Examples of rules are shown in Figure 1. Rule (1) is a typical peephole optimization rule. Rule (2) is a general rule, stating that an immediate in the first operand should be switched with a register in the second operand if the operator is associative. This rule is needed since only registers are allowed in the first operand position in many MDP instructions. Finally, Rule (3) exhibits a rule to help fold `SEND` instructions. Most optimizations in the compiler are performed by rules. The rules language provides an effective mechanism for transforming the intermediate language, and is instrumental in maintaining MDP specific optimizations.

```
(1)  MUL     ?op1, 1, ?op2        ==> MOVE    ?op1, ?op2

(2)  ?op     I?op1, R?op2, R?op3   :  is_assoc ?op
     ==>
     ?op     R?op2, I?op1, R?op3

(3)  SEND    ?a, ?priority
     SEND    R?b, ?priority
     ==>
     SEND2   ?a, R?b, ?priority
```

Figure 1: Examples of rules

Related Work

Concurrent Smalltalk [2], the most substantial programming system for the J-Machine, has been suggested as a systems language, although a large fraction (up to 70%) of program execution is typically consumed by the accompanying runtime system. Additionally, the Id language introduces dataflow computation to the J-Machine [3].

Future Effort

Currently, the J compiler produces assembly code for a suite of several tests. This suite is executed on the hardware regularly, typically on 8x8x1 cubes. Several areas for further effort in the compiler include additional functionality, better optimization, standard libraries, debugging, and profiling. Several groups are interested in using the J language as a programming environment or as a back end to other languages. Finally, a collaboration among the different programming systems to develop a single back end would focus the optimization effort and produce more efficient and uniform object code for the J-Machine.

References

[1] William J. Dally et al. The J-Machine: A fine-grain concurrent computer. In G.X. Ritter, editor, *Proceedings of the IFIP Congress*, pages 1147–1153. North-Holland, August 1989.

[2] Waldemar Horwat. A concurrent smalltalk compiler for the message-driven processor. AI Memo, MIT, 545 Technology Sq., Cambridge, MA 02139, May 1988. SB Thesis.

[3] Ellen Spertus. Dataflow computation on the J-Machine. AI technical report 1233, MIT Artificial Intelligence Laboratory, 1990.

The Coprocessor Host Interface Chip

Nikhil Gautam*
NE43-503
ncgautam@lcs.mit.edu

Introduction

The Telemedia, Networks and Systems group is involved in the design and the deployment of a distributed video system. Host workstations, display units, and cameras are to be interconnected by a high-speed broadband network, called VuNet [3]. This is a Local Area Network (LAN) that will support Asynchronous Transfer Mode (ATM), a new communications standard set by the CCITT for the Broadband Integrated Services Network. In VuNet, data will travel in fixed-size (53-byte) packets called cells.

The functionality of the Coprocessor Host Interface Chip is to transfer cells between an R3000-based Decstation 5000, and VuNet. Typically, host interfaces to ATM networks have been designed so that the network has to be addressed via a memory-mapped standard I/O bus [1, 4]. With the coprocessor host interface, however, the network appears to reside in the registers of a tightly-coupled coprocessor. It is expected that this interface will provide an increased throughput over our Turbochannel-based interface, currently being used.

Functional Description

In basic terms, the chip acts upon the coprocessor instructions that it receives from the MIPS R3000 processor system. These instructions can be loads and stores between the coprocessor's registers and main memory, or moves between the coprocessor's registers and the CPU registers. Other instructions allow the user to set control registers on the chip, or to carry out internal data transfers within the chip.

*This work is being supervised by Prof. David Tennenhouse. This work is part of the Aurora Project, which is funded by DARPA and NSF through the auspices of CNRI. Initial design work on this project was done by David Martin, then a VI-A student in the Telemedia, Networks and Systems group.

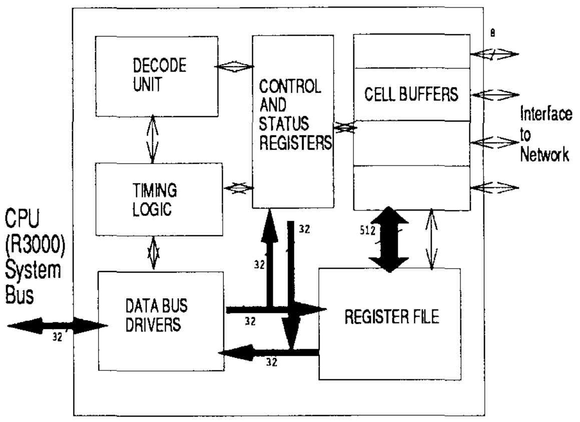


Figure 1: Functional Block Diagram

The block diagram in Figure 1 illustrates the various modules in the coprocessor. The chip consists of cell buffers, a register file[1][2], control registers, a status register, a decoder unit, a timing generator, and memory drivers.

To send a cell into the network, the CPU would read the status register of the chip, to determine whether there is room in the output cell FIFO. Then the CPU can load the cell directly into the coprocessor register file from main memory, in 32-bit word chunks. Next, the CPU would issue an instruction that transfers the entire cell from the chip's register file to an output cell FIFO on the chip in one cycle. Then, it is upto the network to clock the data out from the FIFO. While an ATM cell is 53 bytes, the coprocessor FIFOs can be programmed to deal with cells of upto 64 bytes.

In the opposite direction, the network would clock a cell into an input cell FIFO of the chip. The

[1]The cell buffers and register file were already designed by Jason Hickey at Bellcore for his cell engine chip, and those same designs were used for this project too. The use of these designs influenced parts of the design of the coprocessor chip.

non-empty cell FIFO would then cause the chip to generate an interrupt signal. The CPU would then react to the interrupt by transferring an entire cell (upto 64 bytes) from the chip's input cell FIFO to the chip's register file. And finally, the CPU would issue a number of store instructions to transfer the data in 32-bit chunks from the coprocessor register file to the R3000 main memory.

Chip Description

There are four cell buffers on the chip, which can be configured to be input or output FIFOs. Each cell buffer can hold up to four 64-byte cells. The register file consists of 64 32-bit registers, and has one 32-bit read port, one 32-bit write port, and one 512-bit read/write port, which allows an entire 64-byte cell to be transferred to (or from) a cell buffer in one cycle.

The decoder unit is responsible for decoding the instruction coming from the CPU and generating the appropriate signals for the other parts of the chip. The memory drivers interface to the system bus of the MIPS R3000. This bus is 32-bits wide, bidirectional, and multiplexed between instruction and data values. The timing generator is responsible for buffering the clock signal to the various parts of the chip, and handling CPU stalls.

This chip is currently being fabricated by VTI in 1.2 micron technology. The die size is 7.6mm by 5.2mm, and the packaging used is a 223-lead high-performance ceramic pin grid array.

Performance

It is hoped that this coprocessor-based interface will increase the bandwidth in and out of the workstation, compared with the Turbochannel-based interface currently in use in our group. This increase is expected for several reasons. One reason is that the coprocessor will be directly attached to the R3000's internal system bus, which is a high bandwidth bus (32 bits at 40MHz).

Another reason for performance improvement is the potential gain from avoiding cache misses. When reading values from the Turbochannel interface, a cache miss is bound to occur, since data coming from the network is not already in the cache. However, with the coprocessor interface, the values will be read from the coprocessor register file, which is not memory mapped, thereby bypassing the cache.

Further, the R3000 instruction set allows words to be written directly between a coprocessor's registers and main memory. In the case of the Turbochannel interface, however, data has to be moved via the CPU's general registers. Thus, in the coprocessor interface, a fewer number of instructions need to be issued by the CPU, thereby increasing the bandwidth.

Initial Measurements

Some initial measurements were taken, which compared the bandwidth of the Turbochannel interface with the coprocessor interface. Since the coprocessor is still being fabricated, data was written to and from the *floating point* coprocessor's registers. These measurements show that the bandwidth increased by a factor of about 2.1 (throughput increased from 128 Mbps to 275 Mbps) in the transmit direction, and by a factor of about 4.8 (throughput increased from 56 Mbps to 271 Mbps) in the receive direction.

References

[1] Bruce S. Davie. A Host-Network Interface Architecture for ATM. In *SIGCOMM 1991*, September 1991.

[2] Jason Hickey. *ATM Cell Processor*. Bellcore.

[3] David L. Tennenhouse. The ViewStation Research Program on: The Design and Deployment of Distributed Video Systems. DARPA Proposal.

[4] C. Brendan S. Traw and Jonathan M. Smith. A High-Performance Host Interface for ATM Networks. In *SIGCOMM 1991*, September 1991.

Performance Evaluation of Network Interfaces[1]

Dana Henry and Chris Joerg
MIT Lab for Computer Science
545 Technology Square, Cambridge MA, 02139
dana@abp.lcs.mit.edu, cfj@abp.lcs.mit.edu

We have examined two aspects of network interface design which can dramatically affect the cost of interprocessor communication: the acceleration of frequent operations via simple hardware mechanisms, and the physical placement of the network interface with respect to the processor. Our performance study demonstrates the importance of these features [HJ92].

The hardware mechanisms we consider are those in NIC [HJ91a][HJ91b], a network interface chip which we have designed and extensively simulated at the RTL level. The basic NIC architecture consists of 14 interface registers, an input message queue, and an output message queue. Of these 14 registers, five output registers contain the words of the message being composed; five input registers contain the words of a received message; and the rest provide control and status information (such as current queue sizes).

In addition to accessing the interface registers, the processor communicates with NIC via several commands. The SEND command composes a message from the five output registers and appends it to the output queue for sending. The NEXT command removes the message at the front of the input queue and places its values in the five input registers.

Three hardware mechanisms accelerate the handling of messages in NIC: encoded message types, fast reply and forward modes, and hardwired message interpretation.

Encoded Message Types: Each message must somehow identify the message handler to be invoked when the message arrives at its destination. Typically, the id of the handler is specified in a separate word of the message. This word has to be generated and stored in the interface. To avoid this overhead, we allow the id's of frequently invoked handlers to be encoded into a four-bit type field. This type field is automatically specified as part of each SEND command.

Fast Reply and Forward Modes: When a programmer decides to reply to a message, some fields of the message to be sent are already in input registers. In a typical network interface the programmer will have to explicitly move these fields into the output registers. To avoid this overhead, we have implemented two special modes of the SEND command, the reply mode and the forward mode. When one of these modes is used, the SEND command composes an outgoing message using certain input registers in place of certain output registers.

Hardware Message Interpretation: Once a message arrives at its destination, the programmer must somehow interpret the message. In a typical network interface, the programmer will have to check the status register to find out if there are any exceptional conditions and if a message has arrived, read the type of the arrived message, use the type to compute the instruction address of the handler for that message, and finally jump to that handler. We have reduced this overhead to one or two instructions by precomputing the address of the handler into a special interface register.

In addition to the hardware mechanisms, the efficiency of a network interface is affected by the physical placement of the network interface registers with respect to the processor. We consider three different placements in our study: inside an off-chip cache, inside an on-chip cache, and inside the processor's register file. If the registers are mapped into a cache, as in NIC, each reading or writing of an interface register will consume processor cycles. In our performance study, we have assumed two delay slots for reading via a load from an off-chip cache and no delay slots for reading via a load from an on-chip cache. SEND and NEXT commands can be sent simultaneously via the low bits of load and store addresses. On the other hand, if the interface registers are included in the processor's register file, reading or writing an interface register no longer takes a separate instruction. Moreover, the SEND and NEXT commands can be incorporated into the unused bits of every triadic instruction.

In our performance study, we have analyzed two parallel programs using each of the three network interface placements both with and without our hardware mechanisms. The first program, a 100 by 100 matrix multiply, subdivides matrices into 4 by 4 blocks and computes their products. The second program generates every distinct paraffin isomer, a hydrocarbon, up to size 14 [AHN88]. Both programs have been written in the non-imperative subset of the Id [Nik90] programming language and compiled for the TAM [CSS$^+$91] programming model, a relatively fine-grain programming model.

Figure 1 shows the dynamic number of RISC instructions for the two programs under each network interface model. We have computed these numbers by using a software simulator of TAM to get the dynamic instruction counts. We then replaced the dynamic count of each TL0 instruction by the appropriate number of RISC instructions. Each bargraph in Figure 1 is divided into two components. The clear, top component corresponds to the total number of instructions

[1]This work is being supervised by Prof. Greg Papadopoulos. This report describes research done at the Laboratory of Computer Science of the Massachusetts Institute of Technology. Funding for the Laboratory is provided in part by the Defense Advanced Research Projects Agency under Office of Naval Research contract N00014-89-J-1988.

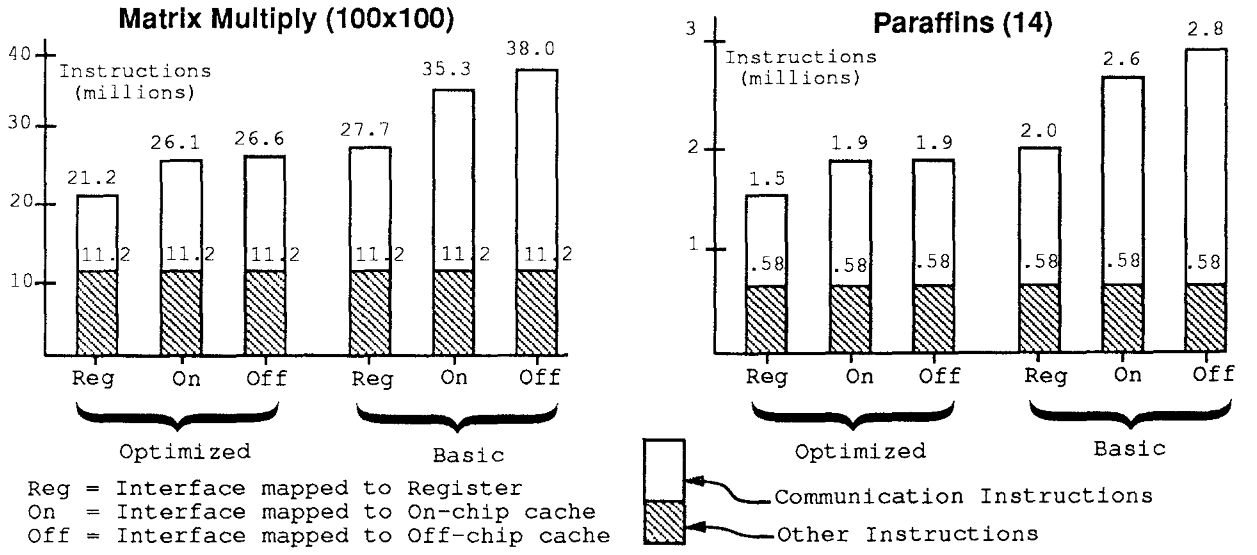


Figure 1: Dynamic instruction counts for 100 by 100 matrix multiply and 14 paraffins using the six different network interface implementations.

executed in order to send, dispatch, and process messages. Although some of the instructions inside message handlers do perform useful work, such as memory allocation or queueing of deferred read requests, most of these instructions can be considered network interface overhead. The shaded, bottom component corresponds to the remaining instructions, ones which are not involved in communication.

The data in Figure 1 leads to several insights. First, communication has a first-order effect on the these fine-grain parallel programs. Although the dynamic frequency of executing a high-level message sending instruction is relatively low, 9% and 11%, more than half of all the RISC instructions are dedicated to communication. In addition, hardware optimizations of the network interface appear more important than the actual placement of the interface. Even existing processors could considerably lower their communication costs by attaching an optimized interface, such as NIC, on their external cache bus. Most importantly, the gains achieved by using a register-based, hardware assisted network interface are substantial. The cost of communication decreases, on average, by a factor of two and one half as we optimize the network interface and incorporate it into the register file.

References

[AHN88] Arvind, S. Heller, and R. S. Nikhil. Programming Generality and Parallel Computers. In *Proceedings of the 4th International Symposium on Biological andArtificial Intelligence Systems*, September 1988.

[CSS+91] D. Culler, A. Sah, K. Schauser, T. von Eicken, and J. Wawrzynek. Fine Grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991.

[HJ91a] Dana S. Henry and Christopher F. Joerg. The Network Interface Chip. Technical Report CSG Memo 331, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge MA 02139, USA, June 1991.

[HJ91b] Dana S. Henry and Christopher F. Joerg. The Network Interface Chip. In *Proceedings of the 1991 MIT Student Workshop on VLSI and Parallel Systems*, pages 12–1,12–2, July 1991.

[HJ92] Dana S. Henry and Christopher F. Joerg. A Tightly-Coupled Processor-Network Interface. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1992.

[Nik90] R.S. Nikhil. Id Version 90.0 Reference Manual. Technical Report CSG Memo 284-1, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge MA 02139, USA, September 1990.

AIDA: Data Structure Optimization on a MIMD Parallel Computer

Waldemar Horwat
Concurrent VLSI Architecture Group
waldemar@ai.mit.edu NE43-630

Introduction

An important impediment to writing efficient parallel software is the difficulty of crafting the code to efficiently map data and control structures to a computer's architecture in order to exploit locality or reduce contention for critical resources such as storage or network bandwidth. The best implementation of a module may differ depending on hardware architecture, the module's interface to other modules in the program, and the resources available when it is used, thus making writing general-purpose libraries difficult. Much of this information is difficult to predict when the program is being written, so the programmer often does not have the information necessary to decide which is the best representation for a program structure.

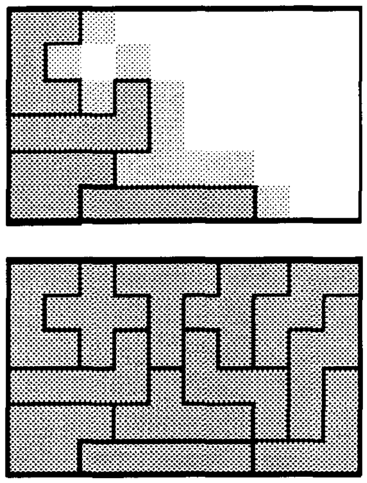
AIDA (Accelerated Implementations of Data Abstractions) is a new language and environment that allows the programmer to specify alternative implementations of a module and provides the computer with considerable latitude about choosing implementations based on compile-time and run-time information. The language permits a range of representation annotations and hints for the compiler. A major goal of this work is the definition of *efficient abstraction mechanisms* for MIMD parallel computers—one should be able to define general abstractions such as a generic sort routine and have the system customize their data and control representations to best fit their usages.

Some methods of choosing sequential data representations have been explored before in SETL [7], LIBRA [3], [6], and [5]. Whereas these systems aimed at making programming easier with some loss of efficiency, AIDA's emphasis is on letting programmers generate *more efficient* modular programs than they could using standard techniques. Parallel programming is more difficult than sequential, and the wider information gap between what the computer can determine about a parallel program and what the programmer knows provides an opportunity for representation optimizations. Unlike FORTRAN optimizers such as [4], AIDA is focused on optimizing complex data structures and symbolic code.

Example

The Pentomino program (see the figure) illustrates several kinds of data structure choices. There are several important data and control structures in this program which can have multiple representations:

- The twelve pieces and their orientations: lists of (x,y) square coordinates or 60-bit bitmaps
- A board with partially placed pieces: a 6×10 array of piece numbers or, for some uses, simply a 60-bit bitmap. The border (marked with light gray) is especially interesting in guiding the search, but it is a bulky data structure (including some information not shown in the figure). When expanding a placement, the system has a choice of rebuilding the border data structure or incrementally modifying the existing one; which one depends on implementation details such as whether the process to search the new position is local or remote and how expensive it is to ship the border data structure to it.
- The search tree. This control/data structure is created and destroyed as the search proceeds. The search must switch from breadth-first at the higher levels to depth-first at the lower levels to avoid exhausting memory.



Pentominoes and one solution

The pentomino program finds all 2339 possible arrangements of the twelve pentominoes in a 6×10 rectangle. The pentominoes can be rotated or flipped over.

The pentomino solver does an exhaustive depth-first search with pruning, placing pieces one at a time. At any particular time the search picks a square along the boundary (light gray) of the already placed pieces and expands the search tree one level by trying to fit each of the remaining pieces so that it covers that square. The search uses a heuristic that tries to pick the boundary square with the smallest number of matches; if there is a boundary square with no matches, that search branch can be abandoned immediately.

This research is supervised by W. Dally and supported by an ONR fellowship, by DARPA under contracts N00014-88K-0738 and N00014-91-J-1698, by Air Force Systems under contract F19628-92-C-0045, and by a NSF Presidential Young

The AIDA Language

AIDA is based on a statically typed version of Concurrent Smalltalk [2], an imperative parallel programming language based on object-oriented programming and futures. AIDA is highlighted by the following constructs:

- The `(choose` *alt1 alt2 ... altn*`)` statement lets the compiler or run-time system choose, at its discretion, which of the alternative statements to execute. This statement is useful when there are several ways to perform a function, none of which is clearly superior when the program is written. *alt1* and *alt2* could, for instance, be two sort algorithms, one of which is more appropriate for tightly localized data, the other of which is better for data spread throughout a parallel computer. `choose` statements can be linked and annotated to make several choices dependent on one another.
- Multiple representations of a data type. A data type such as a set or array can have multiple representations. The compiler and run-time system choose a particular representation, for which the programmer can provide hints (stating a preference for a particular implementation, limiting the size of an array, etc.).
- Variants of a representation of a data type. Some data structure variables can be declared optional; the system will drop them if they are not needed (this interacts well with `choose` statements which can either access these instance variables or calculate the values in some other manner if the data structure variables are not available).
- Transformers and coercers that convert one object representation into another at run-time. For instance, one representation of a set can be transparently transformed into another, and a general array variant can be coerced into a fixed-length one. The `(transform` *obj*`)` statement provides a hint to the system that it might be worthwhile to revise the representation of *obj*'s data at this point in the program.
- Annotations for optimizing hierarchical combination of data structures (inclusion, local or remote pointers, etc.) and for passing arguments to functions.
- Annotations for optional data-driven synchronization. Rather than using return values to signify that a function is done, control flow can be synchronized using auxiliary counters or presence bits. This avoids the need to create, save, restore, and synchronize on contexts, which can be the most expensive operations on fine-grained parallel computers [2].

AIDA is designed to efficiently support both sequential machines and MIMD parallel computers such as the J-Machine [1]. The code generated for them will be very different, and the programmer can use the `choose` and data representation facilities to specify both sequential and parallel algorithms in cases where the best ones differ. The system will decide whether to bring all data to one node and use the sequential algorithm or whether to run the algorithm in parallel; such decisions often depend on what the rest of the program is doing.

AIDA makes choices by using compiler inference, heuristics, and programmer hints where possible to determine the sizes of data structures and operations performed on them. Where a choice cannot be made at compile-time, AIDA tries the various alternatives at run-time and collects statistics on them to make the choices.

At this time we are in the process of implementing AIDA with a J-Machine as the target architecture.

Bibliography

[1] William J. Dally et al. "The Message-Driven Processor: A Multicomputer Processing Node with Efficient Mechanisms." *IEEE Micro*, 12:2, April 1992, pp. 23-39.

[2] Waldemar Horwat. *Concurrent Smalltalk on the Message-Driven Processor.* MIT Artificial Intelligence Laboratory Technical Report 1321, September 1991.

[3] Elaine Kant. "On the Efficient Synthesis of Efficient Programs." *Readings in Artificial Intelligence and Software Engineering*, Charles Rich and Richard C. Waters, ed. Morgan Kaufmann, 1986, pp. 157-183.

[4] Kathleen Knobe, Joan D. Lukas, and Guy L. Steele, Jr. "Data Optimization: Allocation of Arrays to Reduce Communication on SIMD Machines." *Journal of Parallel and Distributed Computing*, vol. 8, 1990, pp. 102-118.

[5] James R. Low. "Automatic Data Structure Selection: An Example and Overview." *Communications of the ACM*, 21:5, May 1978, pp. 376-385.

[6] Lawrence A. Rowe and Fred M. Tonge. "Automating the Selection of Implementation Structures." *Readings in Artificial Intelligence and Software Engineering*, Charles Rich and Richard C. Waters, ed. Morgan Kaufmann, 1986, pp. 245-257.

[7] J. T. Schwartz, R. B. K. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets: An Introduction to SETL.* Springer-Verlag, 1986.

Investigator Award, grant MIP-8657531 with matching funds from General Electric Corporation, IBM Corporation, and AT&T.

Computation Migration in Parallel Systems

Wilson Hsieh*
Large-Scale Parallel Software Group, MIT LCS
e-mail: wchsieh@lcs.mit.edu

1 Introduction

We describe a language feature with which a programmer can control the location of computation in a parallel object-oriented language; we hope that this will lead us to develop automatic methods for deciding where computation should occur. The *Prelude* language [1], a parallel object-based language, provides instance methods, class methods, and free-standing procedures; the execution of class methods and procedures (we shall refer to both as procedures) is not tied to the location of any object. The common paradigm is to have a procedure execute at a single location; when a procedure makes an instance method call, the call occurs at the object that it is invoked upon, and control then returns to the procedure.

We provide the programmer with an annotation for instance method invocations (the annotation occurs at the point of call) that specifies that the calling procedure migrates with the instance method call; in other words, when a procedure invokes the specified instance method, it then finishes executing where the method executes. We call this "computation-migration" (or "continuation-passing") because the continuation that represents the "rest of the procedure" is passed along with the instance method call. Computation migration can be viewed as a generalization of tail recursion; a tail-recursive call consists of passing a continuation at the last call within a procedure.

Computation migration saves messages, as illustrated in Figure 1. A procedure on processor 1 calls two instance methods on objects on processor 2, the second of which calls an instance method on processor 3; the procedure then calls an instance method on processor 3. Without computation migration, this sequence of calls takes eight messages: four call/return pairs. With computation migration, the sequence of calls takes only four messages. This has several effects: it may reduce the load on the network (depending on the size of a continuation message); and it should reduce the overall latency of the procedure, since less time is spent handling message interrupts and there are fewer network transit times in its execution path.

There is a tradeoff involved in deciding when to make a call using computation migration; not every call should use it, for several reasons. A continuation message will tend to be larger than a simple call message, since the state of the procedure must be sent; using computation migration for all calls could increase rather than decrease the network load. In addition, it would be inefficient to migrate a procedure to a processor that is heavily loaded.

2 Alternatives

It is possible to explicitly code a continuation-passing structure in some languages (for instance, in Concurrent Aggregates [2]). However, this would be done by adding procedures that represent the execution of a continuation. For example, consider a procedure *p*, where *p* calls *x.foo* with computation migration. In order for the programmer to achieve the specified behavior, he would have to add a method *x.foo2* that performs *x.foo* and then executes the rest of *p*. This can required substantial amounts of complex code. It also breaks abstraction boundaries, as the design of an object's interface must take into account any computation migration that the object may be involved in. Finally, without tail recursion some extra messages would still be required.

3 Implementation

We would like to implement computation migration as a manipulation of stack frames at runtime: the appropriate data would be passed to the destination processor, where the stack would be set up; the call would be

*Supervised by Professor William E. Weihl. Supported in part by the National Science Foundation under Grant CCR-8716884, by the Defense Advanced Research Projects Agency (DARPA) under Contract N00014-89-J-1988, by an equipment grant from Digital Equipment Corporation.

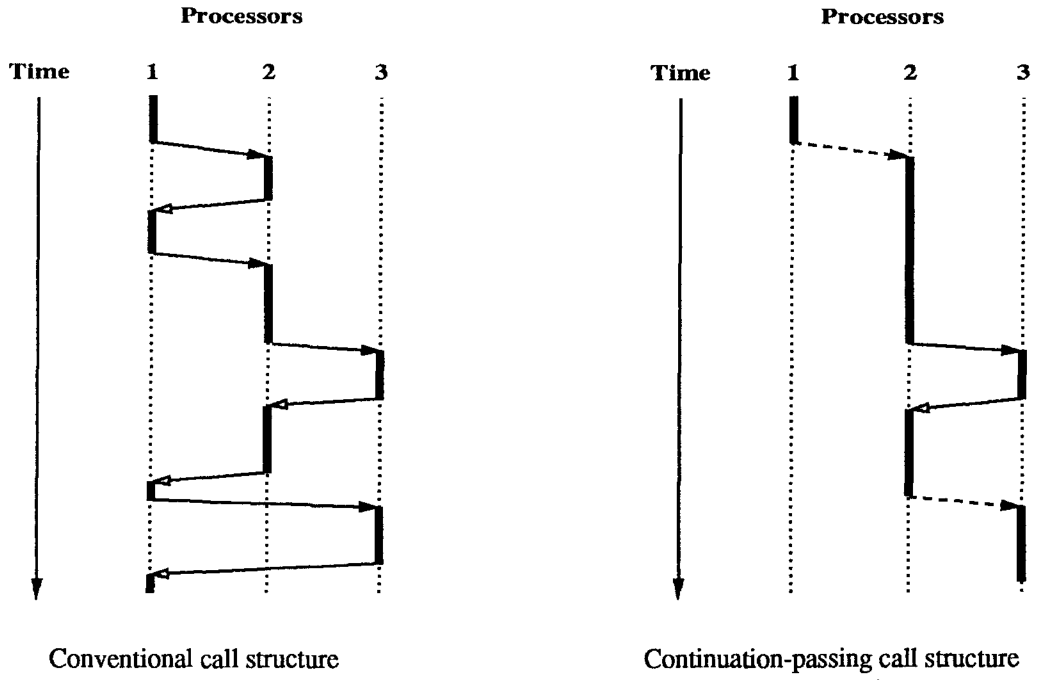

Processors
Time
1
2
3
Conventional call structure
Processors
Time
1
2
3
Continuation-passing call structure

Figure 1: The picture to the left shows a sequence of calls made without computation migration; the picture to the right shows the same sequence of calls with computation migration used twice. Solid arrowheads indicate instance method calls; blank arrowheads indicate returns from those calls; a dashed arrow indicates a continuation-call.

executed by jumping into the code for the procedure. However, the current Prelude compiler produces C code, so we do not have sufficient control of the actual code generation. Our current implementation of computation migration is thus handled by creating a special continuation procedure for each continuation call; instead of using the same code for the continuation, we execute this new procedure at the destination, where the arguments of the procedure are the live variables at the point of the continuation call. Although this solution is inefficient in terms of code space, it allows us to measure the performance gains of computation migration. Preliminary results show that using computation migration can dramatically improve performance.

4 Conclusions

Our current design and implementation only handles a simple form of migration: the "rest of the procedure" can move to another processor — a single stack frame moves. We are also investigating annotations that will allow the programmer to move partial frames (execute part of a procedure remotely, and then return) or multiple frames. Finally, when we gain more experience using computation migration, we will investigate having the compiler decide when to use computation migration.

References

[1] William Weihl, Eric Brewer, Adrian Colbrook, Chrysanthos Dellarocas, Wilson Hsieh, Anthony Joseph, Carl Waldspurger, and Paul Wang, "PRELUDE: A System for Portable Parallel Software." MIT Laboratory for Computer Science, MIT/LCS/TR-519, October 1991.

[2] Andrew Andai Chien, "Concurrent Aggregates (CA): An Object-Oriented Language for Fine-Grained Message-Passing Machines." MIT Artificial Intelligence Laboratory, AI-TR 1248, July 1990.

The Impact of Communication Locality on Large-Scale Multiprocessor Performance

Kirk L. Johnson (tuna@lcs.mit.edu)*
MIT Laboratory for Computer Science
545 Technology Square, Room NE43-635
Cambridge, Massachusetts 02139

1 Introduction

As multiprocessor sizes scale and computer architects turn to interconnection networks with non-uniform communication latencies, the lure of exploiting communication locality to increase performance becomes inevitable. Models that accurately quantify locality effects provide invaluable insight into the importance of exploiting locality as machine sizes and features change. In [4], we present and validate such a model. This abstract provides a brief overview of that modeling framework and presents two interesting results obtained thereby. First, one can show that exploiting communication locality provides gains which are at most linear in the factor by which average communication distance is reduced when the number of outstanding communication transactions per processor is bounded. Second, we obtain rough upper bounds on the performance improvement available on a particular architecture by exploiting locality to minimize communication distance.

2 What is Locality?

Applications often take advantage of *communication locality* to realize performance gains. Communication locality is a property of both applications and architectures. *Application locality* (or *algorithmic locality*) is that which is present in the organization of an application, independent of architectural details. *Architectural locality* represents the ability of an architecture to exploit application locality.

Two components contribute to application locality. The first, *temporal locality*, represents the effect of decreasing the communication frequency between application threads. Applications that minimize inter-thread communication by maximizing data reuse tend to exhibit good temporal locality. The second component, *physical locality*, represents the effect of affinity in the communication patterns amongst an application's threads. Applications tend to have good physical locality to the extent that their inter-thread communication graphs have relatively low bisection width and high diameter. An application in which all distinct pairs of threads communicate equally has no physical locality.

While the modeling framework discussed herein can be used to reason about both temporal and physical locality effects, this research focuses on the latter. Numerous researchers have demonstrated the importance of the former; compilation techniques for increasing temporal locality continue to be an active area of research [5, 6].

*The research reported on herein was supervised by Professor Anant Agarwal and funded in part by NSF grant # MIP-9012773 , in part by DARPA contract # N00014-87-K-0825, and in part by a NSF Presidential Young Investigator Award.

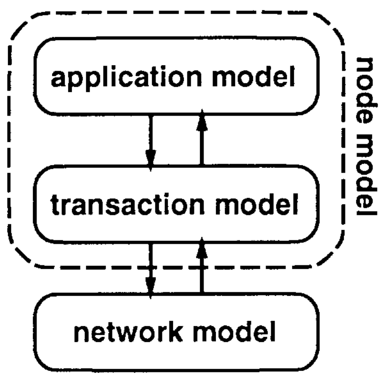


Figure 1: Combined modeling framework.

Multiprocessor systems built around interconnection networks with non-uniform communication latencies can exploit physical locality in applications by mapping application threads to processors such that average communication distances are lower than would result from mappings which ignore the locality available in the network.

3 A Framework for Modeling

This section provides a brief overview of the aforementioned modeling framework; a more detailed treatment can be found in [4].

The modeling framework (Figure 1) consists of three individual component models: an application model describes processor behavior in terms of abstract communication transactions, a transaction model describes the resources required to satisfy said communication transactions, and a network model characterizes the behavior of the underlying interconnection network. The application and transaction models are combined to obtain a node model which describes the behavior of individual multiprocessor nodes as seen by the interconnection network. The final combined model is obtained by joining the node and network models. These models are joined such that applications effectively receive feedback from the network and only inject messages at rates appropriate to the message latencies they actually observe.

One novel aspect of the modeling framework is the simplicity of the application and node models. Essentially, each model has only two parameters; these parameters correspond directly to computational grain size (computation-to-communication ratio) and latency sensitivity (ability to tolerate increases in communication latency). It is straightforward to show that this latency sensitivity parameter is sufficient to describe a wide range of latency hiding/tolerating techniques (e.g. multithreaded processors, relaxed memory consistency models, prefetching).

The network model used in this research is that for packet-switched k-ary n-dimensional torus networks with separate unidirectional channels in both mesh directions presented by Agarwal in [1]. This model assumes that messages are wormhole routed according to an e-cube routing scheme [3].

We obtain the combined model by using the node and network models to provide feedback to one another so that individual nodes "back off" as message latencies increase, injecting messages into the network at rates appropriate to the message latencies they actually observe. Combining the node and network models produces a polynomial quadratic in the average per-node message injection rate. This quadratic is easily solved to obtain the predicted message injection rate predicted. Other values of interest (e.g. channel utilization, average inter-transaction issue time) are obtained by substituting this predicted value into the appropriate model equations.

4 Bandwidth vs. Latency

As machine sizes scale, applications with little physical locality place increasing bandwidth demands on interconnection networks. Increases in application bandwidth requirements in turn cause contention effects to become more pronounced. Using the framework described above, one can demonstrate that under a reasonable set of assumptions about application and processor behavior, the impact of contention effects is bounded, even for very large machines and communication-intensive applications that induce heavy network loads.

Using the combined model, one can show that as machine sizes scale and average communication distance increases, the average time it takes a message to travel a single network hop (T_h) approaches a limiting value which depends only on average message size, latency sensitivity of the application and node models, and network dimension—*independent of machine size*. Intuitively, T_h approaches this limiting value because of the linkage between application and network behavior. If each node can only have some finite number of communication transactions outstanding, increasing transaction latencies cause transaction issue rates to fall. This negative feedback keeps processors from loading interconnection networks to a point where communication latencies become unbounded.

The fact that T_h approaches this limiting value implies that average communication latency is linear in communication distance. This, in turn, has a profound impact on the potential benefit of exploiting physical locality. Any gain due to exploiting physical locality is bounded by the degree by which communication latencies are reduced. Since communication latencies are linear in communication distance, reducing average communication distance by some factor x can only provide performance gains which are linear in x.

5 Exploiting Physical Locality

Intuition dictates that application performance should benefit from thread-to-processor mappings that reduce overall communication distance. The more physical locality present in an application, the greater the gains possible through reducing communication distances. Using the modeling framework discussed above, one can obtain rough upper bounds on the potential benefit of exploiting physical locality. While some benefit is available, it is somewhat less than one might initially expect. For an architecture like the MIT Alewife machine [2] organized as a two-dimensional torus, exploiting physical locality provides no more than a factor of two or so performance improvement for a 1,000 processor machine; with a million processors, the upper bound increases to roughly 50.

An examination of the factors leading to this less-than-expected impact indicates that it is primarily due to the relatively high ratio of communication bandwidth to computation speed in that architecture. In fact, using the modeling framework, the degree to which various factors (e.g. fixed communication overheads, useful work, etc.) contribute can be quantified. Such a breakdown allows identification of the phenomena that lead to the apparent disparity. Recomputing the gains for architectures with progressively slower networks confirms this fact showing that larger gains are possible when processors are faster relative to the speed of the interconnection network.

References

[1] Anant Agarwal. Limits on Interconnection Network Performance. *IEEE Transations on Parallel and Distributed Systems*, pages 398–412, October 1991.

[2] Anant Agarwal, David Chaiken, Kirk Johnson, David Kranz, John Kubiatowicz, Kiyoshi Kurihara, Beng-Hong Lim, Gino Maa, and Dan Nussbaum. The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor. Technical Report MIT/LCS/TM-454, MIT Laboratory for Computer Science, June 1991.

[3] William J. Dally. Performance Analysis of *k*-ary *n*-cube Interconnection Networks. *IEEE Transations on Computers*, pages 775–785, June 1990.

[4] Kirk L. Johnson. The Impact of Communication Locality on Large-Scale Multiprocessor Performance. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 392–402, May 1992.

[5] G. N. S. Prasanna. Structure Driven Multiprocessor Compilation of Numeric Problems. Technical Report MIT/LCS/TR-502, MIT Laboratory for Computer Science, April 1991.

[6] Michael E. Wolf and Monica S. Lam. A Data Locality Optimizing Algorithm. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 30–44, June 1991.

On the Second Eigenvalue and Linear Expansion of Regular Graphs

Nabil Kahale [*]
MIT Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139
kahale@theory.lcs.mit.edu

1 Main Results

Given an undirected k-regular graph $G = (V, E)$ and a subset X of V, we define the expansion of X to be the ratio $\frac{|N_G(X)|}{|X|}$, where $N_G(X) = \{w \in V : \exists v \in X,\ (v,w) \in E\}$ is the set of neighbors of X. Graphs whose all subsets of size lying in a given range have large expansion are called expanders graphs.

Expander graphs are widely used in Computer Science, in areas ranging from parallel and distributed computation to complexity theory and cryptography. The range of the subsets whose expansion is relevant and the magnitude of the expansion needed depends on the nature of the application. For example, in the design of the AKS sorting circuit, we need expanders of fixed degree and such that subsets of size $\epsilon|V|$ have expansion at least $\frac{1-\epsilon}{\epsilon}$, where ϵ is a small fixed positive constant. The depth of the resulting circuit is proportional to the degree of the expander. In other applications, like the construction of non-blocking networks, we need a family of fixed degree bipartite expanders where the expansion of linear-sized subsets is at least $\frac{k}{2}$. Indeed, this guarantees that a constant fraction of any small subset have *unique neighbors*.

It is not hard to show that random k-regular graphs, are good expanders: their expansion coefficient is $k - 1 - \epsilon$, where ϵ is an arbitrarily small positive constant. However, the naive and only known method to calculate the exact expansion coefficient of a graph takes an exponential amount of time.

The best known technique to calculate lower bounds on the expansion in polynomial time relies on analysing the second eigenvalue of the graph. The smaller the second eigenvalue, the higher expansion we get. This technique shows that random regular graphs have provable expansion at least $k/4$. It also shows that Ramanujan graphs, which been constructed explicitly by Lubotzky, Phillips and Sarnak in 1986 and independently by Margulis in 1987, have expansion at least $k/4$. Ramanujan graphs and random graphs are known to have optimal second eigenvalue (up to a $1 + o(1)$ factor). In a previous work, Kahale [1] improved the lower bound on the expansion of the LPS-M and random graphs to $\frac{3k}{8}(1-o(1))$. More recently, Kahale [2] improved this bound to $\frac{k}{2}(1-o(1))$. Moreover, he essentially showed that the $k/2$ bound is the best bound any technique based on the second eigenvalue can yield by exhibiting a family of graphs with asymptotically optimal second eigenvalue and linear expansion only $k/2$. As an application of the improved expansion of Ramanujan graphs, we can build explicit selection networks of asymptotic size $(3 + \epsilon)n \log_2 n$, for

[*]Supervised by Tom Leighton and Supported by the Defense Advanced Research Projects Agency under Contracts N00014-87-K-0825 and N00014-89-J-1988, the Air Force under Contract AFOSR-89-0271, and the Army under Contract DAAL-03-86-K-0171.

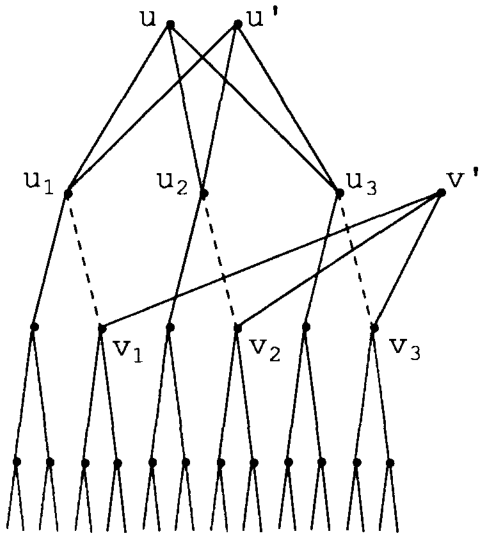


Figure 1: The graph G_{n+2} in the neighborhood of u in the case $k = 3$. The dotted edges are those belonging to $E - E'$.

any $\epsilon > 0$, improving upon the bound $6n \log_2 n$ that was previously known. A selection network is a network of comparators that classifies a set of n numbers, where n is even, into two subsets of $n/2$ numbers such that any element in the first set is smaller than any element in the second set.

2 A family of k-regular graphs with asymptotically optimal second eigenvalue and expansion $k/2$

In the following, we explicitly construct such a family. From Margulis and LPS, we know that we can explicitly construct an infinite family of Ramanujan graphs H_n on n vertices whose girth is at least $(4/3 + o(1)) \log_{k-1} n$. The *girth* of a graph is the length of its shortest cycle. Let $H_n = (V, E)$ be an element of the family and $u \in V$ be a vertex of H_n. Since the girth of H_n is large, the graph H_n looks like a regular tree in the neighborhood of u. Let $u_1, \ldots, u_k$ be the neighbors of u and let $v_1, \ldots, v_k$ be k vertices distinct from u and such that $(u_i, v_i) \in E$. Consider the k-regular graph $G_{n+2} = (V', E')$, where u', v' are external vertices, $V' = V \cup \{u', v'\}$ and $E' = E \cup \{\{u', u_1\}, \ldots, \{u', u_k\}\} \cup \{\{v', v_1\}, \ldots, \{v', v_k\}\} - \{\{u_1, v_1\}, \ldots, \{u_k, v_k\}\}$. Figure 1 shows the graph G_{n+2} in the neighborhood of u in the case $k = 3$. In [1], we show that the second largest eigenvalue of the graphs (G_n) is $2\sqrt{k-1} + o(1)$, and so it is asymptotically optimal. As a consequence the linear expansion is at least $k/2$. On the other hand, the expansion of the subset $\{u, u'\}$ is clearly $k/2$, and so the linear expansion of the family (G_n) is equal to $k/2$.

References

[1] N. Kahale. Better expansion for ramanujan graphs. In *FOCS91*, pages 398–404.

[2] N. Kahale. On the second eigenvalue and linear expansion of regular graphs. To appear in *FOCS92*.

EFFICIENT TECHNIQUES FOR INDUCTANCE EXTRACTION OF COMPLEX 3-D GEOMETRIES*

MATTAN KAMON†

This abstract describes combining a mesh analysis equation formulation technique with the GMRES matrix solution algorithm to accelerate the determination of inductances of complex three-dimensional structures. Results from FASTHENRY, our 3-D inductance extraction program, demonstrates that the method is more than an order of magnitude faster than the standard solution techniques for large problems [1].

Inductance extraction involves the determination of the $c \times c$ frequency dependent impedance matrix, Z_c, where c is the number of conductors. From the impedance matrix, the resistance and inductance matrices are easily extracted. One approach to computing the frequency dependent impedance matrix associated with the terminal behavior of a collection of conductors involves first approximating each conductor with a set of piecewise straight conducting sections. The volume of each straight section is then discretized into a collection of parallel thin filaments through which current is assumed to flow uniformly [2, 3]. The interconnection of these current filaments can be represented with a planar circuit, where the n nodes in the circuit are associated with connection points between conductor sections, and the b branches in the circuit represent the current filaments into which each conductor section is discretized. The system is assumed to be in sinusoidal steady-state.

Determining column i of Z_c involves determining the terminal voltages that result from setting the current in conductor i to 1 and the rest to 0. To determine these voltages, one must 'solve' the circuit described above. To begin, since each of the filaments, or branches, can be approximated by infinitely thin straight wires, one can compute directly the branch impedance matrix Z_b to give

$$V_b = Z_b I_b \tag{1}$$

where V_b is the vector of voltages across each branch and I_b is the vector of branch currents. The usual approach is then to apply Kirchoff's current law and force the sum of the current to be zero at each node. This set of equations can be written as

$$A I_b = I_s \qquad A^t V_n = V_b \tag{2}$$

where I_s is the vector of source currents into each node, A is called the incidence matrix, and V_n is the vector of reference node voltages. These equations can be combined to give

$$A Z_b^{-1} A^t V_n = I_s. \tag{3}$$

The right-hand side, I_s is known and is mostly zeros except for the nodes corresponding to the conductor carrying a current of 1. To determine a column of the final impedance matrix, Z_c, we need only solve for V_n and extract the appropriate voltages.

In most programs, the dense matrix problem in (3) is solved with some form of Gaussian elimination, and this implies that the calculation grows as b^3. For complicated packaging structures, b can exceed ten thousand, and solving (3) with Gaussian elimination can take days, even using a high performance scientific workstation.

The approach to calculating the frequency dependent inductance and resistance matrix described above has some disadvantages if (3) is to be solved with an iterative method. It is difficult to apply the iterative method, because the matrix $A Z_b^{-1} A^t$ contains Z_b^{-1}, which can only be computed by forming the dense matrix Z_b, and then somehow inverting it.

Another approach to generating a system of equations for the currents and voltages in the network representing the conductor system discretization is to use Kirchoff's voltage law, or mesh analysis. KVL implies that the sum of branch voltages around each loop in the planar circuit must be zero. These equations can be represented as

$$M V_b = V_s \qquad M_t I_m = I_b \tag{4}$$

where V_s is the vector of source voltages inside each loop and I_m is the vector of mesh currents. These yield

$$M Z_b M^t I_m = V_s. \tag{5}$$

In this case, in order to find column i of the final admittance matrix, $Y_c = Z_c^{-1}$, one must solve for I_m given V_s. This time, V_s will be 1 volt for the mesh corresponding to all of conductor i and 0 volts for all other meshes.

Notice that (5) does not involve Z_b^{-1}, so to speed up the computation, FASTHENRY uses the conjugate-residual style iterative method, GMRES [4]. Such methods have the general form shown below in Algorithm 1 for solving $Ax = b$.

* This work was supervised by Prof. Jacob K. White and supported by DARPA contract N00014-91-J-1698, an NSF Fellowship, and grants from IBM and Digital Equipment Corporation.

† MIT Room 36-888, E-mail: *matt@rle-vlsi.mit.edu*

Filaments per conductor section	Size of MZM^t (m)	Solution time, direct inversion	Solution time, preconditioned GMRES
1	35	0.0003	0.007
2	210	0.339	0.147
4	560	8.02	1.08
6	910	35.9	3.08
9	1435	135	7.85
12	1960	344	14.4

TABLE 1
Execution time comparison for the 35-pin package example. Execution times are in IBM RS6000/540 CPU minutes.

ALGORITHM 1 (GMRES ALGORITHM FOR SOLVING $Ax = b$).

```
guess $x^0$
for $k = 0, 1, \ldots$ until converged {
  Compute the error, $r^k = b - Ax^k$
  Find $x^{k+1}$ to minimize $r^{k+1}$
    based on $x^i$ and $r^i$, $i = 0, \ldots, k$
}
```

The iterative algorithm can be accelerated by multiplying both sides of (5) by a preconditioner, which is a good approximation to $(MZ_bM^t)^{-1}$. For FASTHENRY, this preconditioner is formed by directly inverting block diagonals of MZ_bM^t and using those to form a block diagonal matrix as the preconditioner. Each block is chosen to correspond to only the meshes for a given conductor.

For a sample pin package from Digital Equipment Corporation (Figure 1), FASTHENRY with the preconditioned iterative algorithm proved much faster than direct inversion (See Table 1). As expected, the solution time for direct inversion grew with m^3 but preconditioned GMRES grew only as m^2. For this small problem with only twelve filaments per section, the iterative algorithm is already more than 23 times faster than direct inversion.

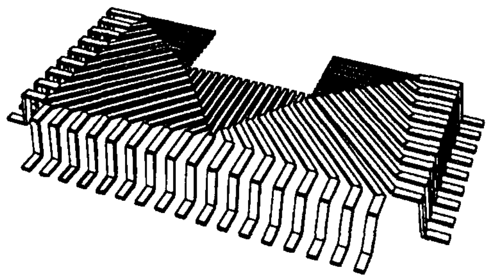


FIG. 1. *Half of a pin-connect structure. Thirty-five pins shown.*

Future work using multipole algorithms will exploit the fact that the off-diagonal elements of Z_b are the partial inductances generated from integrals of $\frac{1}{R}$ [5]. Such methods will avoid forming and storing most of the entries in the dense matrix MZ_bM^t, and reduce the cost of calculating matrix-vector products required for the GMRES procedure to order b operations.

REFERENCES

[1] M. Kamon, M. J. Tsuk, and J. White, "Efficient Techniques for Inductance Extraction of Complex 3-D Geometries," *Proceedings of the Int. Conf. on Comp. Aided Design*, November 1992, to appear.

[2] W. T. Weeks, L. L. Wu, M. F. McAllister, and A. Singh, "Resistive and inductive skin effect in rectangular conductors," *IBM Journal of Res. and Develop.*, vol. 23, pp. 652–660, November 1979.

[3] A. E. Ruehli, "Survey of computer-aided electrical analysis of integrated circuit interconnections," *IBM Journal of Research and Development*, vol. 23, pp. 626–639, November 1979.

[4] Y. Saad and M. H. Schultz, "GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems," *SIAM Journal on Scientific and Statistical Computing*, vol. 7, pp. 856–869, July 1986.

[5] L. Greengard and V. Rokhlin, "A fast algorithm for particle simulations," *Journal of Computational Physics*, vol. 73, pp. 325-348, December 1987.

Logged Commit Dependencies for Highly Concurrent Databases

John S. Keen[1]
MIT AI Lab, NE43-614
johnk@ai.mit.edu

June 12, 1992

Concurrent computers become increasingly attractive for transaction processing applications as throughput requirements continue to increase. Parallelism must be incorporated in all aspects of database management system (DBMS) design. In particular, logging information should be distributed amongst several disk drives, lest a single disk drive constitute a serial bottleneck.

"Hot spot" objects are items in the database which are frequently updated. To ensure that hot spot objects do not limit the maximum throughput for the entire system, a DBMS must offer high throughput on each object. A transaction must first acquire an exclusive lock on an object before it can update it. This lock serializes modifications by independent transactions, but limits the rate at which successive transactions can access the object.

When a transaction has finished its work and wants to commit its updates, it makes a request to the DBMS; this request is called a *precommit*. The DBMS appends a *COMMIT* record for the transaction to the log. The DBMS waits until all log records have been written to nonvolatile disk storage before it finally commits the transaction. A simple DBMS does not allow a transaction to release any locks until after it commits, but this limits throughput on any object to the rate at which blocks can be written to disk.

Previous researchers have proposed the *precommitted transaction* technique so that disk I/O does not limit throughput on hot spot objects. A transaction *t1*'s locks are all released immediately after the transaction precommits. A subsequent transaction *t2* can see *t1*'s updates even though *t1* has not yet committed, in which case it becomes *dependent* on *t1* to eventually commit. The DBMS cannot commit *t2* until after it has committed *t1*.

When log records are serially ordered in a single log stream, it is not difficult to ensure that *t1* commits before *t2* because *t1*'s log records will be written to disk before those of *t2*. In a highly parallel setting, the DBMS may direct the log records from transactions *t1* and *t2* to different log streams. Unless the DBMS regulates the order in which blocks in different log streams are written to disk, it is possible for all *t2*'s log records to be written to disk before those of *t1*. Recovery after a crash might incorrectly restore the updates by *t2* while annulling those of *t1*.

To prevent such anomalies, previous researchers have proposed that the DBMS regulate the order in which *COMMIT* log records are written to disk. This solution is awkward because it introduces dependencies amongst log streams which would otherwise be independent. Furthermore, a cyclic dependency amongst log streams would jeopardize the consistency of the log information on disk, and so the DBMS must prevent the formation of cycles.

[1]Faculty Supervisor: William J. Dally

This research was supported in part by an NSERC 1967 Postgraduate Scholarship, by the Defense Advanced Research Projects Agency under contracts N00014-88K-0738 and N00014-91-J-1698 and in part by a National Science Foundation Presidential Young Investigator Award, grant MIP-8657531, with matching funds from General Electric Corporation, IBM Corporation and AT&T.

An alternative approach [1] is to explicitly record dependency information in the log, so that it is unnecessary to regulate the order in which records are written to disk at different streams. When a transaction *t* precommits, a *PRECOMMIT* record is immediately directed to some log stream. The *PRECOMMIT* identifies any previous transactions on which *t* depends; it also identifies data objects (that were updated by *t*) for which the associated log records have not yet been written to disk. The *PRECOMMIT* record can be written to disk at any time. After a crash, a recovery program can examine the dependency information in *PRECOMMIT* records to determine which transactions actually committed (i.e., had all dependencies satisfied) prior to the crash. To make recovery more efficient, a separate *COMMIT* log record is written for each transaction after it finally does commit.

Figure 1 illustrates an example in which there are four log streams. Data log records, which record updates to objects in the database, are directed to the top two log streams. Transaction log records record when transactions begin, precommit, commit and abort; they are directed to the bottom two log streams.

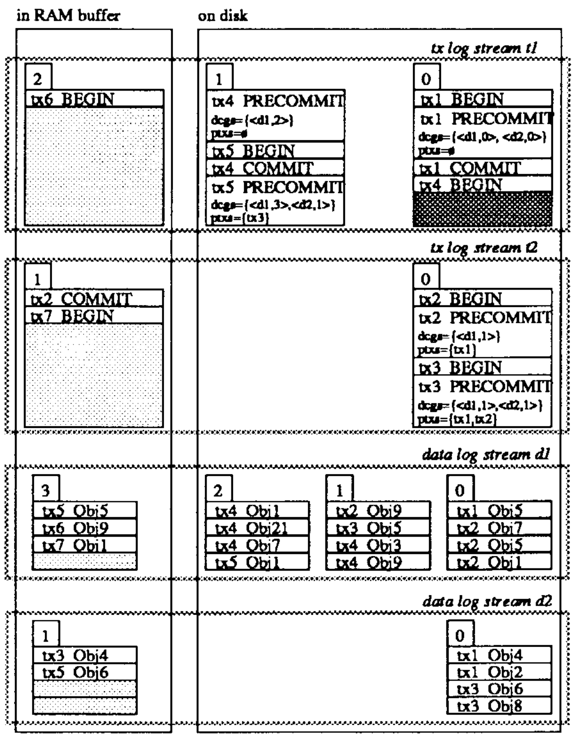


Figure 1: Explicit Representation of Dependencies for Transactions

Preliminary analyses confirm the feasibility of this technique. Experimentation and evaluation of a working implementation on the J-Machine are goals for future work.

References

[1] John S. Keen. Logging and Recovery in a Highly Concurrent Stable Object Store. Technical Report CVA Memo #37, MIT, May 1991. *Revised November, 1991.*

Monte Carlo Radiation Transport Simulation for Benchmarking Intel's Touchstone Delta Machine

Thomas J. Klemas[1]
tjk@abp.lcs.mit.edu
MIT Lab for Computer Science

Abstract

As science and technology advance, researchers are faced with larger and more challenging problems. Often the number of calculations required to solve these problems places great demands on computing resources and can require greater computational power than currently exists. Thus, when new systems become available, it is important for researchers to have some measure of the performance, cost effectiveness, and usefulness of these machines. Benchmark codes provide a means of testing a machine's performance on certain applications.

One application which places great demands on the performance of a computer is the simulation of radiation transport. This project involves writing a Monte Carlo simulation of radiation transport for benchmarking Intel's iPSC2, iPSC860, and Touchstone Delta Machine. Since a significant number of Los Alamos National Laboratory researchers use their computers to perform tasks that are very similar to those performed by the benchmark radiation transport code, the results of this project will help those scientists to evaluate the usefulness of Intel's line of parallel supercomputers.

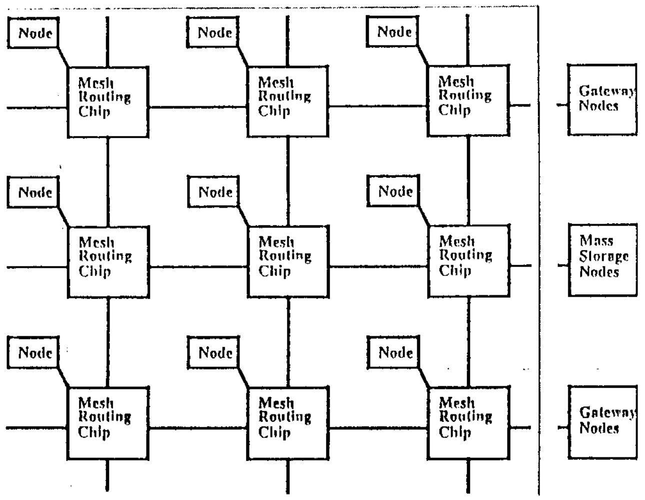
The simulation of radiation transport is important. Scientists and engineers are very interested in simulating particle transport within objects with which they are working. For example, such a simulator would be very useful to engineers involved in the design of a nuclear reactor. Through simulation of neutral particle transport one can determine statistics regarding energy, weight, velocity, position, and flux of particles with respect to position, as that particle travels through an object of an arbitrary geometry and composition. Simulation is performed using standard Monte Carlo techniques [1] based on probability distributions for the type and frequency of particle interactions. The amount of data used to calculate the probability distributions for the various media specified in a Monte Carlo radiation transport problem can be very large (up to 80 Megabytes).

Intel's iPSC2, iPSC860, and Touchstone Delta Machine are message passing computers. The Touchstone Delta [2] is the most recent of these machines and possesses 528 numeric nodes, each of which has up to 16 Megabytes of memory. (See figure 1.) Thus, the machine has a total memory of approximately 8.4 Gigabytes, and its peak speed is 42.2 Gigaflops for 32-bit floating-point operations. Message overhead is very large on all of these machines, and the ratio of processor speed of the nodes to the communication speed is highest on the Touchstone Delta. As a result, it is necessary to minimize communication between the nodes in order to approach to maximum performance. This restriction limits the types of problems that can take advantage of the computing power of this machine.

As mentioned above, some Monte Carlo radiation transport simulation problems can require up to 80 Megabytes of data. Since the node memory is only 16 Megabytes, some communication among the nodes is necessary. In order to develop an approach

[1]. Thesis Supervisor: Prof. Gregory M. Papadopoulos. This project was initiated and directly supervised by Mr. Olaf Lubeck, a research scientist at Los Alamos National Laboratory in the C3 Group. Partial funding for this work was provided by Los Alamos National Laboratory. MIT and Oakridge National Laboratory provide computing resources.

Touchstone Delta Machine



2 Dimensional Mesh Interconnection Network

16 x 33 Mesh of Numeric Nodes

528 Numeric Processors (i860's)

16 Megabytes of Memory on each Numeric Node

Figure 1.

that minimizes this communication, we devised several strategies based on the fact that the data can be divided according to energy levels; As a particle travels through a material it loses energy, and only the data for its current energy level is required to simulate its transport.

In the first approach the host of the parallel machine stores the entire data set in memory. The broadcast of the data set to the nodes is divided into phases according to energy ranges. Each node contains two buffers. One buffer is used in current calculations while the other is filled by the host. This masks message latency with the tracking of particles. Each node tracks all of its assigned particles until the energy level of all particles have dropped below the lower bound of the current data set energy range. In the meantime, the host broadcasts the data set for the next energy range to the alternate buffer at each node. This process continues until all particles have been absorbed.

The second approach creates a pipeline among several processors which each have one energy range of the data set. Particles are fed through the pipeline, passing to the next node when their energy level falls below the energy range of the current node. However, movement of particles through the pipeline is expensive because it requires communication.

The last major approach creates groups of processors that share data and stores a permanent energy range division of the data at each node as in the last method. However, each node also caches a current energy range of data in its memory. Nodes in a group would send requests to other nodes for a copy of required energy ranges and reply to similar requests for copies of their own permanent data.

Currently, implementation of the first approach is nearing completion. The coding challenge involved in each of these approaches is very similar. Thus, once one implementation is complete, others will follow quickly. Results of performance analysis will be compared to tests of versions of the Monte Carlo radiation transport simulation running on other machines, including an ID implementation on the Monsoon Dataflow machine and a Fortran version run on a Cray.

References

[1] L.L. Carter and E.D. Cashwell. Particle-Transport Simulation with the Monte Carlo Method. ERDA Critical Review Series, Technical Information Center, Office of Public Affairs U.S. Energy Research and Development Administration, 1975.

[2] Intel Supercomputers Division. A Touchstone Delta System Description, Intel Corporation, 1991.

Small-Depth Counting Networks

Michael Klugerman[*]
Mathematics Department
Massachusetts Institute of Technology
Cambridge, MA 02139
`klugermn@theory.lcs.mit.edu`

C. Greg Plaxton[†]
Department of Computer Science
University of Texas at Austin
Austin, TX 78712
`plaxton@cs.utexas.edu`

The notion of a "counting network" was recently introduced by Aspnes, Herlihy, and Shavit [1], where it was shown that such networks can be simulated efficiently on an asynchronous shared memory machine to implement counters, producer/consumer buffers, and synchronization barriers. The counting network provides a means for the processors of a parallel machine to obtain successive values from a counter. These values can then be used to obtain unique keys to various resources shared by the processors, to allocate tasks evenly among the processors, or to synchronize processors when necessary.

One solution to the counting problem is to use a single shared Fetch-and-Increment variable that is incremented each time a processor makes a request. This can lead to high memory contention when a large number of processors are making requests simultaneously. Counting networks provide a means by which this contention can be significantly reduced and thus allow for a much higher degree of concurrency. More specifically, a number of shared variables are used to implement a single counter in such a way that contention is reduced and a processor incrementing the counter need only access a small number of memory locations, thus providing fast response time and high throughput.

Counting networks are modelled after sorting networks [3] in that they are composed of 2-input 2-output components called balancers (rather than comparators). A balancer takes in tokens along both input wires and, acting like a toggle, outputs these tokens alternately along the top and bottom output wires (see Figure 1). As in the case of a comparator in a comparator network, balancers are used to construct a balancing network with an arbitrary number of input wires (and an equal number of output wires) called a balancing network. A counting network is a balancing network such that regardless of how many tokens are input on each input wire

1. The number of tokens output on one output wire is within one of the number of tokens output on any other output wire.

2. The number of tokens output on any output wire $\mathcal{W}$ is at least as great as the number output on any output wire located below $\mathcal{W}$.

The counting network can be implemented in software on a shared memory machine by associating a memory location with each balancer [1].

An important measure of the efficiency of a counting network is its depth. This is because the depth of the network is equal to the number of memory locations that a processor must access before its increment request has been fulfilled. In this paper, we present a number of constructions for counting networks of small depth.

[*]Supported by the Defense Advanced Research Projects Agency under Contracts N00014-87-K-825 and N00014-89-J-1988, the Air Force under Contract AFOSR-89-0271, and the Army under Contract DAAL-03-86-K-0171

[†]Supported by NSF Research Initiation Award CCR-9111591, and Texas Advanced Research Program (TARP) Award #003658480.

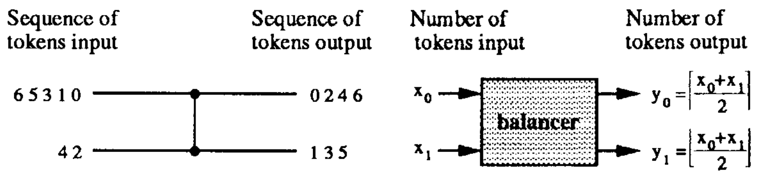

Sequence of tokens input
Sequence of tokens output
Number of tokens input
Number of tokens output
65310
42
0246
135
x_0
x_1
balancer
$y_0 = \lceil \frac{x_0+x_1}{2} \rceil$
$y_1 = \lfloor \frac{x_0+x_1}{2} \rfloor$

Figure 1: A balancer.

Aspnes, Herlihy, and Shavit [1] provide two $O(\lg^2 n)$-depth families of n-input counting networks by proving that the balancing network isomorphic to Batcher's bitonic sorting network [2] and isomorphic to the balanced periodic sorting network of Dowd, Perl, Rudolph, and Saks [4] are counting networks. Later, Klugerman [6] gave an $O(\lg n \lg\lg n)$-depth construction.

Our main result is a proof of the existence of an $O(\lg n)$-depth counting network where n is the number of input wires. This result answers the question posed in [1], which asks whether such an optimal-depth counting network exists. The technique used to obtain this result involves constructing a set of networks $\mathcal{N}^*$ such that for any fixed input sequence I, if a network $\mathcal{N}$ is chosen uniformly at random from $\mathcal{N}^*$, then $\mathcal{N}$ will count I with extremely high probability. "Good" networks are then chosen non-uniformly from $\mathcal{N}^*$ and are used to construct a deterministic counting network with logarithmic depth. The other result in this paper is an explicit construction of a counting network of depth $O(c^{\lg^* n} \lg n)$ (for some positive constant c), which represents an improvement over previously known constructions.

References

[1] J. Aspnes, M.P. Herlihy, and N. Shavit. Counting networks and multi-processor coordination. In *Proceedings of the 23rd Annual Symposium on Theory of Computing*, pages 348–358, May 1991.

[2] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the AFIPS Spring Joint Computer Conference,* vol. 32, pages 307–314, 1968.

[3] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, MA, 1990.

[4] M. Dowd, Y. Perl, M. Saks, and L. Rudolph. The balanced sorting network. Technical Report DCS–TR–127, Department of Computer Science, Rutgers University, June 1983.

[5] M. Klugerman and C. G. Plaxton. Small depth counting networks. In *Proceedings of the Twenty-Fourth Annual ACM Symposium on the Theory of Computing*, pages 417–428, 1992.

[6] M.R. Klugerman. Lecture 17: Counting networks. In F.T. Leighton, C.E. Leiserson, and N. Kahale, editors, *Research Seminar Series 15: Advanced Parallel and VLSI Computation*, pages 153–161. MIT Press, 1991.

Dynamic Alignment in SPMD Compilation[1]

Kathleen Knobe
kathyk@ai.mit.edu, NE43-630, 253-7710

1.0 Basic SPMD Compilation Strategy

The Single Program Multiple Data (SPMD) model of compilation is a straightforward approach suitable for a wide range of scientific applications written in Fortran targeted to massively parallel MIMD architectures. According to this model the data is distributed across the processors and all processors execute the same code, possibly following different control paths. The computation is generally an alternating sequence of local computation and interprocessor communication.

The basic SPMD compilation strategy is characterized by the following four rules:

1 - Alignment:
Directives specify alignments of objects.
2 - Scalars:
Scalars are "owned" by all processors.
3 - Control flow:
Every processor knows the global flow.
4 - Intermediate operations:
The "owner" of the LHS performs operation on the RHS.

Although these rules simplify compilation they place significant limits on performance. We show below how analyzing *where* objects should live (Section 2) and analyzing how they arrive there (Section 3) improves on these rules.

2.0 Alignment

This section discusses how compiler determination of the alignment of objects improves on the four SPMD compilation rules.

2.1 Alignment of Source Objects

Data optimization creates a graph among the textual occurrences of objects in the source. An edge between two occurrences implies a "preference" to align those occurrences. If such occurrences are not aligned, motion is required to align them.

Two types of preferences are required for objects in assignment statements:

[1]This research is supervised by W.J. Dally and supported in part by the Defense Advanced Research Projects Agency under contracts N00014-88K-0738 and N00014-91J-1698, by Air Force Systems under contract F19628-92-C-0045 and by a National Science Foundation Presidential Young Investigator Award, grant MIP-8657531, with matching funds from General Electric Corporation, IBM Corporation, and AT&T.

• the *identity preference* requires alignment of a definition with the use of the defined value (driven by data dependence analysis).

• the *conformance preference* requires alignment of an operation with its operands.

When preferences conflict, some preference cannot be honored, and the semantics must be maintained by communication. Since preferences are honored one by one in order of cost,[2] the communication that results from conflicts tends to be outside of loop nests.

Instead of performing alignment by directives as indicated by the SPMD rule 1, processing the preference graph performs alignment of source objects automatically.

2.2 Scalars

One possible result of conformance preference processing is called *dynamic alignment*, for example in `a(k) + b(j,k)` the vector `a` aligns with the `j`th row of the matrix `b`. As `j` changes, the alignment of `a` changes.

If a scalar is used in an expression with an array section in a loop, for example, `s + a(j)` within a loop on `j`, then `s` is dynamically aligned with respect to `j`. Based on this analysis, the scalar `s` is not required to live in all processors as it would in the SPMD approach based on rule 2. This analysis determines where each scalar will live.

2.3 Control Flow

In the code

```
if s then
   a(1:20) = b(1:20) + c
```

even if there are tens of thousands of processors, only those few involved in the assignment actually need to know the value of `s`. Optimization of the alignment of control variables to improve over the SPMD rule 3 is supported by an additional preference:

• the *control preference* requires alignment of an operation with the value that controls it (driven by control dependence analysis).

In the code above the control variable is scalar and the operation it controls has one dimension. The control variable may well be an array and may or may not be the same shape as the operation. If the control variable is not the same shape as the operation it controls it will be dynamically aligned.

[2]The cost used here is an estimate of the run time cost if communication is required to align data.

Control preferences are incorporated into the preference graph and are processed by the data optimizer with conformance and identity preferences.

2.4 Intermediate Operations

Consider the code fragment:

```
a(i,j,k) = b(i,j) + c(i)
```

within `i`, `j` and `k` loops. According to SPMD rule 4, for each iteration of the loop, the processor holding an element of `a` will receive an element of `b` and an element of `c`. Both the plus and the assignment are performed there. However, we can improve performance by allowing the location of the intermediate operation to be determined by the dynamic alignment that results from conformance preference processing. `c` is dynamically aligned with respect to `j` aligning with each column of `b`. The sum, a two dimensional object, is therefore computed in the location of its `b` operand. This sum is then dynamically aligned with respect to `k` for the assignment.

3.0 Communication

The analysis above determines where scalars, array sections, control variables and intermediate operations will live at each point in the program, but how and when they will arrive there is a separate question.

The communication required depends on how the object is used and on other code in the same loop nest. A dynamically aligned scalar, for example, will fall into one of the following communication categories:

- *privatized* - no communication. The value in each location is independent.
- *replicated* - parallel prefix communication. The value in each location is identical.
- *hopping* - one message from one processor for each iteration. The value in each location is determined from the value in the previous location.
- *scanned* - parallel prefix operation. The value in each location is computed by a simple associative operation on the value in the previous location.
- *implicitly distributed* - no communication. The scalar appears as the subscript of distributed arrays and is not explicitly available.

Dynamically aligned sections, whether they appear as explicit operands, as the result of intermediate operations or as control variables, also fall into the above communication categories.[3] The communication category for an intermediate is determined by the categories of its operands.

4.0 Example

Consider the example in Figure 1.

[3]Notice that a scalar appears to be a section when it is dynamically aligned.

```
do i =
 do j =
  do k =
    if control-expr(i,j,k) then
      c = ...
    a(i,j,k) = j * b(i,j) + c
   enddo
 enddo
enddo
```

Figure 1: Dynamic Alignment Example

In the SPMD model, the value of the control expression is communicated to all processors at each iteration. `c` is communicated to all processors each time it is redefined. `j` is communicated to all processors each time it is redefined. One value of `b` is moved to one element of `a` once for each iteration.

In the data optimization model, `c` is hopping with respect to all three loop indices since its value may or may not be redefined from it previous value in its previous location. Therefore `c` is communicated from one processor to another single processor at each iteration. `j` is implicitly distributed and requires no communication. `j * b(i,j)` is computed in the processors owning `b(i,j)` without communication. The product is be dynamically aligned with respect to `k`. This product is not modified and so is replicated with respect to `k`. The result of the plus is also dynamcially aligned with respect to `k` and since it can be computed only upon arrival of `c`, it is also considered hopping.

References

[1] Kathleen Knobe, Joan D. Lukas, and William J. Dally. Dynamic alignment on distributed memory systems. In *Proceedings of the Third Workshop on Compilers for Parallel Computers*, Vienna, Austria, July 1992. Austrian Center for Parallel Computation.

[2] Kathleen Knobe, Joan D. Lukas, and Steele, Guy L., Jr. Data optimization: Allocation of arrays to reduce communication on SIMD machines. *Journal of Parallel and Distributed Computing*, 8:102–118, 1990.

[3] Kathleen Knobe and Venkataraman Natarajan. Data optimization: Minimizing residual interprocessor data motion on SIMD machines. In *Frontiers '90: The Third Symposium on the Frontiers of Massively Parallel Computation*, College Park, Maryland, Oct 1990. University of Maryland.

The Anatomy of a Message Send

John Kubiatowicz*
NE43-629
kubitron@masala.lcs.mit.edu

1 Motivation

Researchers in parallel computing generally agree that it is important to support a *shared-address space* programming model, where programmers see a global space of data objects without having to worry about explicit data placement and code scheduling. Thus, much research has been geared toward implementing this model directly on shared-memory or message-passing hardware. Unfortunately, each has its disadvantages, as well as its advantages.

In a message passing architecture, each processor has its own private address space, so that a global address space must be synthesized by software which performs object location and renaming, and which explicitly dispatches messages to fetch remote data. This can be prohibitively expensive in the general case. However, in those cases when the compiler has sufficient information to manage data statically, it can bypass these software layers and take complete advantage of the direct, point-to-point messaging facilities which are directly supported by the hardware.

In contrast, shared-memory architectures support a global address space directly in hardware. Data location and renaming are performed directly in hardware, as is the launching of requests for remote data. However, scalability concerns require the introduction of non-uniform memory access latencies and caching to bolster system performance in the face of large network latencies. Caching, in turn, implies replication and a concomitant need for cache-coherence. The drawback to this approach is that *all* communication proceeds through reads and writes to shared memory; consequently, even communication which is explicitly characterized by the compiler or runtime system still suffers the overheads of cache-coherence.

Consequently, the MIT Alewife machine [1] provides hardware support for a shared-address space while at the same time supporting a message facility as efficient as those found in contemporary message-passing architectures. Alewife supports a shared-address space through a combination of hardware and low-level software, including a scalable cache-coherence mechanism [2]. The unique feature of Alewife is that the efficient message-passing mechanism needed to implement a shared-address space is made available to system software and user code via a simple interface. This permits compilers and runtime systems to bypass the shared-memory mechanism and use explicit messages when doing so is known to be more efficient.

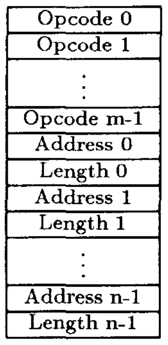


Figure 1: Packet Descriptor

2 Interface

Use of message-passing in a multiprocessor typically produces two classes of message traffic:

1. *Remote Procedure Invocation*, involving short messages with values that are derived from processor registers, and
2. *Block Data Transport*, involving the transfer of large blocks of data directly from memory at the source into memory at the remote.

The first arises from remote procedure calls, runtime-system management, and software-assisted dynamic cache-coherence[2]. Since these messages are quite short (of the order of two to sixteen words), their support requires an extremely efficient interface, both for the transfer of data from registers, and for the launching of the resulting message. The second type of message traffic arises during data and object distribution, block I/O, and software queuing of network messages. To support it efficiently, some form of direct-memory-access or DMA must be available.

*This work was conducted by the Alewife Research Group, supervised by Professor Anant Agarwal. The abstracted research was funded by DARPA contract #N00014-87-K-0825 and by by NSF grant #MIP-9012773.

Store 1	F	D	E_A	E_T	E_W	W						
Store 2		F	–	–	D	E_A	E_T	E_W	W			
Launch			F	–	–	–	–	D	E	W	Q_1	Q_2
Next Instruction						F	–	–	D	E	W	

Figure 2: Pipeline Diagram for Simple Message Launch

The Alewife machine handles both of the above through a single, low-overhead interface[3]. Its network interface permits messages to be sent through a two phase process: *describe* then *launch*. A message is described by writing directly to registers on the network-coprocessor, or *Communications and Memory-Management Unit* (CMMU). These writes proceed at the same speed as cached writes. As shown in Figure 1, the resulting descriptor can be up to 16 words long, and consists of a variable number of explicit *operands*, which will be placed at the head of the message, followed by a number of address-length pairs, describing data to be taken directly from memory and concatenated to the end of the packet. The first word of a packet must have a special format, the remaining words are software defined.

Once a packet has been described, it is then launched via an *atomic*, single-cycle, coprocessor instruction. The encoding of this instruction specifies both the number of explicit operands and the total length of the descriptor[1]. Both the user and supervisor are permitted to send messages, although user-code is prevented from launching "machine critical" messages[2]. To provide atomicity between user and system code, a *descriptor-length register* keeps track of the number of message-descriptor registers which have been written since the last message launch; interrupt code which must send messages can save and restore the user's descriptor.

For efficient reception of messages, the Alewife interface provides a 16-word, sliding window into the network input queue. On reception of a message, the CMMU interrupts the processor while making the first 16 words of the packet visible in the reception window. The processor can examine words within this window by reading coprocessor registers; as with the output interface, these reads complete at the speed of a cached memory access.

Once the processor has examined the packet, it can execute a special coprocessor storeback instruction to remove data from the window. User-code may dispose of user-generated messages. Encoded directly in storeback instruction are two separate fields. First, is the number of words to be simply discarded from the head of the window. Second, is the number of words (following those discarded) to be stored to memory via DMA. If this option is chosen, the processor must write the starting address for DMA to a special controller register before issuing the storeback instruction. Multiple storeback instructions can be issued for a single packet to scatter it to memory. Note that either of these storeback fields can contain a special "infinity" value which denotes "until the end of the packet".

[1]Note that the current implementation of the Alewife machine requires an even number of operands

[2]A set of message opcodes, including those used for cache-coherence, are reserved for use by the supervisor.

3 Implementation

Figure 2 shows the pipelining for a simple, two-operand message launch. Part of the latency here is the result of three-cycle stores in SPARC; a more aggressive processor design would complete each store in a single cycle. The end of the E stage for the launch instruction is the point at which the message is committed to the network. The W stage of the launch is required for the coprocessor interface, while Q_1 and Q_2 represent internal queueing cycles.

The Alewife-1000 CMMU has been completely implemented, and is in the final stages of testing. It consists of a 1μ, 3-layer metal, hybrid gate-array. Of the 14mm x 14mm die, the network interface described above consumes approximately 10mm^2 for random logic, and 4.5mm^2 for RAM. These numbers are pre-layout, statistical estimates.

References

[1] A. Agarwal *et al.* The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor. In *Proceedings of Workshop on Scalable Shared Memory Multiprocessors.* Kluwer Academic Publishers, 1991. An extended version of this paper has been submitted for publication, and appears as MIT/LCS Memo TM-454, 1991.

[2] David Chaiken, John Kubiatowicz, and Anant Agarwal. LimitLESS Directories: A Scalable Cache Coherence Scheme. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 224–234. ACM, April 1991.

[3] John Kubiatowicz. User's Manual for the Alewife 1000 Controller. ALEWIFE Memo No. 19, Laboratory for Computer Science, Massachusetts Institute of Technology, January 1991.

Closing the Window of Vulnerability in Multiphase Memory Transactions

John Kubiatowicz, David Chaiken, and Anant Agarwal*
NE43-629
kubitron@masala.lcs.mit.edu

Multiprocessor architects have begun to explore several mechanisms such as prefetching, context-switching and software-assisted dynamic cache-coherence, which transform single-phase memory transactions in conventional memory systems into multiphase operations. Multiphase operations introduce a *window of vulnerability* in which data can be lost before it is used either through protocol invalidation or cache conflicts. Losing data introduces damaging livelock situations. This abstract summarizes the work described in [1], which discusses the origins of the window of vulnerability and proposes an architectural framework that closes it. The framework is implemented in Alewife, a large-scale multiprocessor being built at MIT.

One of the major thrusts of multiprocessor research has been the exploration of mechanisms that provide ease of programming, yet are amenable to cost-effective implementation. To this end, a substantial effort has been expended in providing efficient shared memory for systems with large numbers of processors. Many of the mechanisms that have been proposed for use with shared memory, such as rapid-context switching, software prefetch, fast message-handling, and software-assisted dynamic cache-coherence enhance different aspects of multiprocessor performance; thus, combining them into a single architectural framework is a desirable goal.

Many of the mechanisms associated with shared memory attempt to address a central problem: access to global memory may require a large number of cycles. To fetch data through the interconnection network, the processor transmits a request, then waits for a response; thus, data accesses are *split-phase*. The request may be satisfied by a single memory node, or may require the interaction of several nodes in the system. In either case, many processor cycles may be lost waiting for a response.

To tolerate long access latencies, architects have proposed a number of mechanisms such as prefetching, weak consistency, multithreading, and software-enforced coherence. All are variations on a central theme: they allow processors to have multiple outstanding requests to the memory system. A processor launches a number of requests into the memory system and performs other work while waiting for responses. This ability reduces processor idle time and allows the system to increase its utilization of the network.

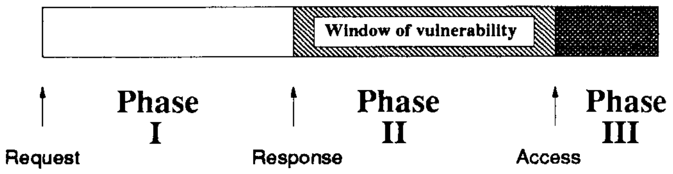


Figure 1: A basic multiphase transaction.

In a traditional shared-memory multiprocessor, remote memory requests can be viewed as split-phase transactions, consisting of a request and a response. The time between request and response may be composed of a number of factors, including communication delay, protocol delay, and queueing delay. Since a simple single-threaded processor can typically make no forward progress until its requested data word arrives, it spins while waiting. When the data word arrives, the processor consumes the data immediately, possibly placing it in the local cache.

Rather than spinning, a processor might choose to do other useful work. A processor with context-switching, for instance, might switch to another context; a system with high-availability interrupts might execute service routines. Once we free the processor from spinning, however, we introduce a third phase of data transactions, namely *access* (see Figure 1). The time between response and access, labeled as Phase II, reflects the fact that the processor does not consume data immediately upon its arrival. During this period, the data must be placed somewhere, perhaps in the cache or a temporary buffer. Note that a simple split-phase transaction can be seen as a degenerate multiphase transaction, with zero cycles between response and access.

The period between the response and access phases

*This work was conducted by the Alewife Research Group, supervised by Professor Anant Agarwal. The abstracted research was funded by DARPA contract #N00014-87-K-0825 and by by NSF grant # MIP-9012773.

of a primary data transaction is crucial to forward progress. Should the data be invalidated or lost due to cache conflicts during this period, the transaction is terminated before the requesting thread can make forward progress. Consequently, the period between response and access is a *window of vulnerability*. Closing the window of vulnerability involves ensuring forward progress for multiphase memory transactions.

The consequences of lost data are more subtle and perilous than simple squandering of memory resources. There exist scenarios in which processors repeatedly attempt to initiate transactions, only to have them canceled during the window of vulnerability. In certain pathological cases, individual processors are prevented from making forward progress by cyclic *thrashing* situations. While such situations may be rare, they are as fatal as any other infinite loop.

The window of vulnerability is also opened by a class of mechanisms that circumvent the shared memory interface, in order to facilitate the efficient use of critical multiprocessor resources. These mechanisms include fast I/O, interprocessor messages, synchronization primitives, and extensions of the memory system through software. All may be supported by providing processors with complete access to the interconnection network, and designing processors to be able to service asynchronous events rapidly (in tens of cycles). Since the ability to handle asynchronous messages quickly is crucial to system performance, processor interrupts that invoke message handling are high priority events. Unfortunately, such *high-availability interrupts* widen the window of vulnerability by extending the period of time that a processor must delay the completion of memory transactions.

This research identifies the livelock and deadlock problems associated with the window of vulnerability, and specifies an architectural framework that solves those problems. A combination of multiphase memory transactions and the mechanisms associated with shared memory may be implemented using an approach called *associative thrashlock*. Using this approach, the system keeps track of pending memory transactions in such a way that it can dynamically detect and eliminate pathological thrashing behavior. The framework consists of three major components: a small, associative set of *transaction buffers* that keep track of outstanding memory requests, an algorithm called *thrashwait* that detects and eliminates livelock scenarios that are caused by the window of vulnerability, and a buffer locking scheme that prevents livelock in the presence of high-availability traps.

What is the appropriate amount of hardware required to close the window of vulnerability? It is possible to imagine architectures that take completely different approaches to solving the problems associated with multiphase memory transactions. For example, the Alewife architecture forces contexts to *poll* until they complete their outstanding transactions. Alternatively, a system could eliminate the window of vulnerability inherent in a polling model by *signaling* or reenabling a context immediately when its memory access completes. Such is the case in dataflow or message-passing architectures. Polling has a smaller hardware cost and optimizes for the common case when average remote access latency is shorter than polling frequency. This is true precisely when the window of vulnerability is long. Signaling is less sensitive to remote access latency, but introduces additional hardware complexity. System parameters or philosophy determine whether polling, signaling, or a hybrid approach is most appropriate.

The associative thrashlock framework provides an inexpensive solution to the window of vulnerability problem in a polled system. The framework allows the use of caches to reduce the bandwidth required from the interconnect, and it permits processors to store just enough information to recreate the pipeline state of a context when necessary. Instead of closing the window of vulnerability by brute force, the Alewife architecture dynamically detects the situations that can lead to deadlock and livelock. Only when these relatively rare situations arise does the system close the window. The fundamental architectural trade-off pits hardware expense and complexity against exceptional events that are uncommon, but potentially fatal.

References

[1] John Kubiatowicz, David Chaiken, and Anant Agarwal. Closing the Window of Vulnerability in Multiphase Memory Transactions. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V), to appear*. ACM, October 1992.

The Data Network of the Connection Machine CM-5

Bradley C. Kuszmaul[1]

NE43-237

`bradley@lcs.mit.edu`

The Connection Machine Model CM-5 Supercomputer is a massively parallel computer system designed to offer performance in the range of 1 teraflops (10^{12} floating-point operations per second). The CM-5 obtains its high performance while offering ease of programming, flexibility, and reliability. The machine contains three communication networks: a data network, a control network, and a diagnostic network[1]. This abstract describes the organization of the data network and how it contributes to the design goals of the CM-5.

The basic architecture of the CM-5 data network is a *fat-tree*. Figure 1(a) shows a binary fat-tree. Unlike a computer scientist's traditional notion of a tree, a fat-tree is more like a real tree in that it gets thicker further from the leaves. Processing nodes, control processors, and I/O channels are located at the leaves of the fat-tree. (For convenience, we shall refer to all of these network addresses simply as processors.)

The CM-5 data network uses a 4-ary fat-tree, rather than a binary fat-tree. Figure 1(b) shows the interconnection pattern. The network is composed of router chips, each with 4 *child* connections and either 2 or 4 *parent* connections. Each connection provides a link to another chip with a raw bandwidth of 20 megabytes/second in each direction. (Some of this bandwidth is devoted to addressing, tags, error checking, and congestion.) By selecting at each level of the tree whether 2 or 4 parent links are used, the bandwidths between nodes in the fat-tree can be adjusted. Flow control is provided on every link. Messages travel up the tree by an adaptive random strategy until they reach a least common ancestor of the source and destination; Then messages travel down the tree following a determistic path.

The rest of this abstract describes three interesting issues: reflected in the design of the CM-5 data network.

- the *fetch deadlock problem* which is solved in the CM-5 by using a split network,
- the *router done problem*, which is solved by *Kirchoff counting*, and
- the *timesharing problem* which is solved by *all-fall-down mode.*

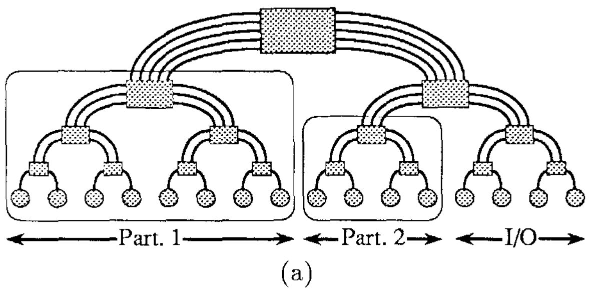
Fetch Deadlock Problem

The network has a contract with processors that guarantees all messages are delivered. The contract says, *"The data network promises to eventually accept and deliver all messages injected into the network by the processors and the processors promise to eventually eject all messages from the network when they are delivered to the processors."* The data network is acyclic from inputs to outputs, which precludes deadlock from occurring if this contract is obeyed. To send a message, a processor writes the destination processor address and data to be sent to a memory-mapped outgoing FIFO in its network interface. The processor then checks whether the message was accepted by the network. If not, which may occur because flow control information indicates that the network has not removed enough of a previous message from the outgoing FIFO, the processor can try again later. The processor may not block when attempting to put a message into the network, however, because that would violate the contract. Instead, the processor must attempt to receive any messages that have arrived. In the current implementation, the processor is involved in all transactions with the network.

Although the simple contract above can implement the sending of data through the network in a deadlock-free manner, it is not strong enough to allow some communication protocols to be implemented straightforwardly. For example, suppose each processor wishes to fetch a value from another processor, and the processors have finite buffer space. One processor may receive requests for data from many processors, but unfortunately, be unable to send responses because its outgoing FIFO to the data network is busy. The outgoing FIFO will eventually free, according to the contract, but only if the processor continues to accept delivery of messages from the network. A naive implementation of the fetch protocol might break the contract and deadlock the system.

The CM-5 solves the fetch deadlock problem in a simple fashion requiring no bookkeeping and only constant buffer space. Each processor has 2 outgoing and 2 incoming FIFO's in its interface to the data network: a *left* port and a *right* port. The topology of the network is such that all links reachable from the left port are unreachable from the right port and vice versa. Thus, the data network is really two independent, interleaved networks. To implement the round-trip protocol, requests can be sent on the left side of the network, and responses returned on the right side. If a processor cannot send a response on the right side and his constant-size buffer is full, he stops receiving on the left side. Since any processor requesting data has a place to put it, however, the processors can satisfy the contract on the

[1]Bradley C. Kuszmaul is a consultant to Thinking Machines Corporation and a graduate student in the MIT Laboratory for Computer Science where he is being supervised by Charles E. Leiserson.



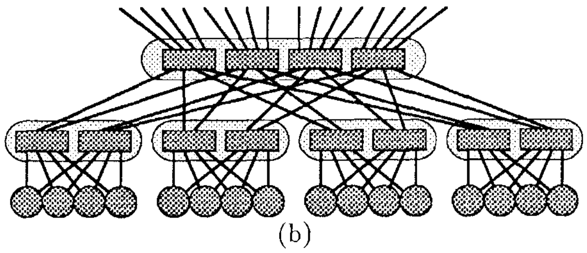


Figure 1: (a) A binary fat-tree. Processors are located at the leaves, and the internal nodes are switches. The hierarchical nature of a fat-tree can be exploited to give each user partition a dedicated subnetwork which cannot be interfered with by any other partition's message traffic.
(b) The interconnection pattern of the CM-5 data network. The network is a 4-ary fat-tree in which each internal node is made up of several router chips. Each router chip is connected to 4 child chips and either 2 or 4 parent chips.

right side and the responses will eventually clear out. Because the responses on the right side will eventually clear out, a processor can always eventually accept every request that arrives on the left side, and thus the processors satisfy the contract on the left side. Consequently, deadlock cannot occur.

In fact, deadlock cannot occur even if responses are sent on both sides of the data network, as long as requests are sent on one side only. The data network requires no more than two sides, even when there are many intermediate destinations, because such a communication pattern can be broken into a collection of round trips.

Router Done Problem

The router-done operation is a specialized reduction that lets the processors know when communications involving the data network are complete. In the data-parallel programming model, this operation is often required so that processors know when it is safe to proceed to the next data-parallel operation.

The basic idea behind the implementation of router-done is "Kirchoff's current law." When all processors have completed sending their messages and the number of messages that entered the data network equals the number that have left, the routing cycle is complete. The network interfaces keep track of the number of messages that enter and leave the data network. After a processor has completed sending all its messages, it pushes a message into the outgoing router-done FIFO. When all processors have sent messages into their outgoing FIFO's, the control network continually monitors the difference between the total number of messages put into the data network and the number removed from the data network. When this number becomes zero, each processor receives a message in its incoming router-done FIFO informing it that the data network is done routing messages. Using this "Kirchoff" method has the additional benefit that if a hardware error causes messages to be lost or created, the error can be detected and signaled, either by a failure of the router-done operation to complete on the one hand or by the unexpected arrival of a message after the router-done operation has completed on the other.

Timesharing Problem

Each user partition in the CM-5 system is capable of being run in either a batch or a timesharing mode. The requirement for timesharing raises the issue of what should be done with messages that are in transit in the routing network when a user's timeslice has expired and another user must be given access to the partition. The system cannot afford to wait until the user completes his communication, since the communication may not terminate for a very long time, and it in fact may not ever complete if the user has deadlocked himself.

This problem of swapping users is solved in the CM-5 by putting the data network into *all-fall-down* mode. Instead of trying to route messages to their destinations, the network misroutes each one down through the network so they are distributed evenly among the processing nodes. In the worst case, each node receives only a small number of misdirected messages, even if all were headed for the same destination processor. The all-fall-down messages are then saved in memory with the user's state. When the user's task is resumed, the system resends them to their true destinations. Even if a timeshared user deadlocks, this context-switching mechanism precludes him from unduly affecting the other users who are sharing his partition.

References

[1] C. E. Leiserson, Z. S. Abuhamdeh, D. C. Douglas, C. R. Feynman, M. N. Ganmukhi, J. V. Hill, W. D. Hillis, B. C. Kuszmaul, M. A. St. Pierre, D. S. Wells, M. C. Wong, S.-W. Yang, and R. Zak. The network architecture of the connection machine CM-5. In *Symposium on Parallel and Distributed Algorithms '92*, June 1992.

Highly Parallel Alpha-Beta Search

Bradley C. Kuszmaul[1]
NE43-237
`bradley@lcs.mit.edu`

Alpha-beta search is an example of a very good serial algorithm for searching minimax trees, which are used to model two-player adverserial games. Even though it looks like minimax tree search ought to be highly parallel, researchers have had difficulty finding an algorithm that achieves good parallel speedup. We have been exploring a new parallel algorithm for minimax tree search, and our results look very promising.

Figure 1(a) shows the standard sequential alpha-beta search algorithm. Note that β is a loop-invariant in the body of the loop, and that if β is infinite, then the loop will execute for each child regardless of what the subsearches produce.

Alpha-beta search works the best when the tree is searched in the right order. A *best-ordered* tree is one in which the first child is always the best (or sufficiently good to produce a cutoff). Figure 1(b) shows Knuth's *critical tree*[1] for a uniform tree of degree 3 and height 4. The critical tree can be thought of as the *proof tree* that the tree is best-ordered, since to prove that the tree is best-ordered, the critical tree must be traversed.

This abstract describes two algorithms: Our non-strict parallel alpha-beta search uses non-strict procedure application to implicitly take advantage of the critical tree. Our non-strict algorithm achieves very good parallel speedup, but seems to have very poor space complexity. Our strict parallel alpha-beta algorithm makes explicit use of the critical tree; We have some good theoretical space and time bounds for our strict parallel algorithm.

Non-strictness gives parallel speedup

Our first algorithm makes use of non-strict procedure application, supported, for example, by the Id programming language. Non-strict application allows a procedure to start executing before all of its arguments are present. For example, in the alpha-beta algorithm, observe that if we search a node, we will always search that node's first child, regardless of what the values of α and β are. Also observe that if $\beta = \infty$ then it does not matter what the values returned by the subsearches are (as long as they are finite values); All of the children will be searched. In fact, if, at the root, $\alpha = -\infty$ and $\beta = \infty$, non-strict procedure application will expand precisely the critical tree.

Non-strict application will expand the tree in parallel, but the "max" operations and the comparisons of results returned by subsearches form a serial dataflow graph of depth equal to the number of nodes in the tree. We use a technique that we call *fast minimax lookahead* to propagate a bound for the values of a node up the tree. For example, if you know that search(c_0) $\leq -v$, then you know that you can achieve at least v at this node by choosing c_0. (Remember that search(c_0) gives the value of c_0 from your opponent's point of view. Thus an upper bound on the value of c_0 from your opponent's point of view is a lower bound on the value of c_0 from your point of view.) If you know that search(c_i) $\geq -v_i$ for each i, then you know that search(n) $\leq \max_i v_i$.

Our algorithm expands the tree in parallel using a left-biased globally synchronous strategy. The parallel computation consists of a sequence of phases. During each phase, we expand the leftmost cP nodes that want to expand, where P is the number of processors and c is some small constant, such as $c = 10$. We push the computation as far as we can without expanding any more nodes, and then we synchronize and start another phase.

Our non-strict algorithm achieves good parallel speedup. Simulations for machines up to tens of thousands of processors indicate speedups to within 50% of linear speedup if the search trees are at least an order of magnitude larger than the number of processors. These simulations were done on synthetic game trees and on real chess trees.

However our non-strict algorithm apparently uses an unbounded amount of space. This happens when some subtree of the search becomes serialized for a while and the processors start expanding part of the tree further to the right. The newly expanded nodes produce a bound on the values of their nodes, but the nodes cannot be deallocated because we may discover later that we need to further expand the partially evaluated tree. I.e., some nodes get stuck where they have no more work to do, but they can not be deallocated. The number of stuck nodes can be very large, especially in real chess trees. It is very difficult to analyse the space requirements of algorithms that use non-strict procedure application.

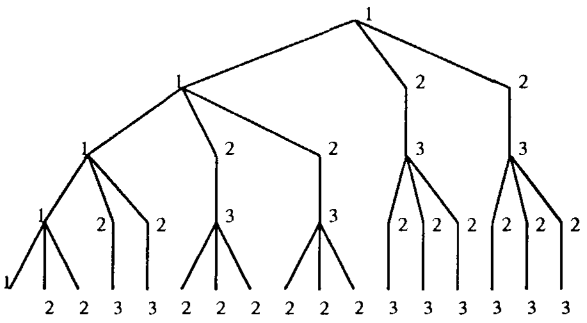
For best-ordered trees, our globally synchronous algorithm apparently achieves very good space bounds, approximately cdh nodes per processor, with a variation that is related to P.

We run out of memory with a left-biased asynchronous version of our algorithm. Our asynchronous variation assigns each node of the tree to a processor. On each time step, each processor expands the leftmost node in its local

[1]This work is being supervised by Prof. Charles E. Leiserson, and is supported in part by the DARPA contracts N00014-89-J-1988 and N00014-91-J-1698

```
procedure search ($n$, $\alpha$, $\beta$)
  int $b$,$s$;
  $b := -\infty$;
  if leaf?($n$) then return staticeval($n$);
  for $c_i$ in children($n$)
    $s := -\text{search}(c_i, -\beta, -\alpha)$;
    if $s \geq \beta$ then return $s$;
    $b := \max(b, s)$;
    $\alpha := \max(\alpha, s)$;
  endfor;
  return $b$;
end search;
```

(a)



(b)

Figure 1: (a) The standard serial alpha-beta search procedure, expressed in the negamax form; The recursive call changes signs and the order of the arguments, and inverts the sign of the result. The node n is being searched with bounds *alpha* and *beta*. If value(n) <= α or β < value(n), we do not need an exact value for n — any returned value outside of (α, β) is interpretted as "fail low" or "fail high". Note that β is a loop invariant and that if $\beta = \infty$, the loop will execute over all the children of node n..
(b) The critical tree for a uniform tree of height $h = 4$ and degree $d = 3$ with $d^{\lceil h/2 \rceil} + d^{\lfloor h/2 \rfloor} - 1 = 17$ leaves. The nodes down the left spline are type 1 nodes. The "other" children of type 1 nodes are type 2 nodes. The first child of a type 2 node is a type 3 node. All of the children of a type 3 node are type 2 nodes. The remaining subtrees below type 2 nodes are not in the critical tree; They are hopefully pruned during alpha-beta search. In a best-ordered tree, exactly the critical tree is examined by alpha-beta search.

collection of nodes. In this case the variation in the number of frames needed grows with N, the number of nodes in the tree, which is a large number. The variation causes us to run out of memory in some nodes.

The non-strict algorithm is currently implemented and running (out of space) on a Connection Machine CM-5 supercomputer.

Strict parallel alpha-beta search

We are currently examining a new algorithm which exploits some of the ideas from our non-strict algorithm, but which should have practical space requirements. The main idea is to strictify the computation, without substantially reducing the parallelism. Most of the parallelism was achieved by computing the bounds on the value of the nodes without knowing the precise values of α and β. We create new procedures that explictly search for a bound, based on the empirical evidence that in chess trees, the first node considered is usually a good enough move to achieve cutoff (failing high).

To evaluate a node for an exact value, evaluate the first child for an exact value v, and then evaluate all of the other children, in parallel, to prove that they are worse than v. One variation on this algorithm simply searches the first child of type 2 nodes and all the children, in parallel, of type 3 nodes. If the proof fails for some child c_i, then re-search c_i for an exact value.

Our strict parallel alpha-beta search may do some extra work, because it may perform re-searches on certain subtrees several times. We have several bounds on the amount of extra work that might be done.

- For best-ordered trees, our algorithm does exactly the same amount of work as serial alpha-beta.
- For worst-ordered trees, our strict algorithm does at most a factor of 2 more work than serial alpha-beta.
- For any uniform tree of degree d and depth h, we do at most a factor of 2 extra work if h is even, and at most a factor of $d/2$ extra work if h is odd.
- We believe that there is a variation on the algorithm that does at most a factor of $h/2$ extra work for any tree (but we have not shown this).

It has been shown that the left-biased globally-synchronous scheduling strategy will achieve very good space bounds.

We are still working on our highly parallel alpha-beta algorithm, and it looks very promising.

References

[1] Donald E. Knuth and Ronald W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence,* 6(4):293–326, Winter 1975.

Automatic Generation and Verification of Sufficient Correctness Properties of Synchronous Array Processors

Stan Y. Liao
Room 36-888, Research Laboratory of Electronics
syliao@rle-vlsi.mit.edu

May 26, 1992

The ever-increasing demands for high performance in real-time signal processing and scientific computations have led to novel computer architectures, among which are array processors [5]. Many research efforts have been devoted to systematically mapping high-level descriptions (such as recurrence equations and dependency graphs) to array processors (*e.g.* [2] [6]).

As with other logic synthesis methods, although the circuits produced by these design methodologies can sometimes be shown to be correct-by-construction, an independent verification is necessary to ensure the correctness of the final design [4]. In this paper we present a strategy for automatically generating and verifying sufficient correctness properties for synchronous array processors. The targeted circuits are array processors designed from localized, highly regular dependency graphs (DGs), such as in [6]. For example, Figure 1 shows a specification-implementation pair of an array processor for computing Gaussian elimination on a 3×3 matrix and a 3-vector.

As in [1], we will take correctness to mean that the implementation is in β-relation with the specification, and we express sufficient correctness conditions as past-tense CTL formulae which can be verified by symbolic model checking using binary decision diagrams (BDDs) [3]. The method presented in [1] is not directly applicable to the verification of array processors, although we would like to use a compositional strategy for array processors, too, because it greatly reduces the complexity of verification. Unlike microcoded processors, array processors cannot readily be divided into datapaths and control circuitries. Each processing element (PE) in an array processor has its local control and local datapath, and interacts with its neighbors. In addition, since array processors perform special purpose computations, they are usually attached to a host computer, from which they receive data streams. The host computer sends data to the array processor only when computations need to be done in it, and the array processor may simply remain idle when it does not receive any data from its host computer. In some sense the control signals from the host computer reside in the data streams. This makes the separation of control circuitries and datapaths even more difficult.

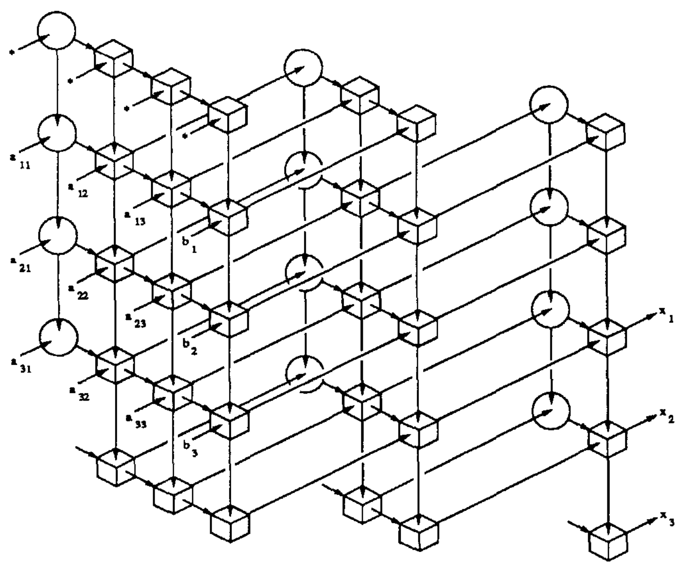
This problem can be remedied by constructing an array of controllers that are connected in the same manner, and an auxiliary machine, which is created solely for the purpose of verification. It usually consists of a counter which keeps a timing reference, and some logic that generates control signals abstracted from a typical data stream. For example, in the Gaussian elimination processor in Figure 1, the control signals would be one of the following: marker (represented as an asterisk), "nothing" (represented by a dot), or datum.

The sufficient correctness conditions for the controllers are then as follows:

1. Each PE receives a marker at the time indicated by the schedule in the specification. This ensures that the connection between the PE are correct.

2. Each PE asserts correct control vectors at relevant time points.

This work is supervised by Professor Srinivas Devadas. It is supported in part by the Defense Advanced Research Projects Agency under contract N00014-91-J-1698.



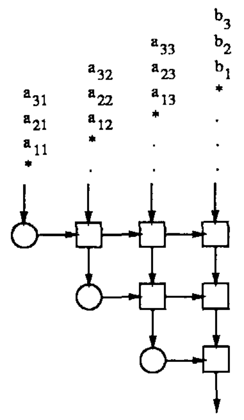


Figure 1: Dependency Graph and Implementation for Gaussian elimination

3. In each PE, the result of a computation needed for the next one is not corrupted until it is used.

CTL formulae representing these conditions can be automatically derived from the dependency graph. These formulae are verified on the composite finite state machine (the auxiliary machine and the array of controllers). Because each PE is not active for all time points during an instance of computation, we restrict the traversal of the FSM to the set of states where the value of the counter in the auxiliary machine is between the minimum and the maximum of the schedules assigned to the PE.

We will present the results for several examples in the talk at the workshop.

References

[1] F. Van Aelten, J. Allen, and S. Devadas. Compositional Verification of Systems with Synchronous Globally Timed Control. In *Advanced Research in VLSI and Parallel Systems; The 1992 Brown MIT Conference*, March 1992.

[2] D. G. Baltus. *The Automated Synthesis of VLSI Array Structures from Algorithmic Descriptions.* PhD thesis, Massachusetts Institute of Technology, 1992.

[3] J. Burch, E. Clarke, K. McMillan, and D. Dill. Sequential Circuit Verification Using Symbolic Model Checking. In *Proceedings of the 27th Design Automation Conference*, pages 46–51, June 1990.

[4] A. Ghosh, S. Devadas, and A. R. Newton. *Sequential Logic Testing and Verfication.* Kluwer Academic Publishers, 1992.

[5] S. Y. Kung. On Supercomputing with Systolic/Wavefront Array Processors. In *Proceedings of the IEEE*, volume 72, pages 867–884, July 1984.

[6] S. Y. Kung. *VLSI Array Processors.* Prentice-Hall, Englewood Cliffs, N. J., 1988.

Concurrent Counting for Multiprocessor Load Balancing*

Beng-Hong Lim
545 Technology Square, Rm. 633
Cambridge, MA 02139
bhlim@mit.edu

1 Introduction

Dynamic load balancing can have a dramatic effect on the performance of irregular parallel programs. Various schemes have been proposed for load balancing such programs. In this research, we consider the implementation of two kinds of load balancing techniques on large-scale multiprocessors: self-scheduling of DOALL loops and the task queue model in which processes dynamically insert and remove tasks from a shared queue.

The notion of *shared counting* is central to each of these load balancing problems. Processes must cooperate to assign successive values from a given range: either loop indices or slots in a queue. A common problem faced by load balancing algorithms is contention at the shared counter. A good implementation should reduce contention *and* allow high levels of concurrency. In this research, we consider the following techniques for counting: (1) spin locks with exponential backoff, (2) Anderson's "queue" locks, (3) software combining trees [2], and (4) "bitonic" counting networks [1]. (See [4] for a study of the locking techniques.)

This research makes the following contributions. Each of the counting techniques we consider has been independently proposed as a way to alleviate contention in highly concurrent systems. Here, for the first time, they are compared directly on a realistic large-scale distributed-memory multiprocessor. Moreover, this work is the first systematic experimental exploration of counting network performance on a distributed memory machine.

In the rest of this abstract, we will briefly describe the experiments that we ran, and present a small sample of results that we obtained. More details about the counting techniques and results can be found in [3].

*Based on joint work with Maurice Herlihy and Nir Shavit.
Faculty Supervisor: Prof. Anant Agarwal.
The research reported here was supported in part by NSF Experimental Systems grant # MIP-9012773, and in part by DARPA contract # N00014-87-K-0825.

2 Experiments and Results

We ran a series of simple benchmarks on a simulated 64-processor Alewife machine, a cache-coherent distributed-memory machine supporting the shared-memory programming model. The experiments were run on an accurate cycle-by-cycle simulator for the Alewife architecture. A sample of results from these experiments is presented in Figure 1 and described below.

Counting Benchmark In this benchmark, 64 threads increment a shared counter 32 times each for a total of 1024 increments. This provides a simple baseline for comparing counting techniques.

The first graph shows that at high levels of concurrency the counting network and combining tree outperform both spin locks and queue locks. This supports the intuition that as concurrency increases, locks need to be distributed to avoid the detrimental effect of contention. The measurements also show that both combining trees and counting networks scale at about the same rate. This is an encouraging result for counting networks because software combining trees are considered the best-known method for updating a shared counter without explicit hardware support for combining.

Self-Scheduling Benchmark To model self-scheduling, n processes execute 2048 increments on a shared counter. Between each increment, each process pauses for a duration randomly chosen between 0 and 1000 cycles. The increment models a process obtaining an iteration, and the random pause represents the execution of that loop iteration.

The second graph compares the performance of spin locks, combining trees, and counting networks on this benchmark. We see again that spin locks do not scale beyond a certain number of processors. Both the combining tree and counting network allow speedups on the benchmark all the way up to 64 processors.

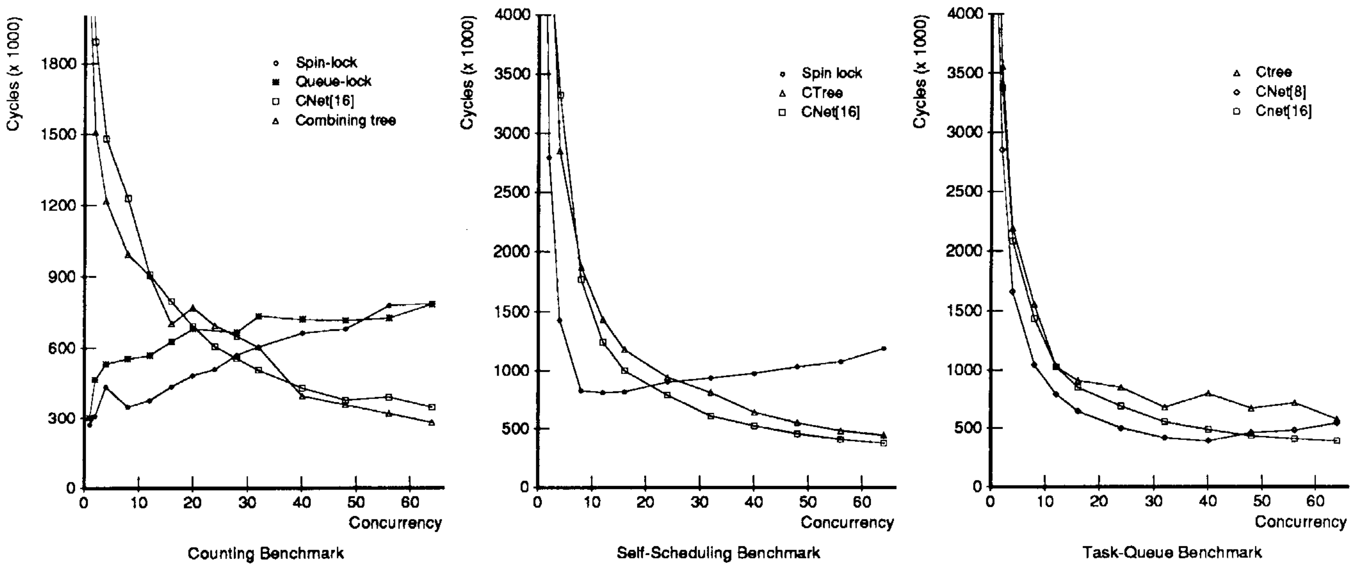


Figure 1: Elapsed times of benchmarks measuring the scalability of techniques for concurrent counting.

Task-Queue Benchmark For this benchmark, n processes repeatedly (1) dequeue a task from a shared queue, (2) pause for a duration randomly chosen from a uniform distribution between 0 and 1000 cycles, then (3) enqueue a new task. The queue itself consists of a buffer, a *head* counter indicating the first full slot, and a *tail* counter indicating the first empty slot. A process dequeues a task by atomically incrementing the head counter and removing one task from the corresponding buffer slot. Enqueues are performed analogously. The benchmark halts when 1024 tasks have been dequeued and executed.

The third graph shows that the combining tree and counting network of width 16 have equivalent performance at low levels of concurrency. We also see that the counting network of width 8 has the best performance at low levels of concurrency, *i.e.*, when the arrival rate of increment requests is low.

At higher levels of concurrency, the counting network outperforms the combining tree and scales less erratically. The reason for this is that the combining tree is sensitive to the arrival times of increment requests at a node. If two arrivals at a node do not arrive sufficiently close enough to each other, combining does not occur and the opportunity for parallelism is wasted.

3 Conclusions

Although the two locking techniques are known to scale well on small-scale, bus-based multiprocessors, our experimental results show that their performance degrades in a distributed memory machine as the level of concurrency increases. This degradation occurs because counting using locks is inherently sequential.

Both counting networks and combining trees substantially outperform the locking methods by both reducing contention *and* taking advantage of concurrency. We also found that combining trees are sensitive to variations in the inter-arrival times of increment requests, thus making counting networks an attractive choice for implementing concurrent counting.

References

[1] J. Aspnes, M.P. Herlihy, and N. Shavit. Counting Networks and Multi-Processor Coordination. In *Proceedings of the 23rd Annual Symposium on Theory of Computing*, May 1991.

[2] J.R. Goodman, M.K. Vernon, and P.J. Woest. Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors. In *Proceedings of the 3rd ASPLOS*, pages 64–75. ACM, April 1989.

[3] Maurice Herlihy, Beng-Hong Lim, and Nir Shavit. Low Contention Load Balancing on Large-Scale Multiprocessors. In *Proceedings of the 4th Annual Symposium on Parallel Algorithms and Architectures*, June 1992.

[4] John M. Mellor-Crummey and Michael L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

AN IMPLICIT PARTICLE METHOD FOR MONTE CARLO DEVICE SIMULATION*

JENNIFER LLOYD†

This abstract presents a new method for integrating the semiclassical motion equations applied to transient Monte Carlo semiconductor device simulation.

Ensemble Monte Carlo simulation entails tracking the evolution of a system of particles in space and time. This motion, in the abscence of collisions and magnetic fields, is described by the semi-classical motion equations:

$$\frac{d\boldsymbol{r}}{dt} = \boldsymbol{v}_n(\boldsymbol{k}) = \frac{1}{\hbar}\nabla_{\boldsymbol{k}}\mathcal{E}_n(\boldsymbol{k}) \tag{1}$$

$$\hbar\frac{d\boldsymbol{k}}{dt} = -q\boldsymbol{E} \tag{2}$$

where $\boldsymbol{r}$ is the position of a particle, $\boldsymbol{k}$ is the wavevector, $\hbar$ is Planck's constant, $\mathcal{E}_n(\boldsymbol{k})$ describes the band structure of the semiconductor, q is the magnitude of the electronic charge, and $\boldsymbol{E}$ is the electric field on the particle. These equations are typically discretized with an explicit time-integration scheme, so that the electric field used to compute the particle positions is based on a calculation at the current timestep [1, 2].

ALGORITHM 1 (IMPLICIT DIRECT-FORCE ALGORITHM).

```
Initialize particle positions, $r_i^0$, velocities $v_i^0$.
For k = 1 to number_of_timesteps of size h
  For i = 1 to number_of_particles N
    Compute explicit position, $r_i(t+h) = r_i(t) + hv_i$.
  For j = 1 to number_of_relaxation_iterations
    For i = 1 to number_of_particles N
      For n = 1 to number_of_Newton_iterations
        Compute $E_i$ contribution from the doping charge.
        Compute $E_i$ contribution from the mobile charges.
        Compute Jacobian matrix and Newton right hand side.
        Compute position update, $\delta_i$, using Newton.
        Update position, $r_i(t+h) = r_i(t+h) + \delta_i$.
  For i = 1 to number_of_particles N
    Update velocity, $v_i(t+h) = v_i(t) - \frac{hq}{2m}[E_i(t) + E_i(t+h)]$.
```

FIG. 1. *The implicit Monte Carlo algorithm using direct force electric field calculations.*

Another approach is to instead discretize the motion equations with an implicit multistep method, so that the electric field used to compute the new particle position is a function of the electric field at the next timestep. Assuming a parabolic band structure and particle effective mass m^*, so that $\boldsymbol{k} = \boldsymbol{v}\frac{m^*}{\hbar^2}$, the motion equations become:

$$\boldsymbol{r}_i(t+h) = \boldsymbol{r}_i(t) + \frac{h}{2}\left[\boldsymbol{v}_i(t) + \boldsymbol{v}_i(t+h)\right] \tag{3}$$

* This work was supervised by Prof. Jacob K. White and supported by DARPA contract N00014-91-J-1698 and NSF contract MIP-8858764 A02.

† Research Laboratory of Electronics, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139. E-mail: *jen@rle-vlsi.mit.edu*

$$(4) \qquad v_i(t+h) = v_i(t) - \frac{hq}{2m}\left[E(r_1(t)\cdots r_n(t)) + E(r_1(t+h)\cdots r_n(t+h))\right]$$

when integrated with the energy-preserving trapezoidal method. This set of equations is implicit and non-linear, and must be solved at each timestep for each particle. Since there is a non-linear system of equations associated with each particle, a Gauss-Seidel relaxation is used to iterate for the particle position updates at each timestep. Within the relaxation scheme, the update for each particle requires solving a three-dimensional problem for which a Newton's method is employed. The algorithm for the new computation is given in Figure 1. The combined relaxation/Newton technique used to solve the implicit problem is robust, and converges within a few iterations of the relaxation (combined with 1 Newton iteration.)

The rationale for using an implicit instead of an explicit method is that the implicit method captures the fact that for large timesteps the particle's electric field will change over the timestep period. Using an explicit method, the electric field used to determine the particle's movement can easily be incorrect. On the other hand, an implicit method makes a correction to this error by using the average electric field over the time-step, thus making it more un-likely that a particle will ever get too close (and see a large force from) another particle.

For simulations of an ensemble of particles, our results indicate that implicit methods show less timestep dependence and more accuracy than explicit methods. This is seen by examining the normalized average temperature of the ensemble over the simulation time, as shown in Figure 2 for both an implicit and an explicit method. Using the explicit method, the total energy of the system grows without bound for large timesteps, although for small enough timesteps, the solution will approach that of the ideal solution. However, by using an implicit method, the total energy of the system remained stable over time, so that the temperature is bounded.

Additionally, increasing the number of simulation particles does not change the stability of the time-integration problem, but does smooth out the variations in the simulation results (e.g. temperature and current.) A rather large number of particles is actually required to accurately simulate these systems, although a small number of particles can be used to show trends in the numerical methods.

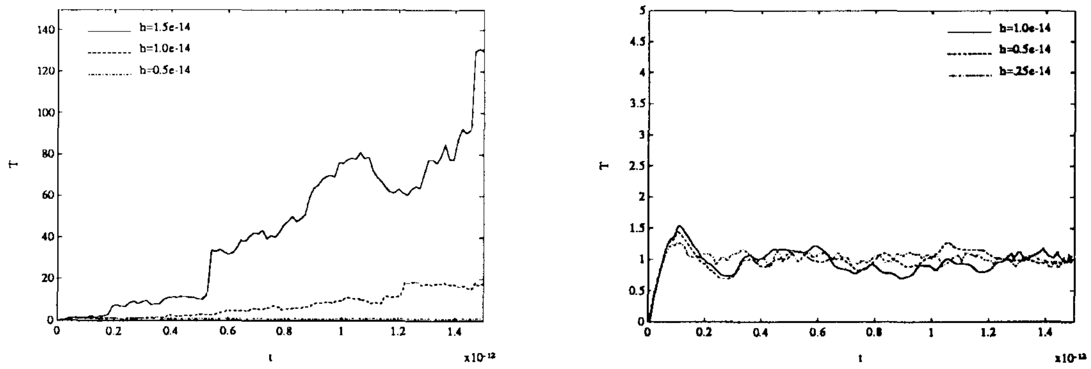


FIG. 2. *Simulation results showing the temperature growth over time with both explicit and implicit simulations, respectively, for several timestep sizes (in seconds).*

REFERENCES

[1] M. FISCHETTI, S. LAUX, AND W. LEE, *Monte Carlo Simultation of Hot-Carrier Transport in Real Semiconductor Devices*, Solid State Electronics, 32 (1989), pp. 1723–1729.

[2] E. VENTURI, E. SANGIORGI, R. BRUNETTI, W. QUADE, C. JACBONI, AND B. RICCO, *An Efficient Monte Carlo Simulator for High-Energy Electrons and Holes in MOSFET's*, in Proceedings of the IEDM, 1990.

Fault-Tolerant Sorting Circuits

Yuan Ma*

Room 2–342, MIT
yuan@math.mit.edu

We study fault-tolerant sorting circuits under 2 types of fault models in both adversary and random cases. A passive-faulty comparator outputs 2 numbers in the wrong order iff the 2 numbers are input in the wrong order. A destructive-faulty comparator outputs 2 numbers in the wrong order independent of the input order. A circuit is called random-fault-tolerant if it works (reasonably) well with probability at least $1 - \frac{1}{N^\alpha}$ for some constant α (so-called high probability) even when each comparator is independently faulty with a constant probability. A circuit is called k-adversary-fault-tolerant if it works (reasonably) well as long as the total number of faulty comparators is smaller than or equal to k. We will use N to denote the total number of inputs to a circuit.

1 Previous Work on Fault-Tolerant Sorting Circuits

Yao and Yao [6] were the first to study fault-tolerant sorting circuits under passive fault model. An easy and natural way to derive passive-fault-tolerance is to replicate each comparator for sufficiently many times. The most interesting and natural question is if one can do anything better. For the adversary passive faults, previous work [4] [5] [6] focused on how to tolerate only constant number of faults effectively. When k is not a constant, the simple replication technique implies a trivial $O(\log N + k \log N)$ upper bound on the depth of k-adversary-passive-fault-tolerant sorting circuit, but no one knew if there existed any such circuit with $o(\log N + k \log N)$ depth. For the random passive fault, Yao and Yao asked what is the optimal size of a random-passive-fault-tolerant circuit for sorting or merging. The trivial $\Omega(N \log N)$ and $O(N \log^2 N)$ (achieved by replication) remained the only bounds for Yao and Yao's question on both sorting and merging, even though many authors had been working on this subject. In other words, like in the adversary case, no one knew if one could do anything better than the simple replication.

Assaf and Upfal [1] introduced the destructive fault model. Under both passive and destructive fault models, they studied the random-fault-tolerant sorting problem in a more powerful network model other than the classical circuit model. The main reason they switched to that model is that the classical circuit can not sort everything exactly to the correct position with a good probability when destructive faults are allowed. (Please note that the replication technique does not work under destructive faults.) In fact it was showed in [3] that under destructive fault model, in any circuit of any depth, with high probability at least one output is $\Omega(\log N)$ away from its correct position. Leighton, Ma and Plaxton [3] took another approach to study random-destructive-fault-tolerant sorting circuits. They restricted their attention on the circuit model, but instead of insisting that the faulty circuit be an exact sorting circuit, they only required that the faulty circuit be a near-sorting circuit which should output every item to within $O(\log N)$ (optimal) from its correct position. They showed an $O(\log^3 N)$ upper bound and an $\Omega(\log^2 N)$ lower bound on the depth of such near-sorting circuits. (For the upper bound, they need to assume that the fault probability is sufficiently small.) They left open the question that if $O(\log^2 N)$ is indeed the tight bound. For the passive fault, they constructed an $O(\log N \log\log N)$ depth circuit which sorts any given permutation (but not necessarily all permutations) with high probability. This is a very important progress on Yao and Yao's question, but does

*Some of the results are joint work with Yuan Ma's advisor Professor Tom Leighton. This research is supported in part by MIT Applied Mathematics Fellowship.

not answer it since that question is on the passive-fault-tolerant sorting (or merging) circuit which works on all (possible) input permutations.

2 New Results

(1) We show that the work of [3] on passive fault actually implies an $O(\log N \log\log N)$ depth random-passive-fault-tolerant merging circuit. This settles Yao and Yao's open question on merging to within an $O(\log\log N)$ factor.

(2) We build a random-destructive-fault-tolerant-near-sorting circuit with the optimal $\Theta(\log^2 N)$ depth. As in [3], we need to assume that the fault probability is sufficiently small in order to prove the upper bound. This answers a question which was left open in [3] and posted again in [2].

(3) We construct a k-adversary-passive-fault-tolerant sorting circuit of $O(k \log\log N + \log N)$ depth. Note that $\Omega(k + \log N)$ is an easy lower bound on the depth of such circuit. Hence our circuit has the optimal $O(\log N)$ depth when $k = O(\log N / \log\log N)$. On the other hand, when $k = \Omega(N^\alpha)$ for some constant α, we have another construction which achieves the optimal $O(k)$ depth. These two results are the first k-adversary-passive-fault-tolerant sorting circuits of optimal depth for non-constant k. Also we have a new construction of so-called correction-network which can be used to tolerate constant number of faults effectively. Even though the construction in [5] is already asymptotically optimal, our construction is simpler and of fewer number of comparators.

(4) We construct a k-adversary-destructive-fault-tolerant near-sorting circuit of $O(k \log(N/k))$ depth and show that this is indeed optimal. Here near-sorting means to output everything to within (optimal) k away from its correct position. The key of this result is an interesting lower bound proof which gives a lot of insight on the structure of any destructive-fault-tolerant sorting circuit and somehow motivates our other work on destructive fault model. No result was previously known on adversary destructive fault.

Our results are summarized in the following table.

	passive	*destructive*
random	$O(\log N \log\log N)$ (for merging circuit only)	$\Theta(\log^2 N)$ (when fault probability is small)
adversary	$O(k \log\log N + \log N)$	$\Theta(k \log(N/k))$

The bounds on the depth of fault-tolerant circuits. The bound under random-passive fault is for merging circuit, the rest are all for sorting or near-sorting circuit.

References

[1] S. Assaf and E. Upfal. Fault tolerant sorting network. In *Proceedings of the 32st Annual IEEE Symposium on Foundations of Computer Science*, pages 275–284, October 1990.

[2] T. Leighton, C. Leiserson, and N. Kahale. *Research Seminar Series 15: Advanced Parallel and VLSI Computation.* MIT Press., 1991.

[3] T. Leighton, Y. Ma, and G. Plaxton. Highly fault-tolerant sorting circuits. In *Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science*, pages 458–469, October 1991.

[4] L. Rudolph. A robust sorting network. *IEEE Transactions on Computers*, C-34:326–335, 1985.

[5] M. Schimmler and C. Starke. A correction network for n-sorter. *SIAM J. Comput.*, 18:1179–1187, 1989.

[6] A. C. Yao and F. F. Yao. On fault-tolerant networks for sorting. *SIAM J. Comput.*, 14:120–128, 1985.

Reduced-Latency Memory Assignment for Multiprocessor Caches

Marios Papaefthymiou Anant Agarwal John Guttag

MIT Laboratory for Computer Science
Cambridge, MA 02139

Multiprocessors that support a shared-memory programming model provide the abstraction of a single coherent memory that is equally easily and equally efficiently accessible by multiple processors. Typically, the shared-memory abstraction is implemented by a large amount of physical memory that is accessed through a network. In bus-based machines, the memory is implemented as a single module accessed over the bus, while in most large-scale machines the memory is physically distributed among all the processing nodes. Virtually all machines that support the shared-memory programming abstraction provide local caches at each processor. Caches automatically replicate memory locations close to the processor and avoid expensive network traversals for most memory accesses. In this paper we are concerned with compile-time techniques that can be used to achieve better performance by improving cache utilization. Specifically, we investigate the problem of assigning data blocks to memory in a way that will minimize the impact of collisions in direct-mapped multiprocessor caches. We characterize the problem in precise mathematical terms and present an efficient procedure for finding approximate solutions to it. The procedure incorporates a new technique, grey coloring, that reduces latency in the presence of collisions by distributing cache misses among processors.

Today, most large caches are direct-mapped. In a direct-mapped cache, each line in the main memory corresponds to a unique entry in the cache memory. If the cache has a total of S lines, this entry is specified by the $log_2 S$ low order bits of the line's address in the main memory. The problem with a direct-mapped cache is that if a processor's working set includes two or more locations in the main memory that correspond to the same entry in the cache, then there is a conflict that leads to cache misses. Whenever a miss occurs, the processor must wait for a line to be read from main memory into the cache, and consequently, as the number of misses in each individual cache increases, the overall latency of a computation also increases.

In this paper we are concerned with the problem of minimizing the impact of cache conflicts. We deal with conflicts between blocks of data, as opposed to conflicts between individual cache lines, because data blocks allow a compiler to exploit the locality of reference exhibited by most programs. Specifically, we are given a set D of memory-resident data blocks and a set C of direct-mapped caches. In each cache we must store a subset of D, and each cache can hold up to k data blocks. A data block may have to be stored in multiple caches. We want to find an assignment of the data blocks to the main memory such that the maximum number of conflicts in any cache is minimized. Consider, for example, the execution of a parallel program running on three processors P0, P1, and P3, as illustrated in Figure 1. Assume processor P0 accesses data blocks a and b, P1 accesses data blocks b and c, and P2 accesses c and d. In addition, assume that each processor accesses an identical code segment x, and that no more than three data blocks can fit in any cache. The memory assignment shown in Figure 1(a) results in conflicts between b and c in the cache associated with processor P1. There exists a conflict-free assignment, however, which is illustrated in Figure 1(b). We show by reduction from graph k-colorability that even the restricted problem of finding a conflict-free assignment is $\mathcal{NP}$-complete. Thus, we try to find approximate solutions to the general memory assignment problem using efficient heuristic techniques.

Our basic strategy is illustrated in Figure 2. Data blocks a, b and c have been assigned in memory without cache conflicts as shown in Figure 2(a). If some data block d is accessed by caches C0 and C1, however, no conflict-free assignment is possible. In this case, either P0 or P1 will have to wait until the l lines of the data

Marios Papaefthymiou is supervised by Prof. C. E. Leiserson. This research was supported in part by NSF under grants MIP 9012773 and 9115797-CCR and in part by the Defense Advanced Research Projects Agency monitored by the Office of Naval Research under grants N00014-91-J-1698 and N00014-89-J-1988. Authors' e-mail addresses are marios@theory.lcs.mit.edu, agarwal@lcs.mit.edu, guttag@lcs.mit.edu.

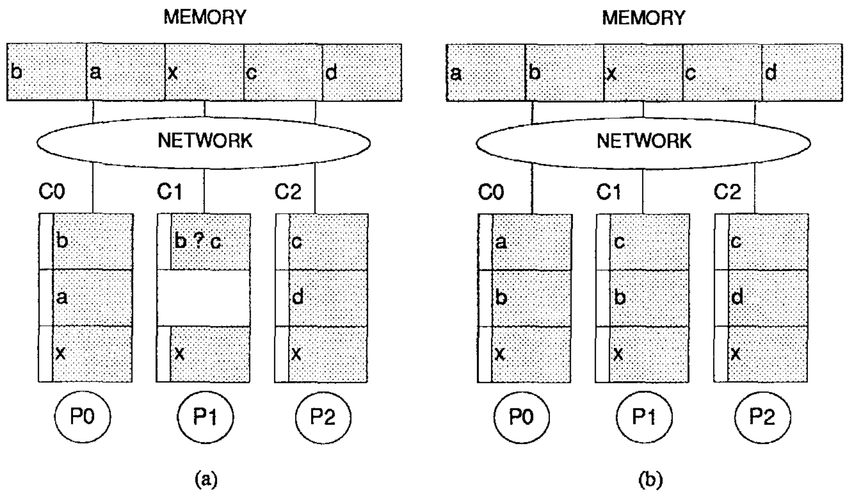


Figure 1: (a) A conflicting assignment. (b) A conflict-free assignment.

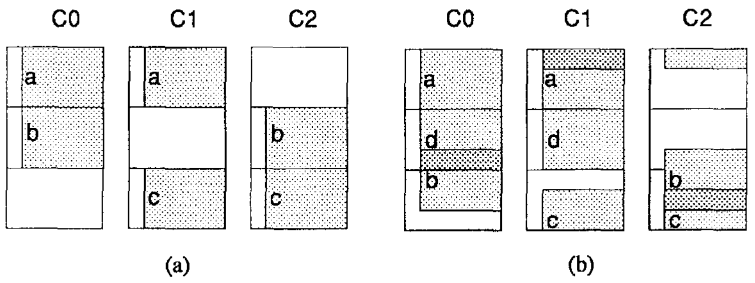


Figure 2: (a) A situation where a conflict-free assignment is impossible if, for example, caches C0 and C1 access a data block d. Any non-overlapping assignment of d in memory will result to l conflicting lines in some cache. (b) A partially overlapping assignment obtained after grey coloring. The maximum number of misses in any cache is at most $l/3$ lines.

block are read from the main memory. By allowing data blocks to partially overlap, however, we can find an assignment that results to at most $l/3$ conflicting lines per cache as shown in Figure 2(b). Our procedure operates in two phases and runs in $O(D^3)$ steps. We begin by encoding the dependencies among the data blocks in a *conflict* graph G, such that a coloring of G with k colors yields automatically an assignment of memory locations to the data blocks. The first phase of our strategy computes a k-coloring for a maximal subset of G. If G does not have any particular structure that allows us to color it efficiently, we apply a minor variation of the general graph-coloring scheme that is used in the context of register allocation for sequential processors. Any vertices that are left uncolored correspond to ***unassigned blocks***, which are placed in memory during the second phase of our procedure. The goal of this phase, which we call *grey coloring*, is to insert the unassigned blocks in a way that minimizes the number of conflicts in any single cache. The key idea is to place blocks in such a way that they straddle cache slots. Since cache slots contain parts of multiple blocks, they can be thought of as having a mixture of colors, hence the name grey coloring.

References

[1] A. Agarwal, J. Guttag, M. Papaefthymiou. "Reduced-Latency Memory Assignment for Multiprocessor Caches," unpublished manuscript, February 1992.

Performance Assertion Checking

Sharon E. Perl*

MIT LCS, NE43-520A
sharon@lcs.mit.edu

Systems often develop performance bugs that go unidentified for long periods of time. By *performance*, I mean some measure of resource usage in the system. A *performance bug* is a failure of the system to meet the performance expectations of its implementors. *Performance assertion checking* is an approach and a related set of tools for performance debugging and testing that addresses the problem of uncovering performance bugs in a timely fashion [1]. The key idea is to have implementors write down their performance expectations precisely, and in a way that permits automatic checking. Information about a program's execution that is relevant to its performance is captured in a monitoring log, generated when the program runs. The user of the tools writes a *performance specification*, consisting of a set of *performance assertions* which are predicates that are expected to hold for the monitoring log. The specification is expressed in the PSpec performance specification language.

Performance assertions provide a means of filtering large quantities of performance data to focus attention on the data that indicate potential problems in a system. This is useful for several kinds of performance-related activities:

- Performance regression testing: once the performance of a program is understood, it can be captured with a set of performance assertions. When the system is changed, the assertions can be rechecked to ensure that the performance still meets expectations.
- Continuous system monitoring: performance assertions can be checked routinely during normal system use to determine whether performance is meeting expectations and whether workloads satisfy the assumptions that were made during system design.
- Performance debugging: successively more detailed performance assertions may be helpful for pinpointing the location of performance problems in the system.

In addition, the act of writing performance assertions forces an implementor to think clearly and precisely about the performance of the system.

The PSpec language is based on the notion of a monitoring log as a sequence of primitive components called *events*. An event has a type (name), a list of named, numeric-valued attributes, and, possibly, a timestamp. A specification writer identifies the event types of interest for a particular specification; these events have a direct correspondence with the events appearing in a monitoring log.

While events contain useful information, often it is necessary to work with subsequences of events in a log when writing assertions. For example, we may be interested in writing assertions about the elapsed time between two events that delimit an operation in the program, or we may be interested in checking whether some particular event occurs during an operation (perhaps a cache hit during a file system read operation). For this reason the PSpec language has the notion of an *interval*.

An interval corresponds to a subsequence of a log starting at some start event, ending at some end event, and including all events in the log between them. Like events, all intervals are of some named interval type. Just as an event type has named attributes, an interval type has named *metrics* that record values of interest (not neces-

*This research was supervised by Prof. William E. Weihl and was supported by the NSF under grant CCR-8716884, by DARPA under contract N00014-89-J-1988, and by Digital Equipment Corporation.

sarily numeric) for intervals of the type. Metrics are computed from the events that comprise an interval. While the set of event types available in a specification is determined by the contents of a log, the specification writer has complete freedom to declare whatever interval types and associated metrics are of interest, using the available event types.

Performance assertions in PSpec are then predicates over the set of events and intervals in a log.

As an example of a performance specification, suppose we would like to write an assertion about the time during which interrupts are disabled on any processor in a multiprocessor. In particular, suppose we would like to express the assertion that "interrupts are disabled for at most 50 cycles." Figure 1 shows how we can do this in PSpec. First, we introduce events, `IntOff` and `IntOn`, corresponding to the disabling and enabling of interrupts on a processor. Each of these events has a timestamp and an attribute, *pid*, recording the processor number. Using these event types, we define an interval type corresponding to an interval in a log during which interrupts are disabled. We declare an interval of type `IntDisabled` to start with an event of type `IntOff` and to end with the next event in the log after the start event of type `IntOn` where the processor id of the end event matches the processor id of the start event. Each `IntDisabled` interval is declared to have a metric, **time**, whose value is obtained by subtracting the timestamp of its start event from the timestamp of its end event. Then we can express the desired assertion using this interval type. The assertion, shown in the figure, can be read as: "for all intervals `i` of type `IntDisabled` in the log, the value of the **time** metric for `i` is at most 50 cycles."

Performance specifications and monitoring logs are input to two tools. The first, a *checker* program, takes a performance specification and a monitoring log from a program run, and reports which assertions failed to hold for the run. The second, a *solver* program, takes a specification with symbolic constants whose values are unknown and a monitoring log from a program run, and estimates values for the unknowns using linear regression based on the data in the log. The output of the solver is the specification with the unknowns bound to their estimated values, which can then be input to the checker.

```
timed event IntOff (pid);
            IntOn (pid);
interval IntDisabled =
  s: IntOff,
  e: IntOn where e.pid = s.pid
 metrics
  time = timestamp(e) - timestamp(s)
 end IntDisabled;
assert {& i : IntDisabled
          : i.time <= 50 cyc};
```

Figure 1: Example performance specificiation.

The generation of monitoring logs can be accomplished with whatever monitoring tools are available for the system whose performance is of interest. The PSpec notion of a log is fairly general. To make the PSpec tools work with a new log format, one need only implement the module that presents this simple log abstraction and which understands the log format.

The PSpec language design is currently in its second iteration. I implemented both the checker and solver for an earlier version of the language, and used the tools to write and check performance specifications for pieces of the language run-time system of Prelude, the new parallel programming language being developed in the Large-scale Parallel Software Group at MIT. The experiments with Prelude were interesting on two counts. They led to ideas for the redesign of the PSpec language, to make it more general and more useful. Also, several performance bugs in the Prelude run-time were found by checking some simple performance assertions concerning the amount of time that interrupts are disabled and the amount of time required to send messages between processors; the Prelude implementors were unaware of these bugs prior to the experiments.

References

[1] Sharon E. Perl. *Performance Assertion Checking*. Ph.D. thesis, Massachusetts Institute of Technology, expected September 1992.

PROJECT
NUMESH

NuMesh CFSM Rev2

John S. Pezaris[1]
MIT Room NE43-616
pz@mit.edu

The NuMesh is a novel approach for constructing scalable heterogeneous multiprocessor systems via a standardized interface as embodied in the Communications Finite State Machine (CFSM). A topology, with large-scale implementation strategy and mechanical packaging, has been chosen for the NuMesh that is isomorphic to the crystal structure of diamond. This paper presents a proposed architecture for the newest revision of the CFSM which includes support for single-cycle static routing and low-latency dynamic routing.

A three-dimensional four-neighbor topology is constructible from rectangular printed circuit boards on which four connectors have been placed, one at each edge with opposite-edge connectors facing in the same direction, one pair facing up, the other facing down. When stacked appropriately, these modules form a three-dimensional tetrahedral mesh with interesting logical and physical characteristics. Logically, the lattice is isotropic and homogeneous; physically, it contains horizontal channels running the extent of the mesh and a complex air path from upper to the lower faces. This topology is, as far as we know, unique to computer science, and minimal in the number of ports.

The CFSM Rev2 architecture centers around a five-port 32-bit crossbar controlled by five synchronous but loosely coupled programmable finite state machines (FSMs). Each of the FSMs is associated with an I/O port: four are designed to support the inter-CFSM routing, while the fifth is intended to interface to a local processing element. Further, each has a small register file, wired-or General Outputs with which communications between FSMs is effected, and similar Control Outputs and Inputs over which negotiations between neighboring nodes occur. The FSMs support three classes of instructions: dispatch on condition (such as data presence, General Output, etc.), set registered value (such as crossbar connection), and assert signal for n cycles (such as write enable to the output register, or load program RAM).

The two-port register file (one read, one write) contains the program counter (PC) and instruction register (IR) to provide easy access to the control path from the data stream and vice versa. All registers may be read onto the crossbar or read from it; register R0 drives the output port. Each register, save the PC and IR, are decrementable counters with all-zeros detection. These last two features allow increased code density by directly implementing looping constructs.

The I/O ports are bidirectional, save for the Control Inputs and Outputs. Because there is only one transmitter-receiver pair on each line, pad and connector design is tightly constrained and can be highly optimized. Special circuitry is included to insure that two neighbors never simultaneously drive their common I/O lines.

By placing part of the control path in the data path, a strong limitation of previous revisions is overcome, namely the inability to have the data stream affect the control stream. Specifically, it is now possibile to embed control information within the data stream. This could be used, for example, to send a section of code as routing information within the header of a message, or, in other terms, to send the complete pair of a Turing machine and its input as a message.

Many communication modes can be efficiently supported on the Rev2 architecture. Although the standard NuMesh model relies heavily on static routing, careful consideration has been given in the design of CFSM Rev2 toward supporting low latency dynamic routing. Taken in increasing order of complexity: Fully synchronous static routing takes 1 cycle per node; synchronous fanin (single reader, time-multiplexed writers) and fanout (single writer, multiple readers) takes 1 cycle; asynchronous static-graph routing takes 2 or 3 cycles; limited-decision dynamic routing takes 2 cycles per comparison with 2 cycles of overhead; fully-dynamic routing based on run-time conditions are unpredictable by nature and therefore no evaluations can

[1] The author is a member of Team NuMesh, a part of the MIT Computer Architecture Group. This research is supervised by Professor Stephen A. Ward. Partial support has been provided by DARPA, Texas Instruments, and AT&T. Patents pending.

be made.

The architectural design has been completed. Circuit schematics are anticipated by September 1992; Functional silicon within a year.

Bibliography

F. HONORÉ, *The Next CFSM: Revision 1 Description*, NuMesh Memo 14, Computer Architecture Group, Lab for Computer Science, MIT, Cambridge, MA, August 1991.

K. MACKENZIE, *NuMesh prototype hardware description*, NuMesh Memo 1, Computer Architecture Group, Lab for Computer Science, MIT, Cambridge, MA, June 1990.

J. S. PEZARIS, *CFSM rev 2: Progress to Date*, NuMesh Memo 9, Computer Architecture Group, MIT Lab for Computer Science, Cambridge, MA, May 1991.

J. S. PEZARIS, *CFSM Desiderata*, NuMesh Memo 12, Computer Architecture Group, MIT Lab for Computer Science, Cambridge, MA, July 1991.

G. A. PRATT, *et al*, *The Diamond Interconnect*, In Process, 1992.

S. A. WARD, *et al*, *The NuMesh: A Scalable, Modular 3D Interconnect*. In process, 1992.

S. A. WARD, J. NGUYEN, J. S. PEZARIS, G. A. PRATT, *3D-4N Meshes*, NuMesh Memo 18, Computer Architecture Group, Lab for Computer Science, MIT, Cambridge, MA, September 1991.

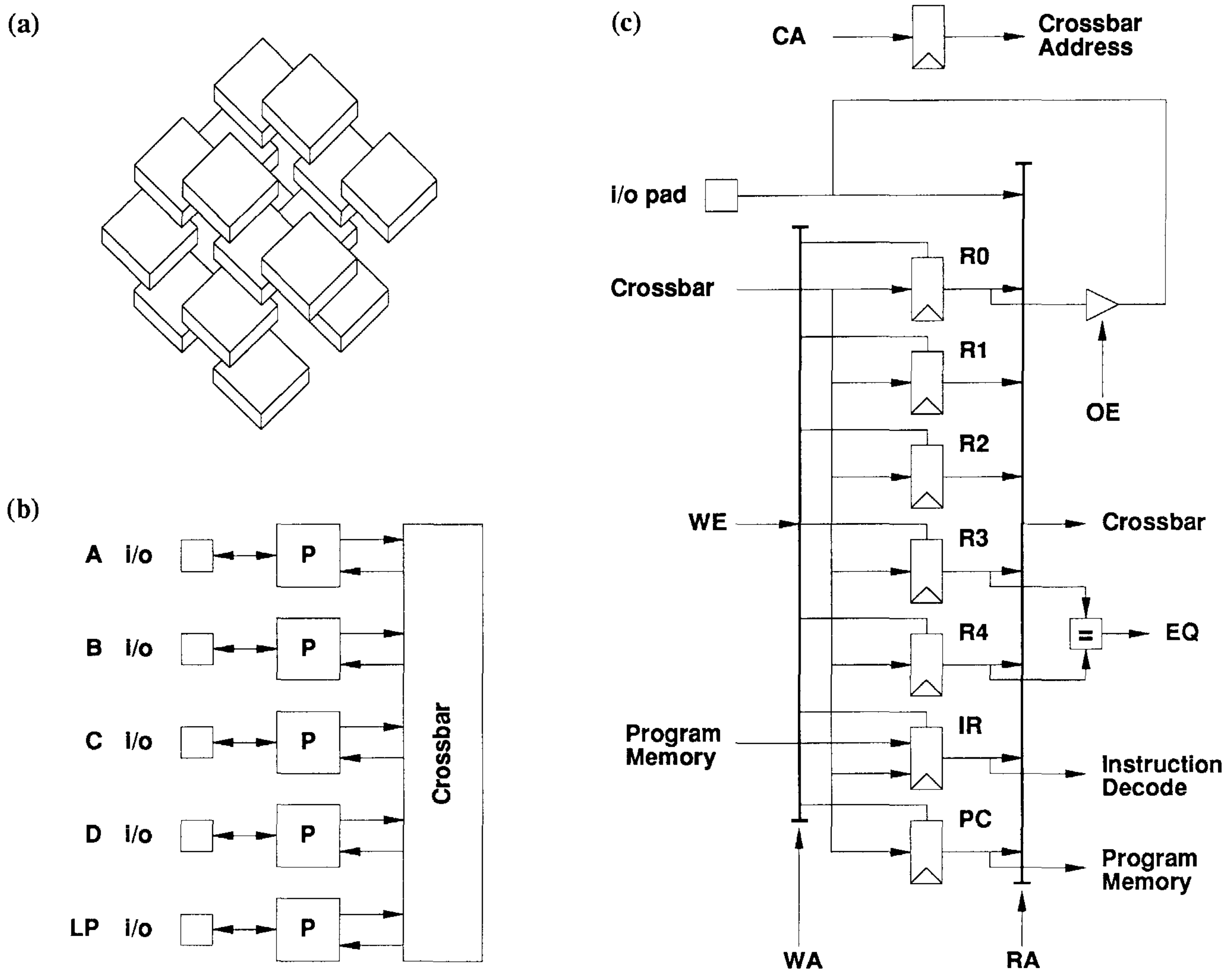


Figure 1: **(a)** Diamond lattice as implemented with printed circuit cards and vertical connectors; **(b)** Block diagram of CFSM Rev2, illustrating the four routing ports **A–D** and the local processor port **LP**; **(c)** Schematic diagram of a single FSM from a Rev2 CFSM, found within one of the **P** boxes from (b).

NUMERICAL STUDIES OF VORTEX DYNAMICS IN JOSEPHSON JUNCTION ARRAYS*

JOEL R. PHILLIPS†

In the past decade there has been considerable interest in the physics of artificially fabricated Josephson junction arrays. In particular such arrays provide a useful model system for high-T$_c$ superconductors[6]. In this paper we describe the numerical study of vortex excitations in a Josephson junction array.

A Josephson junction may be fabricated by sandwiching a thin insulating layer between two superconductors. The junction may support a supercurrent of

$$I_b = I_c \sin\phi$$

where the gauge-invariant phase difference ϕ contains a contribution from the quantum-mechanical phases of the superconducting islands, and from the magnetic field. An array of Josephson junctions consists of a regular lattice of islands of superconductor connected by Josephson junction.

A vortex is an excitation in the phase (ϕ) configuration of the array. The definition of a vortex is that, in the absence of magnetic fields, the sum of the phase differences around any loop formed by the junctions of the array is 2π times the number of vortices contained by the loop. The magnetic field contributes a term

$$\Phi\frac{2e}{\hbar}$$

to the loop-sum of the phases, where Φ is the flux through the loop. Thus, the system can be considered as a sort of nonlinear circuit, with phase analogous to voltage. Vortices and magnetic fields act as voltage sources.

We can analyze this network using mesh analysis[1]. We use the mesh matrix M to express the "voltage law"

$$M\phi = -\Phi\frac{2e}{\hbar} + 2\pi n\delta_{m,v}$$

where the Kronecker delta indicates the number n of vortices in a cell, and ϕ is now to be understood as the vector of phase-differences.

We further define a vector of mesh currents I_m which are related to the branch (junction) currents by

$$M^T I_m = I_b$$

The next question is the computation of the magnetic flux Φ. We separate this into two parts, the flux from an externally applied field Φ_{ext} and the flux Φ_{ind} induced by the currents in the array. The flux Φ_{ind} can be calculated given the current distribution

$$\Phi_{ind}\frac{2e}{\hbar} = MLI_b = MLM^T I_m$$

where the matrix L is the standard partial-inductance matrix[4]. We can interpret the matrix L as a set of dependent "voltage" sources; every junction i contributes a voltage $L_{ij}I_i$ across junction j. The matrix MLM^T will generally be dense, as the mesh current in every cell contributes to the flux through every other cell. This fact will present the main computational difficulty in these calculations.

We can now write the full system of equations to be solved

$$M\phi + MLM^T I_m + \Phi_{ext}\frac{2e}{\hbar} - 2\pi n\delta_{m,v} = 0$$

$$M^T I_m - I_c \sin\phi = 0$$

These nonlinear equations are solved using Newton's method. The computation can be reduced to solving a series of linear systems of the form

$$M(D+L)M^T x = b$$

where L is the partial-inductance matrix previously discussed, and D is a diagonal matrix. This (dense) linear system could be solved using direct Gaussian elimination, which, for an $N \times N$ array, would have memory requirements growing as N^4 and computation time growing as N^6. The memory requirements of forming MLM^T make the study of arrays of more than moderate ($N \simeq 40 - 50$) size impractical on a typical scientific workstation when a direct method is used to solve the equations. The obvious alternative is a conjugate-residual type iterative method[2], which will not require storage

* This work was supervised by Prof. J. K. White and Prof. T. P. Orlando. J. Phillips acknowledges support from an NDSEG fellowship.

† Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Rm 36-886, Cambridge, MA 02139. E-mail: *jphill@rle-vlsi.mit.edu*

ALGORITHM 1 (CONJUGATE-RESIDUAL ALGORITHM FOR SOLVING $Ax = b$).

```
guess $x^0$
repeat {
  Compute the error, $r^k = b - Ax^k$
  Find $x^{k+1}$ to minimize $r^{k+1}$
  k = k + 1
} until $r^k$ small
```

of the matrix, only the computation of matrix-vector products.

The conjugate residual algorithm can be accelerated by applying it to the transformed problem

$$PAx = Pb \qquad P \simeq A^{-1}$$

If the *preconditioning matrix* P is close to A^{-1}, the conjugate residual algorithm will converge in few iterations. If in addition the cost of computing P is small, the resulting algorithm will be fast.

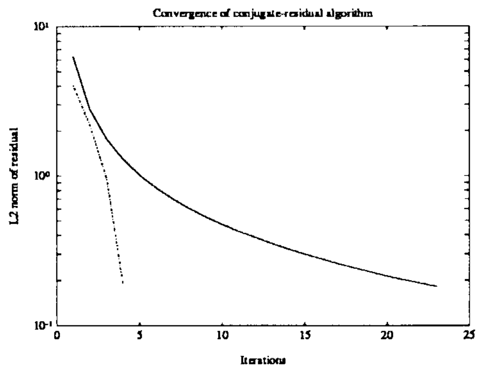


FIG. 1. *Effect of preconditioner on conjugate-residual algorithm. Dashed line shows convergence of preconditioned problem*

To motivate the selection of an effective preconditioner, consider the case of weak self-field effects, where the elements of L are significantly smaller than those in D. It is found upon inspection that the system of equations

$$MDM^Tx = b$$

is structurally identical to the five-point finite-difference discretization of Poisson's equation. Since such a system can be very efficiently solved by use of the fast Fourier transform[5], we suspect it would make an effective preconditioner. For moderate self-field effects (Φ_{ind} small), this preconditioner speeds up the computation by roughly a factor of five, as can be seen in Figure 1.

Once we have obtained vortex solutions to the network equations, physically relevant quantities, such as the vortex energy as a function of position, driving current, and external field, can be easily obtained. A typical single-vortex solution is shown in Figure 2.

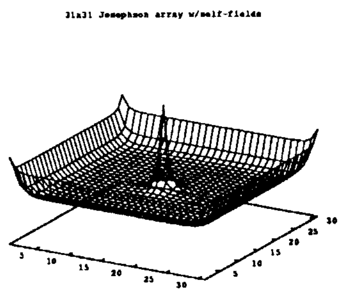


FIG. 2. *Vortex in array with externally applied field. Self-field effects are observed in the screening currents flowing at the edges.*

To model dynamics of vortices, we must add time-dependent terms to the Josephson current relation

$$I_b = I_c \sin\phi + \beta\frac{d^2\phi}{dt^2} + \Gamma\frac{d\phi}{dt}$$

After time-discretization, the addition of these terms will only modify the numerical values of the entries in the matrix D, so that the numerical issues are essentially unchanged.

Future numerical work will focus on using a fast multipole algorithm[3] to compute the matrix-vector products in the conjugate-residual algorithm. This algorithm should allow simulations to be performed in time linear in the number of mesh elements.

REFERENCES

[1] C. A. DESOER AND E. S. KUH, *Basic Circuit Theory*, McGraw-Hill, 1969.

[2] H. C. ELMAN, *Iterative methods for large, sparse, non-symmetric systems of linear equations*, PhD thesis, Yale University, 1982.

[3] L. GREENGARD AND V. ROKHLIN, *A fast algorithm for particle simulations*, Journal of Computational Physics, 73 (1987), pp. 325–348.

[4] A. E. RUEHLI, *Inductance calculations in a complex integrated circuit environment*, IBM Journal of Research and Development, 16 (1972), pp. 470–481.

[5] G. STRANG, *Introduction to Applied Mathematics*, Wellesley-Cambridge Press, 1986.

[6] M. TINKHAM AND C. J. LOBB, *Physical properties of the new superconductors*, Solid State Physics, 42 (1989), pp. 91–134.

An Iterative Approach for the Solution of the Boltzmann Transport Equation for Semiconductors

Khalid Rahmat
Research Laboratory of Electronics
Massachusetts Institute of Technology
rahmat@rle-vlsi.mit.edu

One of the major concerns in the design and fabrication of ultra-short-channel semiconductor devices (with channel lengths as short as a few hundred nanometers) is the effect of the high electric fields that are generated in the device. The steeply graded electric field profile imparts very high energy to carriers (electrons and holes) in a narrow region of the device. These energetic carriers can have a number of deleterious effects on device performance and reliability. These phenomena are usually addressed under the rubric of hot carrier effects and are primarily a reliability issue. For example, in a MOSFET which undergoes hot carrier degradation, device parameters such as threshold voltage, transconductance and current can change substantially over the device lifetime. Thus a circuit designed with certain nominal MOSFET parameters may fail if the device characteristics change significantly as a result of device operation. Hot carrier effects are also a major design constraint on optimizing the design of short-channel devices as the device performance can be traded-off for greater immunity to hot carrier degradation.

Computer programs that simulate the behavior of semiconductor devices have been successfully used to design and optimize devices for a number of years. Unfortunately almost all such programs are incapable of correctly simulating the hot carrier effects outlined above as they only solve for average quantities such as electron concentration and average velocity and energy but provide no details about the high energy carriers. This failure stems from the simplified physics which is incorporated in these programs.

The underlying mathematical description for transport in semiconductors is the Boltzmann transport equation (BTE):

$$\frac{\partial f}{\partial t} + \frac{qF}{\hbar} \cdot \frac{\partial f}{\partial k} + v \cdot \frac{\partial f}{\partial r} = S[f] \qquad (1)$$

Faculty Supervisor: Prof. Jacob White

This work was supported by the Defense Advanced Research Projects Agency contracts N00014-87-K-825 and MDA972-88-K-008, and grants from I.B.M.

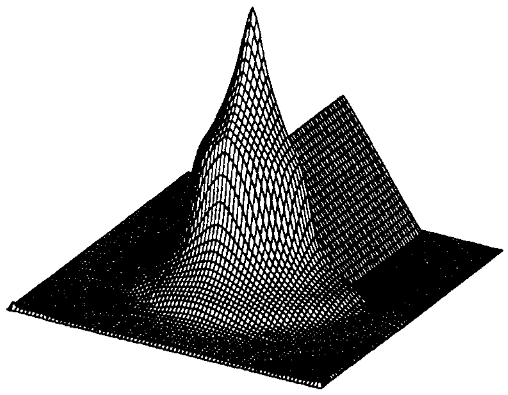


Figure 1: Steady state distribution from an arbitrary initial distribution.

Solution of the BTE yields the distribution function, f, for the carriers (electrons and holes) in momentum (k) and real space (r) given a field, F and the knowledge of the scattering mechanisms to be included in the right hand side and denoted by the scattering operator, S. Then all physical quantities of interest, such as electron density, current, energy etc. can be determined from the distribution function. As the BTE in general is an integro-differential equation (the scattering operator involves integrals of f in k-space) in six-dimensions and time it is extremely difficult to solve. Thus, only an approximate solution is obtained in device simulation programs. For detailed solutions the BTE has been solved using a Monte-Carle method [1]. This approach is stochastic in nature and is equivalent to calculating the detailed path of tens of thousands of electrons over the simulation interval. The Monte-Carlo method can in principle yield the detailed distribution function but is extremely expensive computationally. Moreover, rare events such as high energy phenomena are very difficult to compute due to the stochastic nature of the Monte-Carlo method.

An alternative to the above approaches is the direct solution of the BTE. One way of doing this is to write the BTE as a purely integral equation [2]. This form can be thought of as separating the carrier transport into two problems: scattering and free-flight. The scattering operator involves an integral in momentum space only, while the free-flight operator updates the distribution function in real space after integrating over a time step, $1/\Gamma$. Due to the implicit nature of the integral equation, it can only be solved using an iterative approach, where the iterations correspond to stepping in time. Thus, given an initial distribution function, first the scattering operator is used to generate a new function, g, which incorporates only the effects of the scattering:

$$g^n(k) = \Gamma f^n(k) + \int S(k',k) f^n(k') dk' - f^n(k) \int S(k,k') dk' \tag{2}$$

This new function is then used as the right hand side of Eqn. 1 which can then be integrated to give:

$$f^{n+1}(k) = \int_0^\infty d\eta \, e^{-\Gamma\eta} g^n(k - \frac{qF}{\hbar}\eta) \tag{3}$$

which results in a new distribution function corresponding to its evolution in time. For the steady state case the iterations will converge and the (n-1)th and nth iterations will coincide. To include variation in real space the second equation can be modified and requires an integral in real space also.

For the homogeneous case (i.e. with no spatial variation) the above equations have been implemented in a simple simulator. Only the two most important types of scattering are currently included: acoustic phonon (elastic) and optical phonons (inelastic). Starting from an arbitrary initial distribution with no applied fields the distribution function should reach its equilibrium value after a few time steps as shown in Fig. 1.

In this figure the distribution function is plotted assuming cylinderical symmetry, thus f is considered to be a function of v_z and v_ρ. The top half of the figure shows the initial distribution function which was chosen to be a prism shape whereas the bottom half shows the distribution function after some time steps. Clearly the distribution is now much more isotropic and has roughly a gaussian shape. This is more clearly visible in Fig. 2 where the distribution function is plotted as a function of v_z for a fixed v_ρ.

The effect of an applied field is shown in Fig. 3 where we start from the the distribution obtained in Fig.1 and turn on a constant electric field in the z direction. The distribution function after a few time steps is shifted along the direction of the applied field (ignoring the sign of the electronic charge).

Currently the program is being used as a test-bed to identify the features that are critical in generating

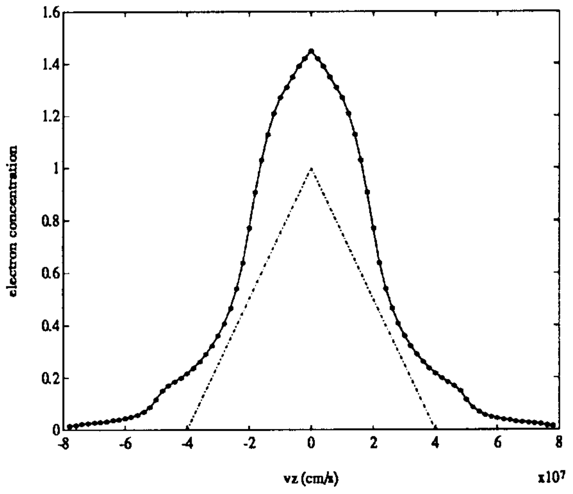


Figure 2: The initial and final distribution of Fig.1 plotted as a function of v_z for a fixed v_ρ.

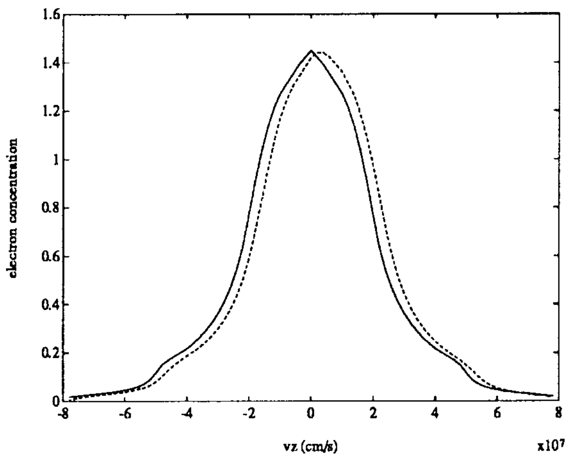


Figure 3: Effect of an applied field on the distribution.

a direct solution to the BTE. Further work will include spatial variation and will require the solution of Poisson's equation consistently with the BTE. Also, extensions to higher dimensions as well as more efficient representations for the distribution function such as basis function expansions are being studied.

References

[1] C. Jacoboni and L. Reggiani, "The Monte-Carlo method for the solution of charge transport in semiconductors with applications to covalent materials", Rev. Mod. Phys, vol. 55, no. 3, p.645, 1983.

[2] H. D. Rees, "Computer simulation of semiconductor devices", J. Phys. C: Solid State Phys, vol. 6, p. 262, 1973.

Edge-Triggering vs. Level-Clocking

Keith H. Randall   Marios C. Papaefthymiou
MIT Laboratory for Computer Science
Cambridge, MA 02139

Synchronous circuits that implement clocked storage elements using level-sensitive latches, instead of the more conventional edge-triggered latches, are becoming increasingly popular. An edge-triggered latch updates its state on the rising edge of its clock input and directly supports the abstraction of a storage element that is synchronized by the tick of a clock. The operation of level-sensitive latches is somewhat different. While the clock input of a level-sensitive latch is low, the latch maintains its value from the last time the clock was high. While the clock input of the latch is high, however, the latch becomes transparent and allows data to flow unimpeded from the input to the output. Level-clocked circuits have the potential to operate faster, in theory, than edge-triggered circuits. In this paper, we develop a methodology for comparing edge-triggered and level-clocked implementations of synchronous circuits, and we investigate under what circumstances and to what extent we can achieve this theoretical improvement in practice. Our experiments indicate a tradeoff between the speedups achieved by level-clocking and the degree of pipelining in a circuit. Specifically, edge-triggering is just as good as level-clocking for circuits with either too few or too many pipeline stages. When the degree of pipelining is between the two extremes, however, the benefits of level-clocking begin to emerge. We give a heuristic to identify the circuits that are likely to improve by level-clocking. We observe, based on our experiments, that circuits with more uniform delays tend to improve more by level-clocking. We also observe that there is no apparent advantage to clocking level-clocked circuits with asymmetric phases.

Implementation ease has made edge-triggered circuits particularly popular among designers. Level-clocked circuits have the potential to operate faster than edge-triggered circuits, however, because they allow computations to extend beyond a single clock cycle. The potential of level-clocked circuits to operate faster than edge-triggered circuits comes at the cost of increased complexity both at the design and at the implementation level. The operation of level-clocked circuits is not as intuitive as that of edge-triggered circuits, and it is more difficult to argue about their timing. Additional layout difficulties arise due to the multiple clocks that must be distributed across the chip. It is virtually impossible to quantify these difficulties. Our work, however, aims at providing the circuit designer with information that allows him to decide whether the switch to a level-clocked circuit is probably worth the additional effort, or whether more conventional solutions work just as well.

Our methodology for comparing edge-triggered and level-clocked circuits consists of the following three steps. First, we retime the edge-triggered circuit so that it achieves the minimum period possible. Subsequently, we convert the edge-triggered circuit into a level-clocked circuit by replacing each edge-triggered latch by a pair of level-sensitive latches that are clocked on opposite phases of a two-phase, nonoverlapping clock. We retime this level-clocked circuit in order to achieve the minimum period possible when the active times of the two clocking waveforms are equal. Finally, we further retime the level-clocked circuit while simultaneously tuning the active times of the two clocking waveforms, so that we achieve the minimum period possible under any retiming and under any two-phase, nonoverlapping clock. We repeat these three procedures for pipelined versions of the original circuit. Each pipelining of the original circuit is obtained by multiplying the initial number of latches on its edges by an integer constant. We also apply our three procedures on pipelined versions of the original circuit with more uniform delays. We vary the original gate delays, that were assigned by the `lib2` library in `sis`, by raising them to the same power $p < 1$. As p decreases, the gate delays approach 1, and the clock period depends on the propagation delay along more paths in the circuit.

A sample of our experimental results for the circuit `mult16a` from the MCNC benchmark is illustrated in Figure 1. The data points were obtained for the original gate delays, and for gate delays obtained by

This research was supervised by Prof. C. E. Leiserson and supported in part by the Defense Advanced Research Projects Agency under Grant N00014-91-J-1698. Authors' e-mail addresses are randall@theory.lcs.mit.edu., marios@theory.lcs.mit.edu

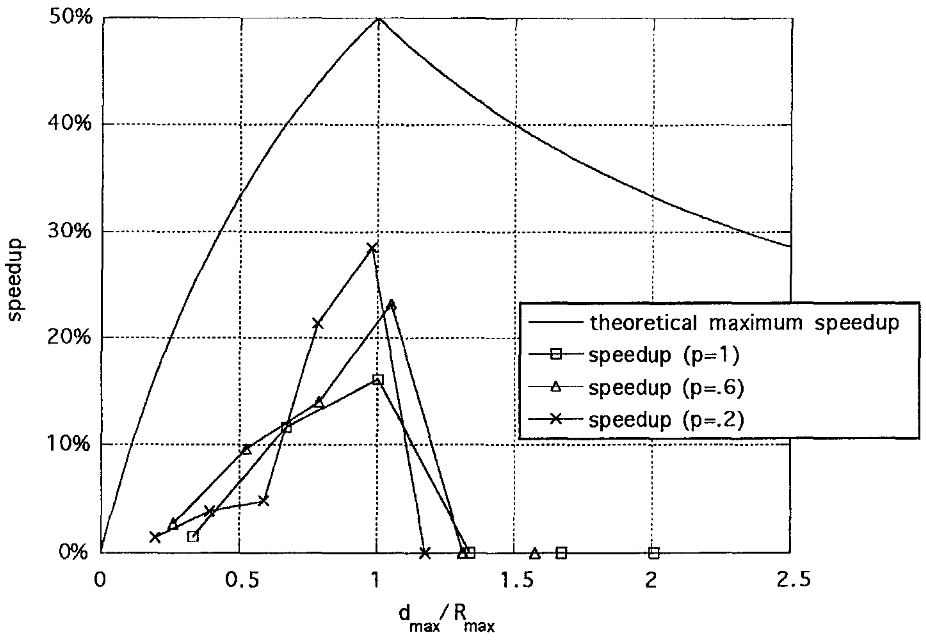


Figure 1: Relative speedup achieved by level-clocking over edge-triggering as a function of the ratio $d_{\max}/R_{\max}$. Data were obtained on the circuit **mult16a** from the MCNC benchmark.

raising the original gate delays to the powers $p = 0.6$ and $p = 0.2$. For each circuit, the figure shows the relative speedup achieved by level-clocking over edge-triggering as a function of the ratio $d_{\max}/R_{\max}$, where $d_{\max}$ is the maximum gate delay and $R_{\max}$ is the maximum ratio of total gate delay and total number of latches around the cycles in the circuit. Higher values of the ratio $d_{\max}/R_{\max}$ correspond to higher degrees of pipelining of the original circuit. A relative speedup of 16% is achieved with the original gate delays and pipelining of the original circuit by a factor of three. A 28.5% speedup is obtained for $p = 0.2$ and pipelining by a factor of five. For all three delay configurations, the maximum speedup is obtained when the maximum gate delay $d_{\max}$ is almost equal to the maximum delay-to-latches ratio $R_{\max}$ in the circuit.

Our experiments indicate that edge-triggering is just as good as level-clocking for low degrees of pipelining in the circuit. As we increase the degree of pipelining, however, the benefits of level-clocking begin to emerge. When we increase the degree of pipelining after a certain point, then the advantages of level-clocking suddenly disappear, and edge-triggering becomes again as good. Our experiments also indicate that the advantages of level-clocking are more apparent in circuits with more uniform propagation delays. We give a heuristic criterion that identifies whether a circuit is likely to improve with level-clocking, by examining the maximum gate delay $d_{\max}$ and the maximum delay-to-latches ratio $R_{\max}$ in the circuit. According to our criterion, the closer $d_{\max}$ is to $R_{\max}$ the more likely it is for the circuit to improve by level-clocking. Our criterion agrees with our experimental results. Moreover, the advantage of level-clocking disappears suddenly precisely when $d_{\max}$ exceeds $R_{\max}$. In our experiments, we observed no instance where simultaneous retiming and tuning led to faster level-clocked circuits than retiming with symmetric clocks.

Our work should not be viewed only as a practical demonstration of the potential speedups that can be achieved by level-clocking over edge-triggering. We have also presented a criterion, expressed in terms of the characteristic parameters $d_{\max}$ and $R_{\max}$ of a circuit, for identifying circuits that are likely to improve by level-clocking. Moreover, our empirical results suggest a design style for generating circuits that are likely to be faster when level-clocked. These circuits will be designed with standard cells of uniform delay, and the delay-to-latches ratio around their cycles will be roughly equal to the longest gate delay. The original design can be edge-triggered, a domain where design is more simple and intuitive. The final level-clocked circuit will be generated automatically using the tools that have been developed for optimizing level-clocked circuits.

References

[1] M. C. Papaefthymiou and K. H. Randall "Edge-Triggering vs. Level-Clocking," unpublished manuscript, April 1992.

WAVEFORM FREQUENCY-DEPENDENT OVERRELAXATION FOR TRANSIENT TWO-DIMENSIONAL SIMULATION OF MOS DEVICES[*]

MARK REICHELT[†]

This abstract presents a new waveform frequency-dependent overrelaxation algorithm and its application to solving the differential-algebraic system generated by spatial discretization of the time-dependent semiconductor device equations.

Device transient simulation is usually performed by numerically solving the coupled Poisson and time-dependent electron and hole current-continuity equations with a low-order implicit time-integration scheme, combined with a Newton or relaxation method to solve the generated sequence of nonlinear algebraic equations [1]. Another approach is to apply WR and standard overrelaxation acceleration (WSOR) to the equation system, as given in Algorithm 1 [4, 2, 3]. Though *fast and parallelizable*, the ordinary WSOR algorithm can unfortunately produce oscillatory results, even with a carefully chosen overrelaxation parameter, as illustrated in Figure 1.

ALGORITHM 1 (ORDINARY WSOR).

```
guess $u^0, n^0, p^0$ waveforms at all nodes
for $k = 0, 1, \ldots$ until converged
  for each node $i$
    solve for $u_i^{k+1}, n_i^{k+1}, p_i^{k+1}$ waveforms
      (like Gauss-Seidel WR):
        $f_{1_i}(u_i^{k+1}, n_i^{k+1}, p_i^{k+1}, u_j^k) = 0$
        $\frac{d}{dt} n_i^{k+1} + f_{2_i}(u_i^{k+1}, n_i^{k+1}, u_j^k, n_j^k) = 0$
        $\frac{d}{dt} p_i^{k+1} + f_{3_i}(u_i^{k+1}, p_i^{k+1}, u_j^k, p_j^k) = 0$
    overrelax $u_i^{k+1}, n_i^{k+1}, p_i^{k+1}$ waveforms
      $x_i^{k+1} \leftarrow x_i^{k+1} + \alpha \cdot [x_i^{k+1} - x_i^k]$
```

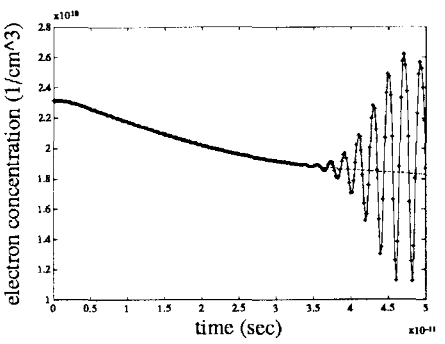


FIG. 1. *The ordinary waveform SOR algorithm for device simulation and a plot of the electron concentration vs. time at a channel node of the* **karD** *example, showing the WSOR frequency amplification.*

To derive a more reliable acceleration, the effect of each iteration of Gauss-Seidel WR can be represented abstractly as

$$x^{k+1} = F(x^k) \quad \text{where} \quad F : C^1 \to C^1, \tag{1}$$

C^1 is the space of continuously differentiable functions, and $\boldsymbol{x}^k(t) = [\boldsymbol{u}^k(t), \boldsymbol{n}^k(t), \boldsymbol{p}^k(t)]^T$. Then if $\Delta^k(t) = \boldsymbol{x}^{k+1}(t) - \boldsymbol{x}^k(t)$ is small, a linearization of equation (1) followed by Fourier transformation yields

$$\Delta^{k+1}(\omega) = G(\omega)\, \Delta^k(\omega) \tag{2}$$

where $G(\omega)$ is the Fourier transform of the linearized F. The WR algorithm can then be accelerated in the frequency domain by overrelaxing:

$$x_i^{k+1}(\omega) \leftarrow x_i^{k+1}(\omega) + \alpha(\omega) \cdot [x_i^{k+1}(\omega) - x_i^k(\omega)] \tag{3}$$

* This work was supervised by Prof. Jacob K. White and Prof. Jonathan Allen and supported by DARPA contract N00014-91-J-1698, NSF contract MIP-8858764 A02, and a grant from IBM.

† Research Laboratory of Electronics, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139. E-mail: *mwr@rle-vlsi.mit.edu*

Inverse transformation yields the following time-domain overrelaxation expression:

$$(4) \qquad x_i^{k+1}(t) \leftarrow x_i^{k+1}(t) + \int_{-\infty}^{\infty} \alpha(t-\tau) \cdot \left[x_i^{k+1}(\tau) - x_i^k(\tau)\right] \, d\tau$$

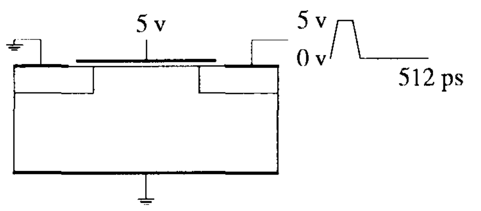
With this waveform frequency-dependent successive overrelaxation algorithm (WFD-SOR) [5], instead of multiplying the delta in the time domain by a constant parameter α as in ordinary WSOR, the delta in the frequency domain is multiplied by a frequency-dependent overrelaxation parameter $\alpha(\omega)$. The rationale for this approach is that different frequency components of x converge at different rates. In practice, the frequency-dependent overrelaxation parameter $\alpha(\omega)$ is computed as follows:

Step 1 Perform enough initial Gauss-Seidel (GS) WR iterations so that the largest eigenvalue $\gamma(\omega)$ of the GS WR operator dominates convergence at each frequency.

Step 2 Estimate the largest magnitude eigenvalue $\gamma(\omega)$ of the GS WR operator with the Rayleigh quotient: $\gamma(\omega) = \dfrac{[\Delta^k(\omega)]^* \Delta^{k-1}(\omega)}{[\Delta^k(\omega)]^* \Delta^k(\omega)}$

Step 3 Compute the overrelaxation parameter: $\alpha(\omega) = \dfrac{2}{1 + \sqrt{1-\gamma(\omega)}} - 1$

Figure 2 shows the convergence of WR, WSOR and WFDSOR for a typical simulation with 256 fixed timesteps and 64 initial WR iterations. Ordinary WSOR, with parameter chosen for frequency 0 (DC), diverges, while the frequency-dependent overrelaxation algorithm WFDSOR converges rapidly (10 orders of magnitude in 256 iterations).



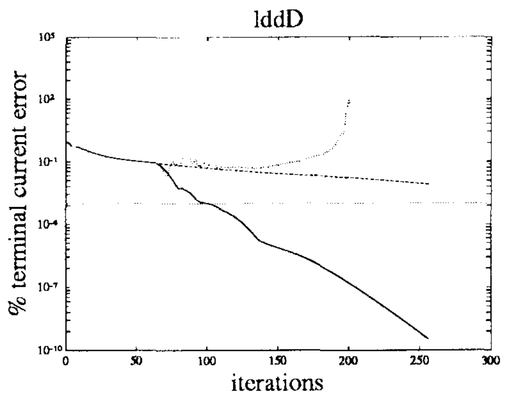


FIG. 2. *An example simulation set up and a plot of convergence (terminal current error vs. iteration) for WR (dashed), ordinary WSOR (dotted), and frequency-dependent WFDSOR (solid). The horizontal dashed line represents an accuracy of 0.1 percent.*

REFERENCES

[1] R. E. BANK, W. C. COUGHRAN, JR., W. FICHTNER, E. GROSSE, D. ROSE, AND R. SMITH, *Transient simulation of silicon devices and circuits*, IEEE Trans. CAD, 4 (1985), pp. 436–451.

[2] E. LELARASMEE, A. E. RUEHLI, AND A. L. SANGIOVANNI-VINCENTELLI, *The waveform relaxation method for time domain analysis of large scale integrated circuits*, IEEE Trans. CAD, 1 (1982), pp. 131–145.

[3] U. MIEKKALA AND O. NEVANLINNA, *Convergence of dynamic iteration methods for initial value problems*, SIAM J. Sci. Statist. Comput., 8 (1987), pp. 459–482.

[4] M. REICHELT, J. WHITE, AND J. ALLEN, *Waveform relaxation for transient two-dimensional simulation of MOS devices*, in Proc. International Conference on Computer-Aided Design, Santa Clara, CA, November 1989, pp. 412–415.

[5] ———, *Waveform frequency-dependent overrelaxation for transient two-dimensional simulation of MOS devices*, in Proc. Workshop on Numerical Modeling of Processes and Devices for Integrated Circuits, Seattle, WA, May 1992, pp. 161–166.

Design of a Multithreaded Processor Architecture

Madhumitra Sharma *
MIT Room NE43-237
e-mail: sharma@abp.lcs.mit.edu

June 12, 1992

Multithreading has long been recognized as a solution to the latency problem in multiprocessor systems. It allows processors to tolerate the long and unpredictable latencies of communication and synchronization operations. With the widening gap between processor and memory speeds, long memory access latencies are now being encountered even on uniprocessor systems. Therefore, today, multithreading appears to be an attractive solution even for uniprocessor systems.

Several multithreaded machines have been built, the most prominent among them being the HEP, HORIZON, and the MONSOON. An evaluation of these architectures reveals two principal shortcomings: First, they perform at or near their rated performance only when a large number (at least 8) of concurrent instruction streams are available. Second, the hardware cost of these architectures, in terms of silicon area and bandwidth requirements, is very high. More recent projects such as the TAM project at Berkeley [3] and the ALEWIFE [1] and *T [2] projects at MIT suggest that multithreading can be achieved on (possibly slightly enhanced) stock processors. Thread-switching, scheduling, synchronization, and context swapping can all be effectively accomplished in software. This class of machines (termed "multithreaded stock processors") are attractive for two reasons: First, they use commodity processors, which keep better pace with technology than specially designed processors. Second, multithreaded stock processors deliver adequate performance on programs with low levels of parallelism. However, they do incur significant thread-switching and context-switching overhead. Further, they inherit several limitations of conventional single-thread processors. We believe that these factors will limit the performance of multithreading in the long run.

We present a new microarchitecture that significantly reduces the overhead associated with multithreading and, at the same time, can overcome performance limitations imposed on single-threaded processors by the processor-memory interface.

First we present a scheme for caching contexts (Fig. 1) in a multi-window register file that essentially eliminates context-switching costs. The register set is configured as a conventional cache with activation frames direct-mapped onto register windows. Register names in instructions denote offsets in the frame. The operand fetch unit of the processor maintains a scoreboard to keep track of the availability of frame values in the register set and fetches unavailable ones implicitly on demand. Coupled with a split data/frame cache, this mechanism masks context switching costs very effectively – at the expense of a somewhat longer pipeline.

Next, we evaluate performance implications of the most basic choices in the design of multithreaded processors with a simple queueing model. The choices are along two dimensions: (1) the processor pipeline organization and (2) the definition of threads.

*This research was supervised by Prof. Arvind and supported in part by DARPA under Grant N000014-89-J-1988

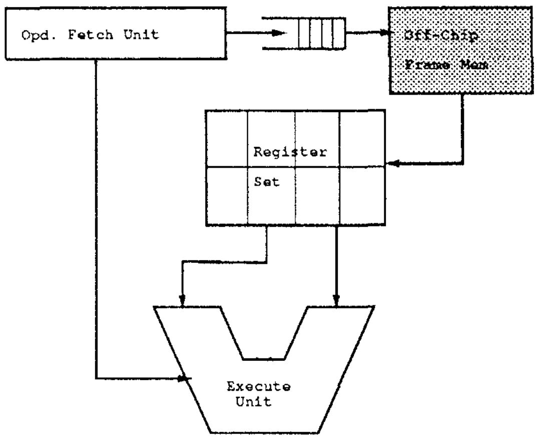


Figure 1: Context Switching Mechanism

- *Processor Pipeline:* Multithreading may be implemented on a *single-threaded pipeline* or on a *multithreaded pipeline*. A single-threaded pipeline is defined as one where the processor pipeline contains only one thread at a time. A multithreaded pipeline, on the other hand, may contain more than one threads at any time.
- *Thread Definition:* Threads, in turn, can be defined *dynamically* or *statically*. Dynamically defined threads suspend on dynamic events such as cache misses and synchronization failures. In the static case, points of suspension are defined by the compiler. For example, the compiler may chose to suspend threads on all references to data memory, on references to non-local data memory, on all branches and other transfers of control, etc.

The model indicates that the paradigm of statically defined threads on multithreaded pipelines is the one most amenable to high cache miss ratios, long pipelines and long memory access latencies.

References

[1] Anant Agarwal, Ben-Hong Lim, David Kranz, and John Kubiatowicz. April: A processor architecture for multiprocessing. In *Proc. 17th Annual Intl. Symp. on Computer Architecture, Seattle, Washington, U.S.A.*, pages 104–114, May 28-31 1990.

[2] Arvind, G. A. Boughton, R. Greiner, R. S. Nikhil, G. Papadopoulos, and K. Traub. *T: General Purpose Parallel Machines . In *CSG Memo (Unreleased)*, 1991.

[3] David E. Culler, Anurag Sah, Klaus Erik Schauser, Thorsten von Eicken, and John Wawrzynek. Fine-grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine. In *Proc. ASPLOS-91*, 1991.

[4] Anoop Gupta, John Hennesey, Wolf-Dietrich Weber, Hourosh Gharachorloo, and Todd Mowry. Comparative Evaluation of Latency Reducting and Tolerating Techniques. In *Proceedings of the 18th Annual International Symposium on Computer Architecture, 1991 at Toronto, Canada*, pages 254–263, May 27-30 1991.

M.I.T.
TRANSIT
PROJECT

Approximation of Performance Parameters for Multistage, Multipath Networks

Patrick G. Sobalvarro
pgs@ai.mit.edu

M.I.T. Artificial Intelligence Laboratory
Cambridge, Massachusetts 02139
June 14, 1992

This work [4] presents a quick means of approximating performance parameters of multistage, multipath networks for parallel computers. The networks modeled are dilated networks like the randomly-wired multibutterflies described in [2]. The bandwidth and probability of successful message transmission in such networks cannot be calculated by the methods of Patel [3] or Kruskal and Snir [1], because those methods are specific to Banyan networks: they assume that the loads on channels entering a switch are independent, whereas in multipath networks these loads are correlated.

Equations that yield the probability of loading on the output channels in terms of the input loading probabilities and switching probabilities have been found and are described in [4]. Consider a switch in a multipath network, shown in Figure 1. Because of the possible correlation of channel loads in a multipath network, in order to calculate the probability of some output configuration $\{L = l\}$ of the switch S, we calculate the joint probability mass function of the loads on the channels $C_{11}, \ldots, C_{iw}$, subsets of which are independent given their input loads.

The resulting equations are solved recursively across the stages of the network to find the loading probabilities on output channels of the final stage. The equations are of the form

$$
\begin{aligned}
&\mathrm{P}\{L_{C_{11}} = l_{C_{11}}, \ldots, L_{C_{iw}} = l_{C_{iw}}\} = \\
&\sum_{l_{B_{11}}, \ldots, l_{B_{it}}} \mathrm{P}\{L_{C_{11}} = l_{C_{11}}, \ldots, L_{C_{1u}=l_{C_{1u}}} \mid L_{B_{11}} = l_{B_{11}}, \ldots, L_{B_{1r}} = l_{B_{1r}}\} \cdot \\
&\qquad \mathrm{P}\{L_{C_{21}} = l_{C_{21}}, \ldots, L_{C_{2v}=l_{C_{2v}}} \mid L_{B_{21}} = l_{B_{21}}, \ldots, L_{B_{2s}} = l_{B_{2s}}\} \cdot \\
&\qquad \ldots \\
&\qquad \mathrm{P}\{L_{C_{i1}} = l_{C_{i1}}, \ldots, L_{C_{iw}} = l_{C_{iw}} \mid L_{B_{i1}} = l_{B_{i1}}, \ldots, L_{B_{it}} = l_{B_{it}}\} \cdot \\
&\qquad \mathrm{P}\{L_{B_{11}} = l_{B_{11}}, \ldots, L_{B_{it}} = l_{B_{it}}\}
\end{aligned} \tag{1}
$$

For networks with oblivious routing and stochastic concentration, for an $M \times N$, dilation K switch, the conditional probabilities in Equation (1) are of the form

$$
\begin{aligned}
&\mathrm{P}\{L_{1,1} = l_{1,1}, \ldots, L_{N,k} = l_{N,k} \mid L_{C_1} = l_{C_1}, \ldots, L_{C_M} = l_{C_M}\} = \\
&\qquad \left(\prod_{i}^{N} \frac{1}{\binom{K}{b_i}}\right) \sum_{d_1, \ldots, d_N} \binom{\sum_{i=1}^{M} l_{C_i}}{d_1, \ldots, d_N} q_1^{d_1} q_2^{d_2} \cdots q_N^{d_N}
\end{aligned} \tag{2}
$$

Here $q_1, q_2, \ldots, q_N$ are the switch's probabilities of switching in each direction, calculated from the addressing probabilities. Also $b_i = \sum_{g=1}^{K} l_{i,g}$, and the sum is over the N-tuples $d_1, \ldots, d_N$ such that for each d_i, $\min(d_i, K) = b_i$, and $\sum_{i=1}^{N} d_i = \sum_{i=1}^{M} l_{C_i}$.

Acknowledgments: The research described in this abstract was conducted under the direction of Professor Thomas F. Knight, Jr. at the M.I.T. Artificial Intelligence Laboratory. The research was supported in part by the Defense Advanced Research Projects Agency under contract N00014-87-K-0825. The author was supported in part by the Charles Stark Draper Laboratory under a Draper Fellowship.

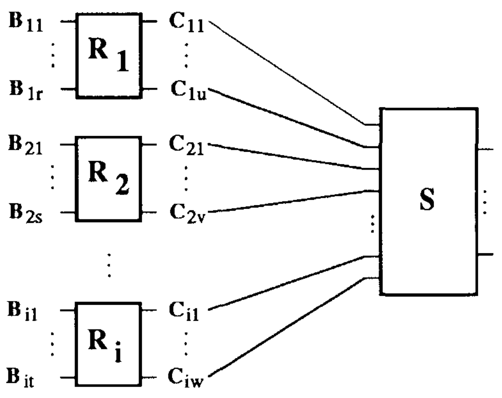


Figure 1: Channels referred to in Equation (1). The recursive step arises because in order to find the probability of an output loading configuration of the switch S, we condition on the loads on its input channels $C_{11},\ldots,C_{iw}$. No subset of $C_{11},\ldots,C_{iw}$ need have mutually independent loads in a multipath network, but the loads on the subset of channels from each switching element are independent given the message loads on the input channels $B_{11},\ldots,B_{it}$.

Because of the summation in Equation (1), there are in the general case an exponential number of these equations (although in the specific case of a Banyan network, these equations reduce to Patel's equations). A program has been written to evaluate them and make simplifications based on independence that can be determined from the network graph. The program can be used for small networks and on special cases with limited path redundancy, but an approximation method is necessary for larger examples.

A Monte Carlo approximation method can be used to estimate the desired probabilities, rather than solve for them exactly. It can be shown that, if $B_1,\ldots,B_m$ are the input channels and $O_1,\ldots,O_m$ the output channels of some number k of final stages of the network, then if we generate tuples $l_{B_1},\ldots,l_{B_m}$ randomly in accordance with the probability mass function $\mathrm{P}\{L_{B_1}=l_{B_1},\ldots,L_{B_m}=l_{B_m}\}$,

$$h(l_{B_1},\ldots,l_{B_m})=\mathrm{P}\{L_{O_1}=l_{O_1},\ldots,L_{O_n}=l_{O_n}\mid L_{B_1}=l_{B_1},\ldots,L_{B_m}=l_{B_m}\} \tag{3}$$

is an unbiased estimator of $\mathrm{P}\{L_{O_1}=l_{O_1},\ldots,L_{O_n}=l_{O_n}\}$. The correlated random variates $l_{B_1},\ldots,l_{B_m}$ are easily generated by a method described in [4], and Equation 3 can be evaluated exactly if k is chosen carefully. Where $k=1$, the scheme always yields an estimate in polynomial time.

The resulting approximation scheme will always have lower variance than the obvious hit-or-miss direct simulation technique, and so achieves a given error bound in fewer iterations. A program that uses this estimation technique to calculate performance parameters for multipath networks has been written, and on typical examples achieves given error bounds in about 1/9 the number of iterations required under direct simulation.

References

[1] Kruskal, C. P., and Snir, Marc. "The Performance of Multistage Interconnection Networks for Multiprocessors," in *IEEE Transactions on Computers,* Vol. C-32, No. 12, December 1983.

[2] Leighton, T., and Maggs, B. "Expanders Might be Practical: Fast Algorithms for Routing Around Faults in Multibutterflies," in *30th Annual Symposium on Foundations of Computer Science,* IEEE Computer Society Press, November, 1989.

[3] Patel, J. H. "Performance of Processor-Memory Interconnections for Multiprocessors," in *IEEE Transactions on Computers,* Vol. C-30, No. 10, October 1981.

[4] Sobalvarro, P. G. *Probabilistic Analysis of Multistage Interconnection Network Performance.* S.M. Thesis, Massachusetts Institute of Technology Department of Electrical Engineering and Computer Science. April, 1992.

Dribbling Registers: A Technique for Latency Tolerance in Large-Scale Multiprocessors

Vijayaraghavan Soundararajan and Anant Agarwal*
NE43-633
ravi@vindaloo.lcs.mit.edu

1 Introduction and Description

As parallel machines grow in scale and complexity, latency tolerance of synchronization faults and remote memory accesses becomes increasingly important. One method of tolerating this latency is multithreading the processor and rapidly context switching between multiple threads. Analyses show[2] that fast context switching between a few (3-4) processor-resident threads is adequate when the latencies being tolerated are short compared to the total run lengths of all the resident threads. If this condition is not met (which is often the case for synchronization latencies), many more threads are needed. Because hardware costs limit the number of processor-resident threads, fast switching between a large number of threads is difficult to achieve.

This paper proposes a mechanism for fast switching between a large number of threads. The basic idea is to implement a few (say, 3) threads in hardware, but attempt to provide a larger supply of threads to switch among by continually unloading stalled threads from the register file and loading runnable threads into it. Dribbling registers (D-registers) facilitate fast context switching and the ability to hide the latency of loading and unloading context state[1].

D-registers, inspired by Sites' dribble-back registers for multiple register window architectures[5], optimize utilization of the processor under loads in which run lengths are short and wait times are long (relative to the run lengths). In this regime, register file designs often provide inefficient performance since loading is frequent and many cycles are wasted in loading rather than used for execution. D-registers alleviate time wasted on loading/unloading by amortizing these cycles with useful processor execution cycles.

On-chip instruction caches decrease traffic to off-chip memory, providing long periods in which the cache/data pathway goes unused (in RISC processors the number of non-load/store instructions and hence the number of free memory bus cycles can approach 70%[4]). D-registers use these free cycles to load context information concurrently with program execution. This is accomplished by adding 1 read/write port to the (typically three) ports already existent in the multiple register set register file. Data accesses

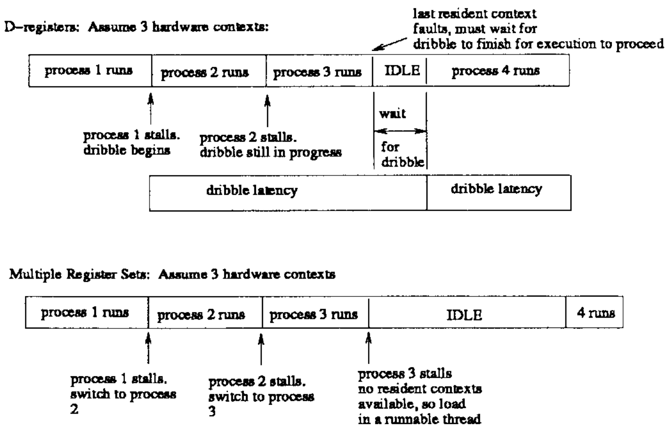

D-registers: Assume 3 hardware contexts:
last resident context faults, must wait for dribble to finish for execution to proceed
process 1 runs
process 2 runs
process 3 runs
IDLE
process 4 runs
process 1 stalls. dribble begins
process 2 stalls. dribble still in progress
wait for dribble
dribble latency
dribble latency
Multiple Register Sets: Assume 3 hardware contexts
process 1 runs
process 2 runs
process 3 runs
IDLE
4 runs
process 1 stalls. switch to process 2
process 2 stalls. switch to process 3
process 3 stalls no resident contexts available, so load in a runnable thread

Figure 1: Load Latencies for Dribbling and Multiple Register Set Designs

to the register file occur through the three normal ports, and the dribble port unloads stalled context information and loads ready context information concurrently with the normal data accesses.

By continually polling the register file contents for stalled contexts after a dribble finishes, D-registers reduce the effective latency of a load as figure 1 illustrates, and attempt to keep the register file full of runnable threads. The dribbler only unloads threads stalled due to synchronization faults, since cache miss latencies are short enough to be tolerated by context switching.

Analytical models and simulation show that processor utilization approaches unity for typical workload parameters.

2 Analytical Model

We can model the processor utilization in a system that employs D-registers by using a simple queueing model. The register file is modeled as a queueing server in which register frames are added at some rate by the dribbler, and register frames are consumed at some other rate by an execution process that causes synchronization faults. A register frame is added into the queueing server the moment the dribble of a frame completes. Similarly, a register frame is considered

*This work was supervised by Professor Anant Agarwal. The abstracted research was funded by DARPA contract #N00014-87-K-0825.

[1] A more thorough discussion is presented in [6]

to be consumed the moment a synchronization fault occurs. The utilization of the processor is the fraction of the time the register file is busy, and is simply the utilization of the queueing server.

Let the service rate or consumption rate be U: 1/U is the average time between synchronization faults, i.e., the average run length. Let the arrival rate be L, i.e., 1/L is the time between completions of dribble operations. In other words, L is the probability that a dribble completes on a given cycle, and a process is added. The utilization, ρ, of the queueing server is given by

$$\rho = \frac{L}{U}. \tag{1}$$

ρ has a maximum of 1. That is, if the rate of synchronization faults is less than the rate of dribbling, then in the steady state, there will be no idle time. This formula only applies when $L \leq U$. For $L > U$ ρ should approach 100% since dribbles furnish runnable threads more frequently than synchronization faults consume them. This formula assumes infinite register file size; in practice, a register file size of 3 or 4 resident contexts is sufficient to approximate this condition.

Validation against experimental data shows that the model agrees to within 10%. [6] presents a more extensive model including load/store instruction considerations, and shows a comparison of simulation and model.

3 Experimental Framework

In order to evaluate the effectiveness of D-registers and to validate the model, three functional trace-driven simulators for each of three different register file models were written. These models include

1. Single Register Set: only one context is stored in the register file at any given time. Cache misses cause idling until satisfaction of the request, and on each synchronization fault the resident context is unloaded and replaced with a runnable thread.
2. Multiple Register Set[1]: multiple hardware contexts enable support of many threads concurrently. Context switching is used to hide cache miss latencies and synchronization fault latencies. If all resident contexts are already stalled by (large-latency) synchronization faults, one of the stalled contexts is unloaded and a runnable thread is loaded.
3. Context Cache[3]: The Context Cache treats the register file as a fully-associative cache, so a context load/unload involves loading only the registers required for the current instruction, rather than loading each register of the context.

The workload parameters used to synthesize traces to drive the simulators include (1) the number of runnable threads, (2) the number of hardware contexts (or resident registers in Context Cache), (3) the number of registers in a context, (4) the frequency of synchronization faults and load/store operations, (5) the cache hit rate on load/store operations, (6) the synchronization fault latency and cache miss latency, and (7) the lengths of the individual threads. Although real program traces are not run through the simulators, the parameters specified above are taken from real parallel applications[4].

4 Results

D-registers provide significantly higher utilization than the multiple register set design, nearly doubling it in some cases. Utilization increases under typical workload parameters and a cache miss rate of 9% from 50% with multiple register sets to 90% with the addition of dribbling registers. D-registers provide slightly lower processor utilization (40% vs. 50%) than the Context Cache for very small runs between synchronization faults (about 30 cycles), because not all registers in the context are used, but with D-registers all are still loaded (thus resulting in wasted register loads): since this region is the intended operating region for the Context Cache, such performance is expected. As run lengths increase (approaching the average time between dribble completions, or about 60 cycles), D-registers outperform the Context Cache (90% vs. 70%). In addition, experimental results show that processor utilization remains virtually constant with increasing numbers of contexts, thus validating the model's assumption of infinite register file size.

References

[1] Anant Agarwal, Beng-Hong Lim, David Kranz, and John Kubiatowicz. APRIL: A Processor Architecture for Multiprocessing. In *Proceedings 17th Annual International Symposium on Computer Architecture*, June 1990.

[2] Wolf-Dietrich Weber and Anoop Gupta. Exploring the Benefits of Multiple Hardware Contexts in a Multiprocessor Architecture: Preliminary Results. In *Proceedings 16th Annual International Symposium on Computer Architecture*, June 1989.

[3] Peter R. Nuth. Parallel Processor Architecture: A Thesis Proposal. MIT VLSI Memo, 1990.

[4] Jaswinder Pal Singh, Wolf-Dietrich Weber and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. To appear in SIGARCH, Computer Architecture News, Spring 1992

[5] Richard L. Sites. How to Use 1000 Registers. In *Proceedings, 1st Caltech Conf. VLSI*, California Institute of Technology. January 1979

[6] Vijayaraghavan Soundararajan. Dribble-Back Registers: A Technique for Latency Tolerance in Multiprocessors. Bachelor's thesis, Dept. Elec. Eng. & Computer Sci., MIT, 1992. Supervised by Anant Agarwal.

Compiling TAM Code to the J-Machine

Ellen Spertus*
NE43-630
ellens@ai.mit.edu

Introduction

The Threaded Abstract Machine (TAM) model was designed to allow efficient execution of fine-grained programs on machines with minimal hardware support [1]. Fine-grained execution is desirable because it hides the latency that occurs in large parallel computers. I developed two different ways of compiling TAM code to run on the J-Machine. In this abstract, I use the straightforward translation scheme as the basis for a comparison of the J-Machine to the CM-5, which also has a TAM implementation, and I compare the two translation schemes for the J-Machine.

TAM

When a program is compiled for TAM, each procedure is divided into a set of *threads* and *inlets*. Threads encode the body of the procedure, while inlets are message handlers. A thread can be either *synchronizing* or *non-synchronizing*. If it is synchronizing, it has an *entry count* which must reach zero before the thread can run. Each thread is comprised of instructions such that once a thread begins, it can always complete without waiting for data [2]. When a procedure is invoked, a *frame* is allocated for the storage of arguments, local variables, entry counts, and a *remote continuation vector* (RCV). The RCV lists which threads are ready to run. A software queue is maintained of frames whose RCVs are non-empty.

When a frame is removed from the queue, its RCV becomes the local continuation vector (LCV), and a specially-designated *entry thread* is run, which loads frequently-used frame slots into registers. Threads are executed from the LCV until it is empty, at which time the *exit thread* runs, storing register values into the appropriate frame slots. When a thread computes a data value or control information on which another thread depends, it *forks* the thread. When a non-synchronizing thread is forked, a pointer to the thread is placed in the LCV. When a synchronizing thread is forked, it is only placed in the LCV if decrementing its entry count yields zero. A similar operation, *post*, is used within inlets, which places the target thread in the RCV.

The only language that is currently compiled to TAM is Id. TAM supports I-structures and M-structures.

Comparison to the CM-5

To judge how well the J-Machine supports fine-grained parallelism, we compared a straightforward implementation of TAM on

*This research was supervised by Prof. William J. Dally and was supported in part by the Defense Advanced Research Projects Agency under contracts N00014-88K-0738 and N00014-87K-0825, by a National Science Foundation Presidential Young Investigator Award, grant MIP-8657531, with matching funds from General Electric Corporation and IBM Corporation, and by a National Science Foundation Graduate Fellowship.

TAM Mechanism	Straightforward Implementation	Flattened Implementation
inlet	priority 0 message handler	priority 0 message handler
post from inlet	placement of thread in RCV	jump directly to thread
activation of frame	message from background to entry thread	n/a
entry thread	background priority code, which jumps to threads within procedure	not used
threads	background priority code	priority 0 code
exit thread	sequence of code run at background priority	not used
fork from thread	jump or push onto LCV	jump or push onto CV
system routines	priority 0 message handlers	priority 1 message handlers

Table 1: Mapping of TAM Constructs to the J-Machine

the J-Machine, to the CM-5 implementation. It highlighted the following architectural differences:

Network Interface On the J-Machine, messages can be of almost unlimited length and are sent directly from the processor. This proved superior to the CM-5, on which messages are limited to 5 words and are sent by system calls operating on memory-mapped queues. Additionally, the CM-5 needed to poll to check for incoming messages, while the J-Machine's Message-Driven Processor (MDP) has hardware dispatch.

Tags To our surprise, the MDP's *cfuture* tag provided little benefit in implementing I-structures and M-structures.

Conventional Architectural Features Due to the J-Machine's longer cycle time, shortage of registers, and lack of a cache, its performance was inferior to the CM-5's.

A Flattened Translation Strategy

While the J-Machine supports TAM well in a straightforward way, the MDP's strengths and weaknesses make a different model more efficient. Table 1 compares the two ways of translating TAM code. Because the MDP does not have a cache and is short on registers, it does not benefit from TAM's two-level hierarchy in which threads from the same procedure are grouped together. A model that flattens this hierarchy was implemented, where, instead of storing a posted thread in a RCV, the thread is executed immediately after the inlet that posted it. By relying more heavily on the MDP's hardware message queues and dispatch mechanism and exposing the code to further optimizations, greater efficiency is achieved, although the risk of queue overflow is increased. On a sample program (paraffins with an argument of 10), the flattened implementation required 77% the instructions of the straightforward implementation.

References

[1] Culler, D. E., A. Sah, K. E. Schauser, T. von Eicken, and J. Wawrzynek. "Fine-grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine," *ASPLOS-IV*, 1991, pages 164 – 175.

[2] Schauser, K. E. Compiling Dataflow into Threads: Efficient Compiler-Controlled Multithreading for Lenient Parallel Languages. Master's Project, Department of EECS, University of California, Berkeley, 1991.

Hindsight: Debugging Parallel Programs using Reordering

by Bradford T. Spiers*
Large-Scale Parallel Software Group, MIT LCS
e-mail: btspiers@lcs.mit.edu

1 Introduction

In this paper we present Hindsight, a debugger for parallel programs. Hindsight is designed to handle nondeterministic errors common in MIMD programs. Hindsight assumes that a program execution is composed of threads, which are in turn composed of a set of atomic code blocks. Threads communicate by exchanging messages. Hindsight helps programmers fix errors caused by atomic code blocks that execute in a different order than the programmer intended. It does not address races in parallel programs in which the execution of atomic code blocks overlap erroneously. This approach addresses the harder of the two problems, as Miller and Netzer [NM88] point out: detecting races is relatively easy (see Emrath and Padua [EP89] for an example), but correcting out-of-order atomic code blocks is NP-hard.

2 Nondeterministic Errors

A nondeterministic error is caused by timing differences. These differences change the order in which atomic code block execute. Such changes can produce unpredictable results, but the underlying problem is an incorrect order of execution of atomic code blocks. The atomic code blocks can execute out of order because they are underconstrained. To fix the program, the programmer must add constraints.

3 Related Work: Debuggers that Provide Just Replay

To handle nondeterministic programs, many debuggers for parallel systems record a log file. The log file contains a history of the order in which an execution's synchronizing events take place. The debugger can use the log file to rerun an execution in the same order as occurred in the logged execution. This reexecution is normally called replay [Mel89]. During replay, a programmer can examine program state on any node using sequential debugger features. Using this approach, a programmer is suppossed to be able to debug nondeterministic programs.

Unfortunately, a programmer who uses replay to debug parallel programs cannot verify if a fix is correct. To see why that is true, consider a debugging scenario for a nondeterministic bug. The programmer uses replay to examine the program and decides what the bug is and how to fix it. Next, he changes the source code to fix the bug and recompiles the program. Finally, the programmer runs the new program, which obtains the correct result. Now we are unsure of what has happened: did the change fix the error, or did it merely change relative timings enough to hide the error for now? The programmer has no way of knowing which possibility actually occurred.

4 Hindsight

Hindsight allows the programmer to reorder concurrent atomic code blocks. It starts from the premise that the programmer suspects that the program lacks constraints. The programmer then adds the constraints necessary to obtain the desired ordering by specifying the order in which a set of concurrent, atomic code blocks will execute.[1] The user may not add or subtract atomic code block executions because the source code is not modified. Hindsight checks whether the reordering violates control dependencies. If none are violated, then the modified execution is replayed. A design with similar goals was presented by Goldberg et al. [GGLS91].

Figure 1 shows a sample reordering using Hindsight. In one thread of the original execution, the atomic code block executions {A, B, C} are underconstrained. The programmer suspects that this ordering could be causing the problem, so he chooses to reorder atomic code blocks in that thread. He specifies a new ordering for that subset of atomic code blocks, {C, A, B}. Hindsight then replays the portion of the computation that happened before [Lam78] the reordering so that the system is in the same state just before the start of the reordering as in the original

*Supervised by both Professor William E. Weihl and Al Davis. E-mail: weihl@lcs.mit.edu, adavis@hplald.hpl.hp.com. Supported by an National Defense Science and Engineering Graduate Fellowship. Supported in part by the NSF under grant CCR-8716884, DARPA under contract N00014-89-J-1988, and by an equipment grant from DEC.

[1]Multiple sets may be reordered at the same time. We explain the one-set case here for simplicity.

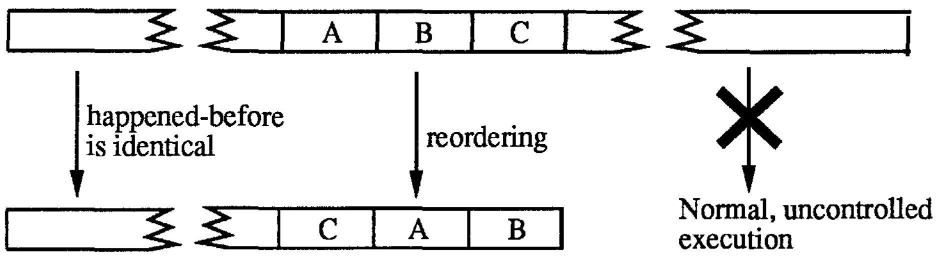


Figure 1: A sample reordering using Hindsight. The user reorders atomic code block executions, {A,B,C}, to {C,A,B}. Hindsight replays the modified ordering through the end of the reordering.

execution. Then Hindsight controls execution so that the new ordering, {C, A, B}, occurs. Hindsight replays the parts of other threads that execute concurrently with the reordering so that different results are not obtained due to different orderings of concurrent threads, whereas we want to isolate the effects of the reordering.[2] Then, Hindsight releases control of atomic code block executions because the state has hopefully changed due to the reordering; thus, the original log is no longer valid. When the program finishes, the programmer has three choices. If the ordering did not produce the correct results, then the programmer may use the original logs to try another reordering. If the ordering produced the correct results, then the programmer can either change the source code to produce that new ordering, called the target ordering, or use the logs recorded during replay of the reordered execution to fix another nondeterministic error.

The specific contributions of this work include:

- Allowing programmers to fix nondeterministic errors with certainty that the error has not been hidden by changed timing.
- Providing a platform on which to build a testing tool that can exercise different orderings. Such a tool was called for by Taylor et al.[TLK92].

5 Conclusion

We are currently implementing Hindsight. A one-node version works on simple test cases; we expect to finish testing a multi-node debugger by the end of this summer.

Hindsight simplifies testing for and debugging of nondeterministic errors due to unanticipated orderings of atomic code blocks. Hindsight's ability to reorder thread executions differentiates it from traditional debuggers which provide only replay. Replay is inadequate because the programmer cannot determine if code changes fix a bug or if relative timing differences hide it. Thus, Hindsight provides a tool for testing and debugging nondeterministic programs that is absent from current systems.

[EP89] Perry A. Emrath and David A. Padua. Automatic Detection of Nondeterminacy in Parallel Programs. *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, ACM SIGPLAN Notices*, 24(1):89–99, January 1989.

[GGLS91] Arthur P. Goldberg, Ajei Gopal, Andy Lowry, and Rob Strom. Restoring Consistent Global States of Distributed Computations. In *Proceedings of ACM/ONR Workshop on Parallel and Distributed Debugging 1991*, pages 140–149, 1991.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[Mel89] John M. Mellor-Crummey. *Debugging and Analysis of Large-Scale Parallel Programs.* PhD thesis, University of Rochester, 1989.

[NM88] Robert Netzer and Barton Miller. What are Race Conditions? Some Issues and Formalizations. In *Proceedings of ACM/ONR Workshop on Parallel and Distributed Debugging 1991*, pages 251–253, 1988.

[TLK92] Richard N. Taylor, David L. Levine, and Cheryl D. Kelly. Structural Testing of Concurrent Programs. *IEEE Transactions on Software Engineering*, 18(3):206–215, March 1992.

[2]However different orderings of concurrent threads is useful for testing.

Exploiting Algorithmic Locality in Water*

Bradford T. Spiers†
Large-Scale Parallel Software, MIT LCS
btspiers@lcs.mit.edu

Donald Yeung‡
Alewife Research Group, MIT LCS
piano@lcs.mit.edu

June 12, 1992

1 The Water Application

Water is an N-body moleculer dynamics simulation which appears in the SPLASH benchmark suite [1]. Assuming quasi-statics, atomic-level interactions between water molecules situated in three-dimensional space are computed using Gear's sixth-order predictor-corrector method [2]. Under the influence of these interactions, the Water application simulates the trajectories of the molecules using Newtonian equations of motion for a user-specified number of time-steps.

In the Water application, the force that a molecule exerts on other molecules has a limited radius of influence, so only molecules that are situated within some cutoff radius of one another are allowed to interact. An algorithm that is oblivious to the relative locations of molecules must consider all possible pairs of molecules even if most of them are too far apart to influence each other. However, an algorithm that is cognizant of such spatial information can consider only those pairs of molecules that are close enough to interact and can therefore significantly reduce the amount of extraneous computation.

The original ipmlementation of Water appearing in the SPLASH benchmark does not exploit the algorithmic locality inherent in the Water application. Our study provides an implementation of Water that takes advantage of algorithmic locality. We evaluate the performance gains of our implementation over the original implementation and identify some important issues.

2 Analysis

The following expression represents the amount of work that each processor must perform in the original implementation of the Water application. In all expressions, n denotes the number of molecules and p denotes the number of processors.

$$\left(\frac{n}{p}\right)\left[\frac{n-1}{2}\right] \Rightarrow O\left(\frac{n^2}{p}\right) \quad (1)$$

In the original implementation, each processor is responsible for n/p molecules, and for each molecule, interactions with every other molecule in the simulation must be considered. There is a factor of 1/2 because when the interaction for a pair of molecules is computed, a solution for *both* molecules is obtained. As the order of complexity indicates, the total work done in the entire simulation grows as the square of the problem size.

In comparison, the equivalent expression for our modified implementation is as follows.

$$\left(\frac{n}{p}\right)\left[\left(\frac{1}{2}\right)\left(\frac{n}{p}-1\right)+13\left(\frac{n}{p}\right)\right] \Rightarrow O\left(\frac{n^2}{p^2}\right) \quad (2)$$

Again, each processor is responsible for n/p molecules. The difference in our implementation is that the simulation space is divided into p regions, one for each processor, and each processor is responsible for all the molecules which are situated in its region. Processors no longer have to consider interactions between all possible pairs of molecules; instead, only the interactions between molecules residing in regions which are close together in space need to be computed. These interactions can be divided into two components. The first represents the interactions between all possible pairs of molecules residing in the same region. This term is similar to the expression for the original implementation and contains a factor of 1/2 for the same reason given above. The second component represents the interactions with all the molecules in near-neighbor regions. In our implementation, the number of near-neighbor regions for a given region is thirteen. The order of complexity indicates that the total amount of work done in the entire simulation grows as the square of the problem size divided by p. Therefore, we expect our implementation to be faster by a factor of p over the original implementation.

*This work was supervised by Anant Agarwal

†Supported by an National Defense Science and Engineering Graduate Fellowship. Supported in part by the NSF under grant CCR-8716884, DARPA under contract N00014-89-J-1988, and by an equipment grant from DEC.

‡Supported in part by NSF grant # MIP-9012773, in part by DARPA contract # N00014-87-K-0825, and in part by an NSF Presidential Young Investigator Award

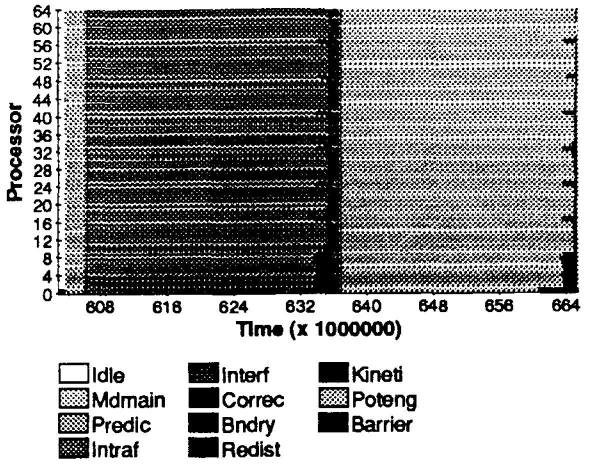


Figure 1: The original Water application with good load-balance.

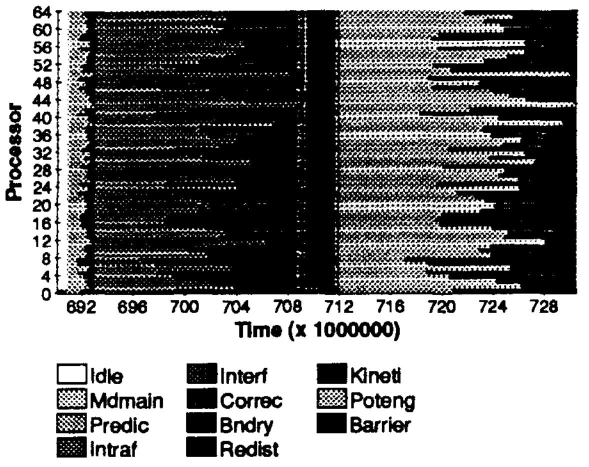


Figure 2: Our modified version of the Water application with poor load-balance.

3 Experimental Results

We conducted our experiments on Proteus, a simulator written by Brewer and Dellarocas [4]. We configured Proteus to simulate an 8x8 mesh with end-around connections, no caches, and an analytically modeled network. Proteus assumes that each node is a RISC processor which executes every instruction in one cycle.

Although our analysis predicts a speedup of p, this outcome is not reflected in our simulation results. As Figure 1 and Figure 2 illustrate, our modified version of Water achieves a speedup of only 1.5. The theoretical speedup is not obtained because of load-imbalance: the original application, shown in Figure 1, displays almost perfect load-balance, while our modified version, shown in Figure 2, shows large load-imbalance. This difference is clear when looking at the largest contributors to an iteration time, the procedures *poteng* and *interf*. Our analytical analysis assumes that we can divide the work evenly, but that is possible only if the molecules are distributed in a perfectly uniform manner.

We expect that the problem of load-imbalance will be much less severe when the problem size is increased sufficiently. The load-imbalance that we observe in our simulations is due to statistical fluctuations in the number of molecules assigned to each processor. Because water has a constant density, this statistical effect will become less significant when the problem size increases.

4 Conclusion and Future Work

In our study, we have demonstrated that an implementation of the Water application that exploits algorithmic locality affords good performance gain over one that does not. Theoretical analysis predicts a speedup of p over the original implementation. Actual simulations show performance gains as well though they are far less substantial due to load-imbalance. Load-imbalance will always be a problem for executions with small to medium problem sizes; however, the effect of load-imbalance will be less detrimental to performance when the problem size is sufficiently large. One area for future work will be to identify the point at which load-imbalance becomes negligible.

Although the primary benefit from exploiting algorithmic locality comes from minimizing extraneous computation, there is also another benefit, namely locality of data reference. Assuming an architecture that can exploit data locality (such as NUMA architectures), the assignment of molecules to processors based on their spatial locations can greatly reduce the amount of interprocessor communication. This, however, comes at the expense of good load-balance, as is discussed above. In fact, for the Water application, locality and load-balance are conflicting goals. One can be improved at the expense of the other, but both cannot be achieved simultaneously. In future work, we plan to investigate this tradeoff and try to understand how each affects the performance of the Water application.

[1] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. *SPLASH: Stanford Parallel Applications for Shared-Memory*, SIGARCH Computer Architecture News, Spring 1992.

[2] C. W. Gear. *Numerical Initial Value Problems in Ordinary Differential Equations*, Prentice-Hall, New Jersey, 1971.

[3] Eric A. Brewer and Chrysanthos N. Dellarocas. *User Documentation*, Version 0.2, October 31, 1991.

[4] Eric A. Brewer, Chrysanthos N. Dellarocas, Adrian Colbrook, and William E. Weihl. *Proteus: A High-Performance Parallel-Architecture Simulator* MIT/LCS/TR-516, September 1991.

Evaluating Game Trees in Parallel

Clifford Stein[1]
Laboratory for Computer Science
Massachusetts Institute of Technology
`cliff@theory.lcs.mit.edu`

The evaluation of game trees is a central problem in Artificial Intelligence. A typical game tree (also called a MIN/MAX tree) is rather large, so fast algorithms are essential if we are to make progress in computer aided game playing. Any algorithm that can hope to play in "real-time" must use some pruning techniques to avoid searching the whole tree. Many researchers have proposed pruning strategies for this problem and a number of good sequential search algorithms have been developed, most notably the α–β algorithm. In spite of the fact that parallel computers offer a promising approach for speeding up game tree search, less is understood about the complexity of parallel algorithms for this problem. We will show that part of the difficulty in parallelizing α–β search arises from an inherent lack of *locality*. We will then give an algorithm that employs *scaling* to partially overcome this difficulty. We will show that the number of bits needed to represent the range of leaf values is a measure of how hard the problem is to parallelize.

We will focus on the case when wish to exploit small amounts of parallelism. By small we mean that if the tree has height h, we will use $O(h)$ processors. As a starting point, we consider the problem of evaluating AND/OR trees. We will describe an algorithm by Karp and Zhang [1] for evaluating AND/OR trees that, on a tree of height h, gets $\Omega(h)$ speedup using h processors. The key idea for this algorithm is that if an AND-node has its first input evaluate to 0, all the other inputs to that AND-node can be ignored. Further, this is the only type of pruning that can occur. Thus all pruning is based on local information, namely the value of a node's siblings.

Karp and Zhang originally believed that this algorithm could be extended to work for general game-tree search. We will show that there are substantial obstacles to overcome in extending this approach to MIN/MAXtrees.

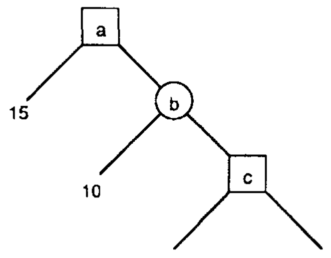


Figure 1: A (shallow) cutoff in α–β search. Squares are MAX-nodes and circles are MIN-nodes.

In α–β search, the pruning is not necessarily local. We illustrate the idea with the example in Figure 1. MAX-node a has a child of value 15, hence it's value must be at least 15. The only way that the value of MIN-node b can affect the value of a is if it is more than 15. But b has a child that is 10 and hence can have a value of no more than 10. Thus the value of the subtree rooted at c has no effect on the value of a and does not need to be evaluated. In fact,

[1]Support provided by NSF PYI Award CCR-89-96272 with matching support from UPS and Sun, by an AT&T Bell Laboratories Graduate Fellowship, and by DARPA Contract N00014-89-J-1988.

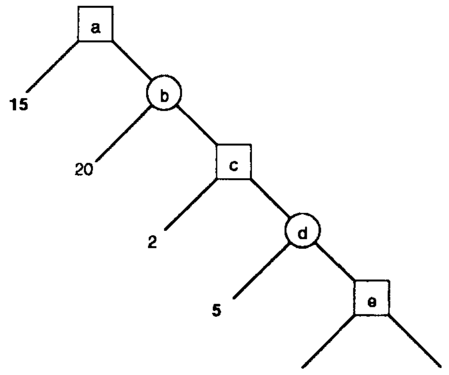


Figure 2: A (deep) cutoff in α–β search. Squares are MAX-nodes and circles are MIN-nodes.

the node being cut off does not need to be so close to the nodes that cause it to be cutoff, as the example in Figure 2 shows. Here, the value of the subtree at e does not affect the value of a. This creates difficulties for parallel algorithms because work done by one processor may turn out to be unnecessary due to some pruning that occurs far away in the tree and much later in time.

We will give a new algorithm for the problem of evaluating a MIN/MAX tree in parallel based on the idea of scaling. We will reduce the problem of evaluating a MIN/MAX tree to the problem of evaluating a series of AND/OR trees.

Our algorithm provably achieves $O(h/\gamma)$ speedup using h processors , where γ is the number of bits needed to represent the range of possible leaf values. While this is not quite optimal speedup, we have implemented a simulation of this algorithm and run it on a number of randomly generated trees. The interesting thing to note is that when the number of bits is small, we get close to optimal speedup, and that when the number of bits is large, the speedup seems to be somewhat better than (h/γ), particularly for larger h. In other words, the incremental cost of adding the 25th bit is less than that of adding the 4th bit. This is because the predicted sequential running time of a subproblem is much less that the predicted sequential running time of the original problem. When we execute many iterations, this serves to increase the effective speedup.

This approach gives one solution to the problem that the information needed to prune may arrive after a node has been evaluated. Since we do not now how to disseminate enough exact information about node values, in our algorithm we disseminate approximate information about the node values. In other words, when a node is evaluated in an iteration, it does have some information about all the left siblings of its ancestors. By the final iteration it has all the information that sequential α–β would have. So by focusing on obtaining inexact information about many nodes as opposed to exact information about fewer nodes, we are able to get better provable bounds on the algorithm's performance.

References

[1] R. Karp and Y. Zhang. On parallel evaluation of game trees. In *Proceedings of the 1989 ACM Symposium on Parallel Algorithms and Architectures*, pages 409–420, 1989.

A Parallelizing Compiler Based on Partial Evaluation

Rajeev Surati
raj@martigny.ai.mit.edu
NE43-439

Andrew Berlin
aab@martigny.ai.mit.edu
NE43-434 *

MIT AI Laboratory
545 Technology Square
Cambridge MA, 02139

We have constructed a compiler that uses partial evaluation to achieve outstandingly efficient parallel object code from very high-level data independent source programs. On several important scientific applications, our compiler attains parallel execution and overall performance equivalent to or better than the best observed results from the manual restructuring of code. Although partial evaluation has been used successfully to compile efficient sequential code for uniprocessor machines, this effort represents one of the *first* attempts to capitalize on partial evaluation's ability to expose low-level parallelism. New static scheduling techniques are used to utilize the fine-grained parallelism on a multiprocessor machine. The compiler accepts ordinary Scheme programs as source, and generates code for the Supercomputer Toolkit, a parallel computer with 8 VLIW processing nodes, by mapping the computation graph resulting from partial evaluation onto the Toolkit's architecture.

The compiler has been evaluated on two different highly abstracted programs written in Scheme which simulate n-body problems which are important in the fields of celestial mechanics and particle physics. The results reveal that it is possible to automatically achieve a factor of 6.2 speedup on an eight-processor configuration of the Supercomputer Toolkit over a highly optimized uni-processor version of the program. (The uni-processor version is executing a floating point operation in over 99% of the cycles.) The compiler's speedup is impressive because the target architecture (the Supercomputer Toolkit) has extremely low bandwidth, essentially allowing each processor to send a value once every 8 cycles on average, with a latency of 6 cycles. Our results also reveal that although the static scheduling techniques work well for computers the size of the Supercomputer Toolkit, they do not scale well to larger machines.

By reconstructing the data dependencies of the computation expressed by a program, partial evaluation succeeds in "exposing the low level parallelism in a computation by eliminating inherently sequential data-structure references." [1] Furthermore, elimination of data-independent branches produces huge basic blocks of easily parallelizable straight-line code. Huge basic blocks make it feasible to use fine-grained parallelism to spread the execution of a basic block across multiple processors, rather than assigning each basic block to an individual processor.

Currently work is being pursued in three areas. One area is the modification of the static scheduling technique so that it will scale well to computers with more processors. The second area is extending the compiler to handle data-dependent branches. The last area is the pursuit of optimizations possible only because the fine grain computation graph is available at compile time. An example of this would be

*This work is supervised by Prof. Hal Abelson and Prof. Gerry Sussman and supported in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-89-J-3202 and by the National Science Foundation under grant number MIP-9001651.

to deduce that it is faster to compute a value on the processors which require the value rather than to compute the value on one processor and then send the value to the rest of the processors.

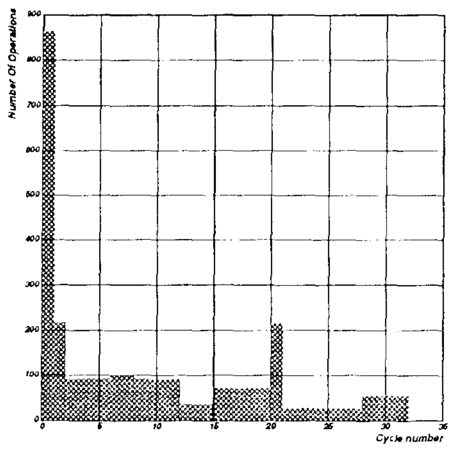


Figure 1: Parallelism profile of Stormer integration.

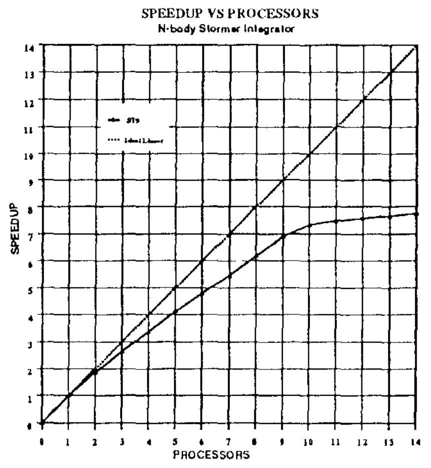


Figure 2: Speedup graph of Stormer integration.

References

[1] A. Berlin, "Partial Evaluation Applied to Numerical Computation", in proceedings of the 1990 ACM Conference on Lisp and Functional Programming. Also see "A Compilation strategy for numerical programs based on partial evaluation," MIT Artificial Intelligence Laboratory Technical Report TR-1144, July, 1989.

[2] R. Surati, "A Parallelizing Compiler Based on Partial Evaluation" S.B. Thesis, MIT, 1992.

O^2SA Arrays for Fast Sparse Matrix Decomposition

Ricardo Telichevesky [1]
MIT Rm 36-897
ricardo@rle-vlsi.mit.edu

There has been great interest in exploiting the speed of parallel and pipelined machines in order to accelerate sparse matrix factorization. This task is difficult due to the irregular structure of most sparse matrices, which demands a complex sequence of instructions and data access in order to properly match the source and target row elements during a row update. In the following, we discuss the O^2SA technique, which combines scheduling and storage allocation techniques to enhance multiprocessor efficiency.

The *Scatter-Gather* approach, used in the YSMP code [1], proposes a solution to the matching problem by scattering the elements of a target row into a vector of size n. Computers with pipelined indirect addressing could then perform the source-target match in constant time. After all the updates to a given target row are finished, its elements can be gathered back in a dense vector. Figure 1, which depicts the processor utilization for the factorization of a test matrix, suggests that the *Scatter-Gather* method exhibits poor performance on a parallel machine.

The OSA representation of a sparse matrix [2] is an effective solution to the source-target element matching problem during update operations. The row sparsity is exploited to share the memory efficiently, by overlapping the scattered rows into a single linear array. The rows are shifted by some *offset* in such a way that the nonzero elements of one row would fill-up the nonused elements of another. During the update, elements a_{ij} of the source row are accessed sequentially, and the corresponding column indices j are simply added to the target row k offset to compute the address of the matching target element a_{kj}. This technique allows more concurrency, as shown in Figure 1.

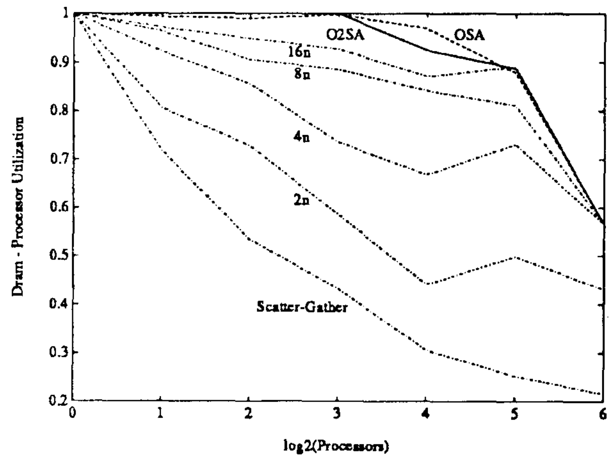


Figure 1: Processor Utilization for Test Matrix(Dram)

[1]Faculty supervisors: Prof. Jacob White and Prof. William Dally. This work was supported in part by the CNPq-Brazil under contract 205541-88.7.

We are also interested in exploiting the *locality of reference* for high execution speed in each individual processor. In order to achive this objective, we must restrict the number of active target rows per processor, keeping them in a small, very fast cache. Figure 1 shows the processor utilization for different cache sizes, n (or Scatter-Gather), $2n$, $4n$ and so on. OSA corresponds to the limit case, where the entire OSA-form matrix fits in the cache.

Instead of restricting the active set to R target rows, we envision the cache space as a dynamic OSA structure, which evolves with the execution of the sparse matrix decomposition. In the beginning of the factorization, hundreds of small rows are scattered and overlapped, fitting in a small cache of size n, allowing a large degree of freedom to the scheduling mechanism, which in turn allows a high degree of parallelism. A small cache, on the other hand, allows a very high execution speed in each processing element. After all the updates to a target row are finished, its elements are gathered back in a dense vector and possibly other targets can be scattered in the cache. The factorization proceeds in this fashion, dynamically trading available cache space for concurrency. We call this technique Overlapped-overlapped Scatter Array (O^2SA). The solid line in Figure 1 depicts the estimated processor utilization for the O^2SA technique. It achieves almost the same degree of parallelism as the OSA case, but using a very small cache, comparable to the size required by the Scatter-Gather technique.

$$\begin{array}{c|cccccccc|}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
1 & X & & & & & & X & \\
2 & & X & & & X & & X & \\
3 & X & & X & & & & X & \\
4 & & & & X & X & & & \\
5 & & X & & & X & & X & \\
6 & & & & & & X & X & X \\
7 & & & & & & & X & X \\
8 & X & X & X & & X & X & X & X
\end{array}$$

Row	Offset
1	●
2	●
3	2
4	●
5	-1
6	●
7	●
8	0

when	1	2	3	4	5	6	7	8	9
initial	a_{52}	●	a_{31}	a_{55}	a_{33}	a_{57}	●	●	a_{37}
after $N5$	●	●	a_{31}	●	a_{33}	●	●	●	a_{37}
after $N3$	a_{81}	a_{82}	a_{83}	●	a_{85}	a_{86}	a_{87}	a_{88}	●

Figure 2: O^2SA Representation of a Matrix

Figure 2 shows the O^2SA evolution for of a small test matrix. The algorithm starts with rows 3 and 5 scattered in the array, and row 8 is only scattered after their processing is finished. This example illustrates the *adaptive* utilization of the cache space, depending on the structure of the rows and the scheduling heuristic.

References

[1] S. C. Eisenstat, M. C. Gursky, M. H. Schultz and A. H. Sherman, *Yale Sparse Matrix Package II: The Nonsymmetric Codes*, Yale University Computer Science Department Research Report 114, 1977.

[2] P. Sadayappan and V. Visvanathan, "Parallelization and Performance Evaluation of Circuit Simulation on a Shared Memory Multiprocessor", *IEEE Trans. Comp,* Vol. 37, No. 12, Dec. 1988

A Sparc-based Processing Element for the NuMesh

Russell Tessier [1]
MIT Computer Architecture Group
NE43-616
tessier@lcs.mit.edu

Project NuMesh is an experimental, highly scalable interconnect designed to replace the backplane bus as a major hub of digital communication [3]. A NuMesh system consists of a grid of interconnected modules, each possessing a network interface and a processing element [2]. Computational flexibility is achieved by maintaining a standard network interface throughout the system while varying processing elements as computing needs change.

A NuMesh processing element based on the Sparc architecture, the NuSparc Element, has been designed and implemented. This element addresses a need for greater general-purpose computational support in NuMesh systems and provides a mechanism for straightfoward access to large quantities of low-cost, volatile storage. A parallel configuration of these processing elements will be used to judge the NuMesh as a digital interconnect.

The NuSparc Element logically consists of four distinct subsystems: a processing unit, a data storage unit, a status unit, and a network interface unit. These units communicate with each other through an industry-standard, sixty-four bit channel referred to as Mbus. Due to space constraints a non-coherent version of the Mbus protocol is used.

A Sparc CPU consisting of an integer unit, a floating point unit, a cache/memory controller, and 64 kbytes of cache memory serves as the NuSparc processing unit. CPU chip dies are enclosed in a 256-pin multi-die package manufactured by Ross Technology [1]. The CPU operates at clock speeds up to forty MHz.

The data storage unit consists of a DRAM controller and either eight or thirty-two Mbytes of DRAM. Block data transfers of thirty-two bytes may be made between DRAM and cache storage. The DRAM data path is enhanced with parity signals to provide for error detection.

NuMesh network status may be obtained from the status unit. Status signals indicate the availabilty of network data and bandwidth. The network interface unit may be used by the NuSparc Element to transact data with a NuMesh network interface. Data exchanged with the network interface is held by first-in, first-out (FIFO) storage.

The NuSparc Element may operate asynchronously with respect to other processing elements in a NuMesh system. Communication between these elements must be coordinated in software. Software implementations based on common computational models such as shared memory and message passing are planned for the near future.

An operational NuSparc Element is under analysis at the MIT Laboratory for Computer Science. A comprehensive NuSparc software environment is currently being developed. Future plans call for the consolidation of several NuSparc Element units into a single VLSI package.

[1]This research is supervised by Professor Stephen A. Ward. Partial support has been provided by Amp Incorporated, AT&T, Cypress Semiconductor, LSI Logic Corporation, and Sun Microsystems.

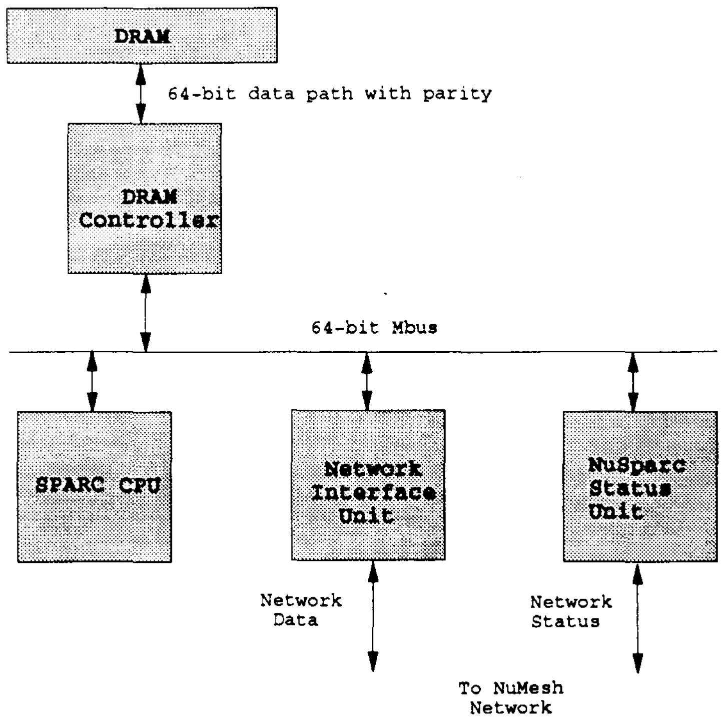


Figure 1: Block Diagram of the NuSparc Element

References

[1] Cypress Semiconductor, *CYM6111 Multi-die Package CPU Data Sheet*, Cypress Semiconductor, January 1992.

[2] Stephen Ward, "The NuMesh: A Scalable, Modular, 3D Interconnect." MIT Laboratory for Computer Science's Computer Architecture Group, internal document, February 15, 1989.

[3] Stephen Ward, "Toward LegoFlops, Recognizing Space in The Digital Abstraction." MIT Laboratory for Computer Science's Computer Architecture Group, internal document, January 24, 1991.

Competitive Fault-Tolerance in Area-Universal Networks

SIVAN TOLEDO[*]
sivan@theory.lcs.mit.edu
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

A universal network is a network that can simulate any other network which uses the same resources with only a small slowdown. In this context, a fault-tolerant universal network is a network that can simulate any other network which uses the same resources, even after both networks have undergone the same amount of damage. We study universal networks in which the measure of resources is the *layout area* required for the network.

The Fat-Tree network introduced by Leiserson can be tailored to be area-universal. This means that a Fat-Tree laid out in a square of area A can simulate any other network laid out in the same area with $O(\log A)$ slowdown. Once we obtain such a result, we need not overly concern ourselves with the suitability of the Fat-Tree for specific tasks, such as image processing or matrix multiplication, since we know that it can perform almost as well as any special purpose network of the same size. We propose a new area-universal network, show that it has some fault tolerance properties, and obtain some impossibility results that imply that a much more fault-tolerant universal network cannot be constructed.

Our goal is to design a network which is almost as efficient and fault tolerant as any other network laid out in the same area. We would like our network to be area-universal, and to retain its universality even after it is damaged. That is, we require that the remaining undamaged parts of the network will be area universal for the remaining undamaged layout area (see Figure 1). Unfortunately, this goal is too ambitious, and we prove it impossible to achieve.

Our proposed network, the *mesh of ladders* is constructed from n ladders (see Figure 2) laying horizontally one above the other, and n vertical ones, laying side by side (see Figure 3). The vertical ones, called column ladders, share the leaves with the horizontal row ladders. This network is similar to Leighton's mesh of trees network, with the trees replaced by ladders, which are very similar to X-trees.

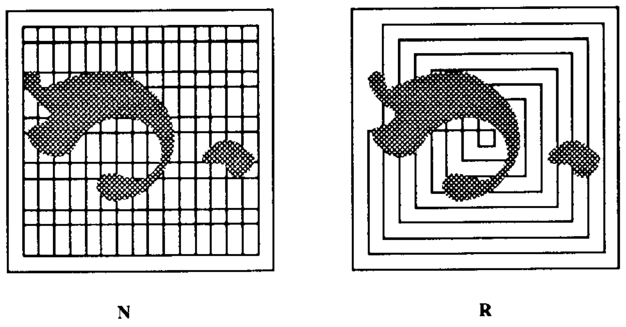


Figure 1: We require that the universal network N be able to continue to simulate the network R even after both have undergone similar damage.

The advantage of the ladder over a tree is that on a binary tree, a message traveling from node i to node j to the right of it may be moving left at times. There is a very simple routing rule on the ladder that avoids this problem. Therefore the ladder has some fault-tolerance — if a message needs to be routed between two leaves, and the part of the ladder between them is intact, then it is possible to route the message. Let us now assume that faults happen in *blocks*, or squares, whose side is the same side as the width of the ladder in the mesh of ladders. This assumption meshes well with the universality

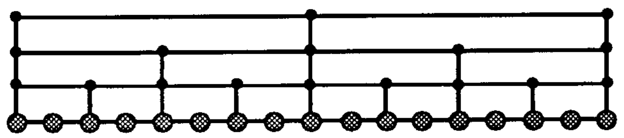


Figure 2: A layout of a ladder graph. The gray nodes are called leaves, and the smaller black nodes are routing nodes.

[*]Supervised by Charles E. Leiserson. This research was supported in part by the Defense Advanced Research Projects Agency under Grant N00014–91–J–1698.

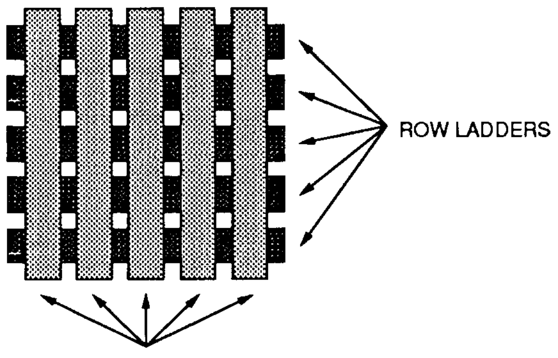


Figure 3: A sketch of a mesh of ladders with 5 row and 5 column ladders.

arguments. Our network can simulate the other network with a small slowdown any other network that suffered a slightly worse damage. By following the layout of the wires in the simulated network, it is possible to simulate any network in a fault tolerant manner. If the mesh of ladders cannot deliver a message because a ladder is damaged, then the wire carrying the message in the original network is damaged too, and the simulated network also cannot deliver the message. The slowdown is small only for network layouts in which the number of bends in each wire is small however, since whenever there is a bend in a wire, we need to switch the message from a row ladder to a column ladder or vice versa.

We come short of our goal at two points. One is the assumption that faults happen in blocks, and the other is that our network is not fault-tolerant and area-universal over *all* networks, but only over networks with a small number of bends in each wire. We have been able to prove that both points cannot be overcome by any other network. Regarding the first point, we show that under the assumption that an area universal network must exhibit the same I/O behavior as the simulated network, no area universal network exists that can fit in a sufficiently narrow rectangle. Thus, if we allow any shape of damage to occur, we might be left with such a shape, a narrow rectangle. But since there is no area universal network in such a shape, it cannot be that the remaining parts of our network can simulate any other network in the same area. The impossibility result relies on the following lemma.

Lemma 1 *Let G be a graph with n nodes and diameter at most $n^{1-\epsilon}$ for some fixed $1/2 < \epsilon < 1$, and assume $n \geq n_0$ for some constant n_0. Then G contains a subgraph T which is homeomorphic to a complete binary tree of height $(\epsilon - \frac{1}{2}) \log n = \Omega(\log n)$.*

G cannot be laid out in an area in which its subgraph T cannot be laid out. Using a result of Leiserson that shows that a complete binary tree cannot be laid out in a narrow rectangle, and showing that an area universal network in a narrow rectangle must have a small diameter, we obtain a contradiction.

The other and more general impossibility result is obtained in a different way, which is based on combinatorial counting arguments. We count the number of possible faults, and the number of ways of overcoming faults in a given network. We find out that there are more faults then ways to overcome them, and hence there must be some faults that the network cannot tolerate while still being able to simulate any other network with a small slowdown. Again we assume that the I/O behavior of the simulated network has to be preserved. This result is expressed in the a theorem in the form of a tradeoff between the number of faults, the size of faults, and the smallest slowdown that can be achieved.

Theorem 2 *Let the functions p, t, b, δ satisfy*

1. $p(n) \log \delta(n) < t(n) \log t(n)$ *and*
2. $t(n) \leq \frac{n}{4b(n)}$

for all $n > n_0$ for some constant n_0. For any network layout N in an area of $n \times n$ with maximum degree $\delta(n)$, there is a way of damaging at most $16t^2(n)$ blocks of size $b(n) \times b(n)$ each and a network R laid out in the same $n \times n$ area, such that N takes more than $p(n)$ time to simulate a step of R, under a geometric mapping of terminals.

References

[1] S. Toledo, Competitive Fault-Tolerance in Area-Universal Networks, *Proceedings of the 4rd Annual ACM Symposium on Parallel Algorithms and Architectures*, 1992, to appear.

Software-Managed Variable-Size Contexts for Multithreading

Carl A. Waldspurger*
MIT LCS Parallel Software Group
e-mail: carl@lcs.mit.edu

1 Overview

Multithreading is an important technique for tolerating latency in multiprocessor systems. Support for multiple contexts and rapid context switching permits high latency operations such as remote memory references and synchronization events to be overlapped with computation, which improves processor utilization.

This paper presents a new mechanism that efficiently supports multiple variable-size processor contexts with minimal hardware support. It adheres to the RISC philosophy [Pat85] by maintaining a simple processor architecture and relying upon the compiler and runtime system to manage the allocation and use of contexts.

Instead of statically dividing contexts in hardware, the division of the register file into contexts is managed in software. Because the size of contexts is not dictated by the hardware, the register file can be organized into a collection of contexts with varying sizes. This provides considerable flexibility in the use of the register file to support multithreading.

Since the optimal number of contexts needed to maximize processor utilization is application-dependent [Saa90], this flexibility provides an opportunity for significant performance improvements. For example, the register file can be divided into a small number of large contexts, as is conventionally done in hardware. Alternatively, the register file can be divided into a large number of small contexts, providing support for many fine-grain threads. Finally, the register file can also be divided into a diverse combination of context sizes, supporting a mix of both coarse and fine-grain threads.

2 Hardware Support

A *register relocation mask* (RRM) is maintained in a special hardware register that can be set via a special LDRRM instruction. The RRM register requires $\lceil \lg n \rceil$ bits for a processor architecture with n general registers.

RISC architectures typically employ a *fixed-field decoding* scheme in which register operands are always specified at the same location within an instruction [Pat90]. During every instruction decode, a bitwise OR operation is performed with each of the instruction's register operand fields and the RRM, yielding *relocated*

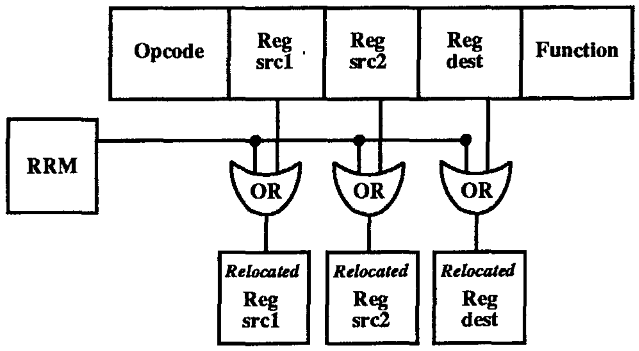


Figure 1: Register Relocation Hardware

register operand fields. After the instruction decode phase, no additional work needs to be performed.

The only other hardware change that may be necessary is to widen the internal paths that carry the register operands specified by an instruction. This is because a relocated register operand requires $\lceil \lg n \rceil$ bits to address the entire register file, while an original register operand may only be able to address a smaller portion of the register file, due to limitations on the width of a machine instruction. Such a constraint on the number of addressable registers would also place an upper bound on the size of a single context, which we will denote by s_{max}.

3 Software Support

3.1 Context Allocation

A context can be allocated with size 2^k registers, for any $k \geq 0$. However, the maximum context size is limited to s_{max} by the number of address bits used for register operands. Also, the minimum context size should be large enough to maintain some state other than a program counter. For example, practical context sizes for an architecture with 256 registers and 6-bit register operands would be 4, 8, 16, 32, and 64 registers.

Context allocation is performed entirely in software, and is thus extremely flexible. One option is to partition the register file statically into contexts (with identical or differing sizes) for a particular application, making allocation and deallocation extremely cheap. Another option is to partition the register file dynamically into contexts of varying sizes as needed.

As a proof of concept, we have coded general-purpose dynamic context allocation and deallocation routines

*Supervised by Professor William E. Weihl. E-mail: carl@lcs.mit.edu. MIT office: NE43-521a. Supported in part by an AT&T USL Fellowship, the NSF under grant CCR-8716884, DARPA under contract N00014-89-J-1988, and by an equipment grant from DEC.

for a RISC architecture with 128 registers. The implementation employs simple shift and mask operations to binary search an allocation bitmap. General-purpose allocation executes in approximately 25 RISC cycles in the worst case, and general-purpose deallocation requires only 2 RISC cycles.[1]

3.2 Context Switching

After a scheduler has chosen the next thread t to run, it performs a context switch to t's loaded context:

- Store the current program counter in a register associated with the current context.[2]
- Execute a `LDRRM` to switch to t's context.
- Jump to the program counter stored in t's context.

If t is not associated with any loaded context, then it must first be loaded as described below.

3.3 Context Loading

The runtime system can provide separate context load (and unload) routines for each supported context size s. Each routine would simply load (or unload) all registers numbered 0 to $s-1$. The `RRM` automatically provides the necessary relocation for the active context, or can be explicitly loaded to specify another context.

3.4 Compiler Support

Compilers can essentially generate code as usual, and may assume that the available registers are numbered from 0 to $s_{max} - 1$. Although the compiler is permitted to use all s_{max} registers, many threads will require fewer registers.

For each thread, the compiler must inform the scheduler of the number of registers that the thread requires. By guaranteeing not to use any additional registers, the compiler – not the hardware – is responsible for ensuring protection among thread contexts.

4 Extensions and Future Work

We have also devised a related approach to multithreading that requires *no* hardware support, and can be used with many existing processors. The basic idea is to have the compiler generate *multiple versions* of code that use disjoint subsets of the register file. Thus, register relocation is effectively performed statically at compile-time. This scheme has the obvious disadvantage of code expansion. However, the restrictions on context sizes no longer apply, and *any* partitioning of the register file is possible.

Another interesting issue is the tradeoff between improving processor utilization and exacerbating cache interference as the number of contexts is increased [Aga91]. We are currently investigating methods for adaptively limiting the number of contexts at runtime.

[1]If an operation such as the J-machine's FFB instruction is available that can find the first bit set in a word [Noa92], then general-purpose allocation can be performed in fewer than 10 RISC cycles.

[2]It may be convenient to adopt a convention of always storing the PC in a fixed register relative to a context, such as register 0.

5 Related Work

A number of processor architectures that include multiple hardware contexts have been proposed. Finely multithreaded processors, exemplified by the Denelcor HEP [Smi78], execute an instruction from a different thread on each cycle. Coarsely multithreaded processors, such as APRIL [Aga90], execute larger blocks of instructions from each thread, and typically switch contexts only when a high-latency operation occurs. Our register relocation mechanism supports coarse multithreading, but permits a more flexible organization of the register file by managing contexts in software.

A completely different approach is the Named State Processor [Nut91], which replaces a conventional register file with a *context cache*. The context cache binds variable names to individual registers in a fully associative register file, and spills registers only when they are immediately needed for another purpose. Since our register relocation mechanism supports variable-size contexts, it permits a binding of variable names to contexts that is finer than conventional multithreaded processors, but coarser than the context cache approach.

6 Conclusions

We have presented a new mechanism that efficiently supports multiple variable-size processor contexts with minimal hardware support. Simple register relocation hardware, combined with software support, provides significant flexibility in the use of the register file to support multithreading. We are currently investigating software methods to adaptively control the number of loaded contexts and optimize performance.

[Aga90] A. Agarwal, B. Lim, D. Kranz, and J. Kubiatowicz. "APRIL: A Processor Architecture for Multiprocessing", *Proc. 17th Annual International Symposium on Computer Architecture*, June 1990.

[Aga91] A. Agarwal. "Performance Tradeoffs in Multithreaded Processors", Technical Report MIT/LCS/TR-501, MIT Lab for Computer Science, April 1991.

[Noa92] M. Noakes. "MDP Programmer's Manual", Concurrent VLSI Architecture Memo #40, MIT AI Lab, 1992.

[Nut91] P. Nuth and W. Dally. "A Mechanism for Efficient Context Switching", *Proc. IEEE Conference on Computer Design*, October 1991.

[Pat85] D. Patterson. "Reduced Instruction Set Computers", *Communications of the ACM*, January 1985.

[Pat90] D. Patterson and J. Hennessy. *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, 1990.

[Saa90] R. Saavedra-Barrera, D. Culler, and T. von Eicken. "Analysis of Multithreaded Architectures for Parallel Computing", *ACM Symposium Parallel Algorithms and Architecture*, July 1990.

[Smi78] B. Smith, "A Pipelined, Shared Resource MIMD Computer", *Proc. International Conference on Parallel Processing*, 1978.

A Model of a Hierarchical Cache Coherence Protocol[1]

Deborah A. Wallach
Room 614
Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139
kerr@ai.mit.edu

As the number of processors that are connected together to form multiprocessors grows, efficiently supporting a shared memory programming model becomes difficult. We have designed PHD, a hierarchical directory-based cache coherence protocol, to allow shared-memory support for systems containing massive numbers of processors. We have created an analytical model of the protocol in order to study the behavior of the protocol on machine configurations too large to simulate.

The Protocol for Hierarchical Directories (PHD) supports a shared memory model by synthesizing a global shared memory from the local memories of processors. Two of the global primitives of the protocol are modeled: `read` and `write`. The protocol is designed to work on a tree (the hierarchy); this tree is mapped onto the actual multiprocessor topology.

Read-only copies of blocks may be stored in the caches of any number of processors. To find a block, a processor asks its parent for a copy. The parent must know which of its children has copies. If none do, it forwards the message upwards. Otherwise, it forwards the read message to any child processor which already has the block. Read operations can therefore be satisfied locally.

Write operations involve finding all of the copies of a block in the system and deleting them. Only the nodes in the smallest subtree completely containing all copies of the block are involved in the write process. The *owner* of the block transfers ownership to the node requesting the write. Acknowledgments of deletion from all of the nodes which previously had copies are combined, and exactly one acknowledge message is sent to the node requesting the write.

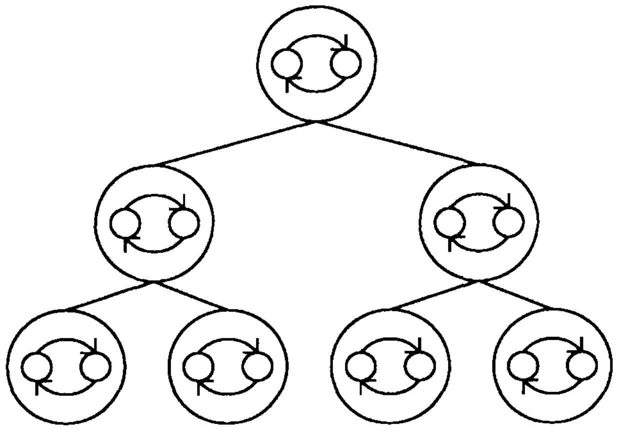


Figure 1: Every node in the hierarchy is modeled as a finite state machine.

In the model, we follow average read and write requests and determine how far up the hierarchy they must travel in order to be satisfied. From the average read and write heights, we can approximately determine the average number of messages sent in order to satisfy read and write requests.

The inputs to the model include the number of levels in the tree hierarchy, the radix of the tree, the frequency of writes in the memory request stream, and the type of sharing which occurs. Associated with each type of sharing are additional parameters which allow us to approximate the average sharing characteristics.

The model includes a finite state machine at every node, as shown in Figure 1. This machine models the state of a single cache block. The FSM, shown enlarged in Figure 2, contains only

[1]William Dally both supervised and contributed to this work. The research described in this paper was supported in part by the Defense Advanced Research Projects Agency under contracts N00014-88K-0738 and N00014-87K-0825 and in part by a National Science Foundation Presidential Young Investigator Award, grant MIP-8657531, with matching funds from General Electric Corporation and IBM Corporation and by a Office of Naval Research Graduate Fellowship, grant N00014-90-J-1778.

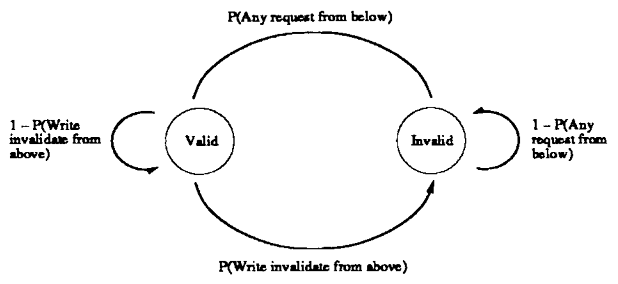

P(Any request from below)
1 – P(Write invalidate from above)
Valid
Invalid
1 – P(Any request from below)
P(Write invalidate from above)

Figure 2: Every node in the hierarchy is modeled by a separate copy of this finite state machine.

two states: *valid* and *invalid*. A node with its FSM in the *invalid* state has no copy (if it is a leaf node), or no descendants with copies (if it is not a leaf node). By definition, the top node of the tree is always in the *valid* state.

A leaf node in the *invalid* state transitions to *valid* if it makes a read or write request. It transitions from *valid* to *invalid* if any other node makes a write request. This transition corresponds, in the actual protocol, to the write invalidate which the leaf would eventually receive. This FSM is identical to the one Anant Agarwal proposed in [1].

The transitions for a node in the interior of the tree are similar to that for the leaf node. The transition from *invalid* to *valid* occurs if a descendant leaf node makes a request. The transition from *valid* to *invalid* occurs if a non-descendant leaf node makes a write request.

We consider the accesses occurring to the address under consideration from the point of view (POV) of one leaf node and all of its ancestors in the tree. We designate those nodes as *POV* nodes.

From the previously described system, we can calculate the probabilities that nodes in the hierarchy will have copies of the address. We can use these probabilities to directly calculate the average read and write heights.

Three applications are being studied. One is a uniform reference pattern, meaning that every processor is equally likely to reference a particular piece of data. The second is a basic relaxation, where in an iteration every point in an n-dimensional mesh updates its own value by a function of the value of its $2n$ neighbors. The third application attempts to simulate groups or *clusters* of processors working on data. Clusters are said to *own* data. The processors within a given cluster are made more likely to reference data owned by the cluster than data owned by other clusters. This model is similar to the one proposed by Qing Yang, in [2].

We have written a trace-driven simulator to model the operation of the PHD. We use synthetic address traces matching the three applications as input to the simulator. The output statistics from the simulator are then compared to the predictions of the model, in order to verify the model. Once the model has been fully characterized, we will be investigating the behavior of the protocol for machine sizes we cannot simulate.

References

[1] A. Agarwal, "A locality-based multiprocessor cache interference model," VLSI Memo MIT VLSI Memo 89-565, Massachusetts Institute of Technology, 1989.

[2] Q. Yang, "Performance analysis of a cache-coherent multiprocessor based on hierarchical multiple buses," in *PARBASE-90 International Conference on Databases, Parallel Architectures and Their Applications* (N. Rishe, S. Navathe, and D. Tal, eds.), pp. 248–257, IEEE Computer Society Press, 1990.

[3] D. A. Wallach, "A scalable hierarchical cache coherence protocol." SB Thesis, MIT, May 1990.

Embedding Leveled Hypercube Algorithms into Hypercubes

(Extended Abstract)

David Bruce Wilson *

dbwilson@mit.edu
Department of Mathematics,
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

Introduction

In many parallel computers with hypercube networks, the processors can communicate simultaneously along all the wires connected to them. Often the software that these machines run use only a few hypercube dimensions at a time. Many hypercube algorithms are leveled: they use only one dimension at a time (see for instance [2]). A programming language may allow the programmer to specify communication in a mesh, but along only one mesh axis at a time. If the mesh is embedded into the hypercube using Gray codes, then only a few hypercube dimensions are used at a time. If the mesh or leveled hypercube algorithm is embedded into the host hypercube in the straightforward way, much of the machine's bandwidth will be unused.

There are embeddings that better utilize the machine's bandwidth; each guest edge is simulated by multiple paths of wires in the host, so the communication throughput of the guest edges is larger than that of a host wire. We will say that the guest edges are *accelerated.* At one extreme, embeddings that accelerate the edges of a cycle have been worked out [1]. The cycle embeddings can be composed to make a mesh embedding. This paper concentrates at the opposite extreme, and gives embeddings which accelerate the edges of a hypercube, assuming only one guest dimension is used at a time. If only one mesh axis is used at a time, then in some cases the above cycle and hypercube embeddings can be combined to make a better mesh embedding [3].

Before proceeding, I would like to point out that this paper is primarily of theoretical interest. In practice, communication throughput is significant when there are many more virtual processors than physical processors. But in this case, many leveled algorithms can be made to better utilize the machine's wires with pipelining techniques. It is surprising, however, that embedding a hypercube algorithm into a hypercube in the straightforward way can be suboptimal when the algorithm uses one dimension at a time.

*Supervised by Charles E. Leiserson. This research was supported in part by the Defense Advanced Research Projects Agency under Contracts N00014-87-0825 and N00014-91-J-1698, and in part by an ONR-NDSEG fellowship.

An Example

There is an embedding $\mathcal{E}_3$ which accelerates each edge of the hypercube Q_3, assuming that only one guest dimension is used at a time. Recall that the nodes of the n-dimensional hypercube Q_n are labeled by a string of n address bits. There is an edge between two nodes if their addresses differ by one bit. In this paper all edges are undirected.

An embedding $E : G \hookrightarrow H$ bijectively maps the guest vertices of graph G to the host nodes of graph H. An embedding also maps each guest edge to a network of host wires which simulate that edge.

The node map of embedding $\mathcal{E}_3$ is given by

$$\mathcal{E}_3(b_2b_1b_0) = \begin{cases} b_1b_0b_2, & b_2 + b_1 + b_0 = 2; \\ b_2b_1b_0, & \text{otherwise,} \end{cases}$$

where $b_2b_1b_0$ are the guest address bits and $\mathcal{E}_3(b_2b_1b_0)$ are the host address bits. The networks for the dimension-0 guest edges are shown in Figure 1. Each individual dimension-0 edge is simulated by a host network which transmits 3/2 packets per unit time. Furthermore, the host graph is able to simulate each of these edges simultaneously. Informally we will say that $\mathcal{E}_3$ accelerates dimension 0 by 3/2.

The host networks for the other guest edges are easily constructed because of symmetry in the node map. We may cyclically shift the guest address bits, apply $\mathcal{E}_3$, and shift the bits back; the node map would remain unchanged. To get the networks that simulate guest edges crossing other dimensions, we may shift the address bits so that the edges cross dimension 0, take the appropriate dimension-0 networks, and shift the address bits back.

If $E : Q_n \hookrightarrow Q_n$ is an embedding for which each guest edge is simulated by a host network that can transmit a packets per unit time, we will say E has acceleration a. If for each dimension d, the host hypercube can simultaneously simulate all guest edges crossing dimension d, then E will be called a *leveled* embedding. Therefore $\mathcal{E}_3$ is a leveled embedding with acceleration 3/2.

There is a different leveled embedding $\mathcal{E}_6 : Q_6 \hookrightarrow Q_6$ which has acceleration 2. Embedding $\mathcal{E}_6$ has the additional property that no host wire is multiplexed between guest edges. This embedding is described in [3].

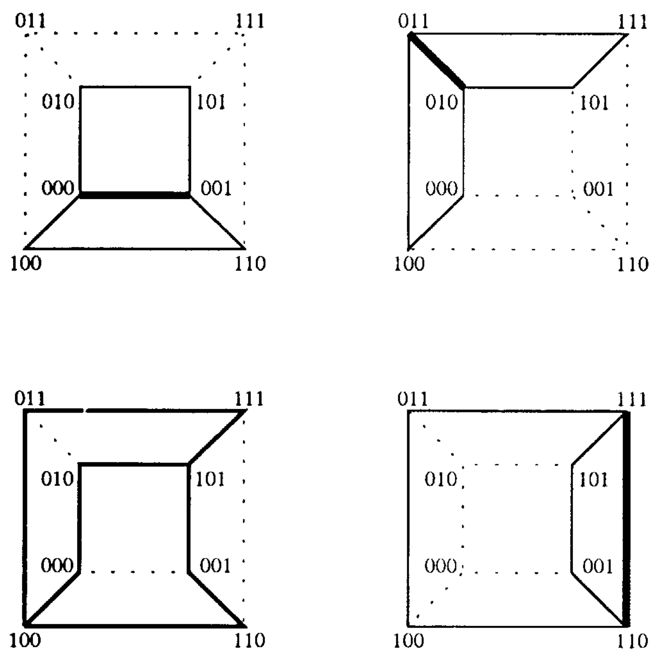


Figure 1: The flow networks simulating the dimension-0 guest edges. The edges $\langle 000, 001\rangle$, $\langle 010, 011\rangle$, and $\langle 110, 111\rangle$ are each simulated by a host path of length one (carrying one packet per unit time) and two additional paths of length three (each carrying one-quarter packet per unit time). The edge $\langle 100, 101\rangle$ is simulated by three paths of length three (each carring one-half packet per unit time).

Asymptotics

Leveled embeddings can be composed with one another to make new embeddings. The following theorems are proved in [3], I simply state them here.

Theorem 1 *If $E_1 : Q_p \hookrightarrow Q_p$ and $E_2 : Q_q \hookrightarrow Q_q$ are leveled embeddings with acceleration a_1 and a_2, then $E_2 \otimes E_1$ is a leveled embedding from Q_{pq} to Q_{pq} with acceleration $a_1 a_2$. If E_1 and E_2 don't multiplex the host wires, then neither does $E_2 \otimes E_1$.*

By repeated application of Theorem 1, using a leveled embedding $E : Q_p \hookrightarrow Q_p$ with acceleration a, we get for $n = p^m$ a leveled embedding from Q_n into Q_n with acceleration $a^m = n^{\log_p a}$. For general n we lose a log factor.

Theorem 2 *If $E : Q_p \hookrightarrow Q_p$ is leveled with acceleration a, then for each n there is a leveled embedding E_n from Q_n into Q_n with acceleration $\Omega\left((n/\log n)^{\log_p a}\right)$. If E does not multiplex host wires, then neither does E_n.*

In particular, there is a leveled embedding from Q_n into Q_n which does not multiplex host wires and has acceleration $\Omega\left((n/\log n)^{\log_6 2}\right) = \omega(n^{0.386})$. The next theorem provides an upper bound on the accleration of any leveled embedding.

Theorem 3 *If $E : Q_n \hookrightarrow Q_n$ is a leveled embedding, then E accelerates Q_n by at most $O(n/\log n)$.*

Practical Hypercubes

Table 1 gives the acceleration of the "best" leveled embedding I have found for Q_n, where n is not too large. One leveled embedding is considered "better" than another if it has higher acceleration; the multiplexing of host wires is ignored by this metric. This table also gives an upper bound on the acceleration of any leveled embedding for Q_n. Each of these upper bounds is in fact also an upper bound on the amount by which any single dimension can be accelerated.

n	a	u	n	a	u
1	1	1	11	2	4.33
2	1	1	12	2.25	4.67
3	1.5	1.5	13	2	5
4	1.5	2	14	2	5.33
5	1.5	2.33	15	2.25	5.67
6	2	2.67	16	2.25	6
7	2	3	17	2	6.33
8	2	3.33	18	3	6.67
9	2.25	3.67	19	2	7
10	2	4	20	2.25	7.33

Table 1: As a function of n, a is the acceleration of the "best" leveled embedding for Q_n I have found, and u is an upper bound on this acceleration.

Acknowledgements

I would like to thank Charles Leiserson and Tom Leighton for the technical advice they gave me while I worked on this problem. I also found a program written by Cliff Stein to be useful for exploring embeddings of small hypercubes.

References

[1] D. S. Greenberg and S. N. Bhatt. Routing multiple paths in hypercubes. In *Proceedings of the 2nd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 45--54, 1990.

[2] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgan Kaufmann Publishers, 1992.

[3] D. B. Wilson. Embedding leveled hypercube algorithms into hypercubes (extended abstract). To appear in *Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, 1992.

"What are the Grand Challenge Problems in Supercomputing Technology?" Panel Discussion

Moderator: Eric Brewer
Scribe: David Chaiken

1 Introduction

In 1900, David Hilbert proposed twenty-three problems covering all areas of mathematics that guided the field for decades. These problems served a driving force for mathematicians, providing instant fame (and fortune) to the people who could solve them, or even parts of them. The goal of this panel session was to propose several grand challenge problems, similar in spirit to those of Hilbert[1]:

> [A] problem should be difficult in order to entice us, yet not completely inaccessible, lest it mock at our efforts. It should be to us a guide post on the mazy paths to hidden truths, and ultimately a reminder of our pleasure in the successful solution.

To this end, six graduate students were asked to identify some of the long-term goals in the field of supercomputing technology. A summary of each of their talks follows, in order of presentation.

2 Panel Talks

2.1 Carl Waldspurger: Portable Resource Management

Optimizing the performance of a parallel application is hard even when a programmer disregards the proliferation of different architectures. When faced with the task of moving from one parallel system to another, the task becomes nearly impossible. Providing the abstraction mechanisms that will allow programmers to easily migrate applications over a range of parallel architectures is a significant problem to solve.

While there is no cohesive "big picture" that addresses the portability problem, a huge literature on computational resource management exists. Static analysis by compilers promises to address the problem in a large class of programs, but this technique is inadequate for data-dependent applications. Dynamic feedback mechanisms implemented in runtime systems provide automatic resource management, but they tend to impede human control when it is necessary. Some systems leave the multidimensional resource allocation problem to the programmer, thereby permitting human control, but forcing the programmer to perform tasks that might be done automatically. (For example, virtual memory relieves programmers of the dreaded overlay problem.)

In order to make parallel applications portable, it will be necessary to develop language constructs that allow programmers to express performance tradeoffs without having to write low-level resource allocation. The first step is to develop a uniform model of parallel computation that incorporates both tasks and resources into a coherent framework. Using this model, the cost of consuming each resource should be quantified in terms of uniform, abstract units that may be used for a variety of different architectures. Allowing the programmer to express the contention between tasks for resources is a critical feature of this framework. The key to developing this abstraction lies in balancing the amount of human control (expressivity) with the level of detail that programmers must address (transparency).

2.2 Mike Klugerman: Fault Tolerance

Given the goal of massively parallel systems, a simple calculation shows that fault tolerance is a problem that must be solved. Consider the failure rate of a system as a whole. If we make the (optimistic) assumption that a single processor will fail at the rate of 10^{-5} failures per hour, then a machine with one million (10^6) processors will incur a rate of 10 failures per hour. In order to build this machine with a reasonable failure rate, it should be able to tolerate faults in up to 10% of its processors (10^5). The resulting failure rate for the system would then be 10^{-4} failures per hour.

Two key problems must be solved to achieve fault tolerance: Maintaining data integrity and fault tolerant routing. Maintaining data integrity involves preserving the state of memory in the presence of hardware failures. Checkpointing provides integrity by periodically copying memory to stable storage. This offline technique tends to waste time that could be spent doing processing. Replication of data using redundant hardware maintains integrity in an online fashion — at the expense of the extra hardware required to implement the scheme. The challenge in maintaining data integrity is to reduce the amount of replication that is required to tolerate faults.

Fault tolerant routing involves maintaining connections between working parts of a system, even when some of the components of the system fail. Theorists understand the behavior of routing algorithms under the assumptions of worst case or random failure models. Unfortunately, these types of models do not correspond to the behavior of real-world systems, which generally do not encounter worst case scenarios but do suffer from correlated faults. Solutions to the fault tolerant routing problem require a better understanding of the correlations between failures in real systems.

2.3 Mark Reichelt: Sparse Matrices

The problem of solving $Ax = b$ efficiently on a parallel supercomputer has not been solved completely. While efficient algorithms exist for solving dense matrices, an efficient parallel algorithm for sparse matrices does not exist. Sparse matrices, which have only a small number ($O(1)$) of elements per row of A (and lots of zeros), occur naturally in a large number of applications.

For certain types of sparse matrices with special structures, it is possible to use iterative techniques to solve the related system of equations. These techniques take advantage of the structures of certain problem domains, which are reflected in the pattern of non-zero elements in the associated sparse matrices. Unfortunately, each of these iterative techniques requires special synchronization, error calculation, and termination conditions. No existing iterative technique will work for all sparse matrices. In addition, the iterative techniques that do work for special structures tend to take a long time to run.

It is possible to use a parallel algorithm for dense matrices to solve sparse matrices. However, this method is not efficient because an fast sequential algorithm for sparse matrices exists. That is, the best serial solution for dense matrices requires $O(N^3)$ steps, while the parallel algorithm takes $O(N \log N)$ steps on $O(N^2/\log N)$ processors. This yields an efficiency of $O(N^3/((N \log N)(N^2/\log N))) = O(1)$. In contrast, the sequential algorithm for solving sparse matrices requires only $O(N^{1.5})$ steps, yielding an efficiency of $O(1/N^{1.5})$ when using the parallel algorithm for dense matrices. As a result, scientists who require solutions to sparse systems of equations challenge us to develop better parallel algorithms. Until we develop faster algorithms, they will opt for fast sequential implementations.

2.4 Madhu Sharma: Scheduling for Locality

Over the last decade, two critical problems in parallel processing have been solved. First, the issue of expressing and identifying parallelism has been addressed by a number of parallel languages with constructs that reveal implicit parallelism, allow non-strict evaluation, and permit synchronization between processes. Second, the problem of tolerating communication latency has been solved by a host of mechanisms, such as prefetching, relaxed memory models, and multithreading.

Today's challenge is to deal with the principal performance bottleneck in parallel architectures: communication bandwidth. In order to utilize this resource most efficiently, systems must minimize the bandwidth requirement of applications by maximizing the locality of communication. Current systems do not perform this task well. While some combinations of languages, compilers, and mapping schemes achieve limited success in maximizing the locality for programs with regular communication patterns, they do not allow for the dynamic effects associated with fine-grain parallel execution. Even for applications with regular communication patterns, cache hit rates remain low and demand on the processor interconnect remains high.

The problem that needs to be solved is constructing schedulers that effectively manage fine-grain parallel activity. Such schedulers should maximize both locality of reference in caches and communication locality between processors. In order to achieve this goal, it might be necessary to incorporate a model of locality into parallel languages, thereby allowing systems to take advantage of the programmer's vantage point.

2.5 Rich Lethin: The Hardware Creativity Crisis

"The Hardware Creativity Crisis" is a challenge to change our way of thinking about developing computer systems, rather than identifying a particular problem to solve. The current methodology for doing hardware research stifles creativity. Today's researchers are locked into a pattern of making incremental improvements in existing architectures or building new systems that are compatible with old ones. The process of tuning systems with stock benchmarks (*e.g.* SPECmarks) and then using the same benchmarks to guide the development of the next generation of computers results in an incestuous cycle: if we set out to build a better VAX, then that is exactly what we will build. Furthermore, by binding ourselves to the current generation of CAD tools, we confine ourselves to a single design method and to a previous generation of technology.

Instead of attempting to build yet another CDC 6600 in the latest-and-greatest technology, researchers should balance theory and tuning. We should focus on the fundamental building blocks of computer systems, avoid excessive performance tuning, construct prototypes quickly, and draw real conclusions from our prototypes. More "dark horse" projects, in the spirit of neural networks and hardware implementations of genetic algorithms, would be healthy contributions to the creative process.

2.6 Kirk Johnson: Communication is the Crux

The grand challenge in high performance supercomputing is effectively delivering the cost/performance advantages promised by parallel systems to the end user. The obstacle to this goal is the difficulty of programming large-scale parallel systems due to the problem of managing communication and synchronization costs. Current systems take one of two approaches to this problem: *explicit* or *implicit* control of communication. Explicit control, required by message passing models of computation, allows programmers to manage communication directly and results in architectures that are easy to build. However, demanding the programmer to take control complicates the job of developing applications. Implicit control, offered by shared memory models, simplifies the job of programming but results in architectures that are harder to build. The lack of human control also makes it harder to manage communication and synchronization.

In order to manage communication effectively, we need to develop hybrid models of parallel computation with the benefits of implicit control and the power of explicit control. For example, shared memory systems could be augmented with communication annotations that allow a system to exploit information about an application from the programmer as in the work on "Cooperative Shared Memory" at Wisconsin [2]. This approach would require architectures (such as Alewife) that efficiently support both message passing and shared memory models. New programming systems should also allow the programmer to express the structure and costs of communication at a high-level.

3 Open Discussion

Early in the discussion following the panelists' presentations, the participants decided that for an engineering discipline, it is impossible to propose grand challenges in the spirit of Hilbert's problems. Rather than identifying problems that have distinct solutions, the panelists proposed long term goals that may be satisfied to a lesser or greater extent but will always leave room for improvement. Several themes emerged during the presentations and the ensuing discussion.

Balancing human control and automatic resource management. Several of the panelists discussed the unresolved tradeoff between the ease of programming and the efficiency of parallel applications. This tradeoff impacts every aspect of parallel system research from models of computation to programming languages to hardware mechanisms. One participant suggested that our expectations might be too high for the efficiency of automatic systems. As in the sequential world, the user might have to sacrifice some efficiency in order to gain convenience. A good example is the efficiency/convenience tradeoff implicit in uniprocessor compiler analysis. Another participant wondered if the sequential model of computation is fundamental to human thinking. In this case, it will never be easy for programmers to write applications for large-scale parallel architectures.

Communication bandwidth is a principal bottleneck. Of all of the resources that must be managed through direct human control or by automatic mechanisms, communication bandwidth received the most attention. Judging by the emphasis of the panelists, developing methods for efficiently allocating this resource will be critical to the acceptance of large-scale parallel systems.

The role of universities in supercomputing research. Research groups in universities can not compete with the work being done in industry, nor should they try. Companies are very good at turning the incremental improvement crank and generating the latest and greatest VAX implementation. Industry also produces far more polished, production-quality software than do universities. The participants generally agreed that universities should not duplicate the efforts of industry.

The group disagreed on the exact role that universities should perform. Some students saw no problem with assisting industry by developing new analysis techniques or by proposing small changes to existing systems. Others argued that universities should provide the force to move research out of local minima by demonstrating the benefits of radically different approaches to supercomputing (as they did with the Connection Machine and VLIW architectures).

Portability. While it would be beneficial to have a standard model of parallel computation and a single language that could be supported by many parallel architectures, no consensus on the structure of the model or the constructs in the language exists. Several of the panelists proposed research that could be done in the area of portable language constructs. These ideas (and more) must be evaluated before reaching the goal of portability. At the current stage of research, diversity is important to the evolution of languages and models of computation. In the long term, the most viable programming methods should dominate the current field of candidates.

Building a user community. We can interpret Mark Reichelt's challenge in the context of a broader user community. There are a large number of users who have applications that are large enough to benefit from the cost/performance tradeoffs available from large-scale parallel systems. However, the state of the art of parallel algorithms has not yet reached the same stage as sequential algorithms. Since efficient parallel algorithms have not yet been found for many problems, potential users do not have sufficient motivation to migrate their applications to parallel systems.

Unfortunately, this situation poses a Catch-22: in order to make parallel systems efficient and easy to program, researchers need a variety of significant applications to study. On the other hand, users will not develop such applications until parallel systems are efficient and easy to program. This problem is critical to the future of research in supercomputing and led to an answer to the question, "What should we be doing at MIT?" The only confident answer from the group (besides increasing graduate student salaries) was to "open the doors [of Tech. Square] to real people who need FLOPs." Our research groups need better communication with potential users, especially those within MIT itself.

References

[1] David Hilbert. Mathematical Problems. *Bulletin of the American Mathematical Society*, 8:437–479, 1902. Reprinted in Proceedings of Symposia in Pure Mathematics. Volume 28, 1976.

[2] Mark D. Hill, James R. Larus, Steven K. Reinhardt, and David A. Wood. Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessors. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V) – to appear*, New York, October 1992. IEEE.