Multiprocessor-Based Placement by Simulated Annealing*

Saul A. Kravitz and Rob A. Rutenbar

Department of Electrical and Computer Engineering
Carnegie-Mellon University
Pittsburgh, PA 15213

Abstract

Simulated annealing methods have proven to be particularly successful in physical design applications, but often require burdensome, long run times. This paper studies the design and analysis of standard cell placement by annealing in a multiprocessor environment. Annealing is not static: we observe that the temperature parameter which controls hill-climbing in simulated annealing changes the behavior of an annealing algorithm as it runs, and strongly influences the choice of multiprocessor partitioning strategy. We introduce the idea of *adaptive strategies* that exhibit different speedups across different temperature ranges. Measured performance of parallel placement algorithms running on a multiprocessor demonstrate practical speedups consistent with our predictions.

1. Introduction

Physical design tools are often among the first to feel the pressures of increases in the density and integration of VLSI systems. Simulated annealing is an approach to combinatorial optimization problems that has proven to be particularly successful in physical design applications. In particular, annealing methods exhibit desirable solutions, often unacceptable run-times, and performance degradation with increases in circuit size; these characteristics motivate us to study hardware accelerators. This paper studies the design and analysis of standard cell placement by annealing in a multiprocessor environment. Our goal is to accelerate the placement task by exploiting parallelism inherent to annealing methods.

The key issue in parallel tool implementation is the *partitioning* of an algorithm across communicating processors. In this paper we present new observations on how the dynamic nature of the annealing process influences the choice of multiprocessor partitioning strategy. Annealing is not static: the temperature parameter which controls hill-climbing in annealing also profoundly changes the quantitative behavior of an annealing algorithm over the course of its execution. This paper shows that the dynamic character of annealing presents new opportunities for exploiting the inherent parallelism of the process. Measured performance of parallel placement algorithms running on a shared memory multiprocessor demonstrate practical speedups.

*This research was supported in part by a fellowship from the Schlumberger Foundation and by the Semiconductor Research Corporation.

2. Background

Before discussing parallel versions of placement by annealing, we review here the relevant aspects of simulated annealing algorithms. We distinguish between the *design* of annealing algorithms, and their *restructuring* for parallel implementation. Some related work on hardware accelerators for placement is surveyed.

2.1 The Design of Simulated Annealing Algorithms

Iterative improvement algorithms have long been employed in placement tasks. Starting with an initial placement, obtained by random or constructive means, we generate a sequence of proposed placement improvements, called *moves*, usually module translations or pairwise swaps. Those actually improving some metric for the goodness of the placement (typically measures of aggregate wirelength or channel congestion) are accepted, and the placement is modified. Those that worsen the placement are rejected. Such algorithms are essentially greedy. They are easily trapped in locally optimal, globally inferior solutions, since they can never accept a perturbation that worsens the solution, in hopes of climbing out of this local optimum.

Simulated annealing algorithms [4] provide just such a hill climbing mechanism. Placement moves that worsen the current solution by an amount ΔE are accepted with probability $e^{-\Delta E/T}$. T is a new control parameter analogous to temperature in the annealing of physical systems. In an optimization framework, we anneal a placement by performing standard iterative improvement steps, but now accept *some* worsened solutions according to this temperature rule. As iterative improvement proceeds, the temperature T is slowly reduced. During initial annealing, T is large; in this *hot regime* most uphill moves are accepted. Near the end of annealing, T is small; in this *cold regime* few uphill moves are accepted. The change in the fraction of moves accepted at each temperature is often referred to informally as the *statistics* of the annealing process. The fact that annealing is a dynamic process whose statistics change over the course of a run is influential in the choice of a partitioning strategy for parallel implementation.

Design of good annealing algorithms is non-trivial. For our purposes, the *design* of an annealing algorithm comprises four parts:

- **Move Set:** The set of feasible rearrangements (e.g., pair swaps), called *moves*, must be rich enough so that all reasonable solutions can be found. The moves must be relatively inexpensive to compute, since we will perform many moves (perhaps 10^8 for a large problem).

- **Cost Metric:** The metric must be incrementally computable, so that the time to evaluate each move is minimal, and must be physically meaningful: the most routable placement with the smallest area should have the least cost.
- **Annealing Schedule:** The manner in which temperature T is lowered during annealing, the temperature *schedule*, is crucial. Starting too cold, stopping too hot, or cooling too quickly all produce sub-optimal solutions.
- **Data Structures:** The ability to propose and evaluate moves efficiently hinges on a good representation for the basic objects in the problem. For placement, this means structures for modules and nets arranged so that connectivity and spatial location are quickly assessed.

In our multiprocessor work, we concentrate on the activity of partitioning an annealing algorithm for execution on a parallel machine. The basic moves, metrics, schedule and data structures of the algorithm are taken as given. Our goal is an optimal partitioning of these computations across communicating processors. The starting point is the choice of a particular serial annealing algorithm.

2.2 Choice of Serial Placement Algorithm

For this work we chose the Timberwolf algorithm for standard cell placement [6]. The algorithm is well-documented, and used in industrial applications. It compares very favorably in quality with other approaches, and most importantly, exhibits the central performance limitation of many annealing algorithms: extreme run-times for problems of useful complexity (e.g., [6] estimates 84 VAX CPU hours for a 2700 cell design). For our purposes, Timberwolf has these essential characteristics:

- **Move-Set:** The move set consists of single cell displacements, and pair interchanges, and is simplified by permitting *overlap* between cells.
- **Cost Metric:** Estimated wire length and cell overlap constitute the metric. Arrangements leading to long wires are penalized by high costs, as are physically unrealizable arrangements with high cell overlap.
- **Annealing Schedule:** A simple logarithmic schedule ($T_{new} = \alpha T_{old}$, $\alpha < 1$) with empirically determined initial and final temperatures is used.
- **Data Structures:** The cells in each row are sorted into bins according to their location within the row to allow rapid determination of cell overlap. Cells migrate from bin to bin as annealing proceeds.

We have constructed a simplified version of the Timberwolf standard cell placement algorithm, called *SerialPlace*, for use as a standard against which to benchmark our parallel algorithms. *SerialPlace* is slightly faster than Timberwolf, but converges to the same solutions and preserves the algorithm's overwhelmingly dominant feature: the time-intensive computations to propose and evaluate moves by which we progress toward a good global placement.

Figure 1 introduces an example placement problem of approximately 100 standard cells used in our parallel implementations. This example is large enough to exhibit all the interesting behavior of an annealing algorithm, but small enough to permit many benchmarks to be evaluated. Figure 1 illustrates two basic facts about placement by annealing. First, as temperature decreases (i.e., as annealing proceeds from left to right), the total system cost decreases rapidly (note the log scales). Second, the fraction of moves rejected decreases as annealing proceeds until in the very cold regime, almost all moves are rejected.

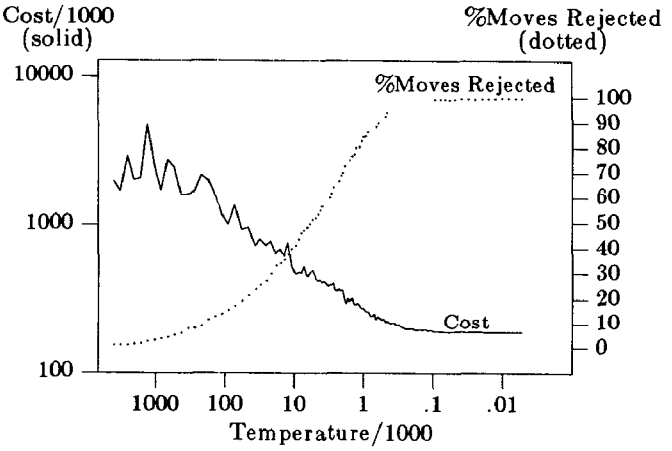

Figure 1: Cost and %Reject for Example Placement

2.3 Hardware Accelerators in Physical DA

Although hardware accelerators have been employed mainly in simulation tasks, applications in physical design have recently received greater attention. Routing, design rules checking, and placement are the tasks most studied for hardware implementation. We are aware of five studies of placement machines. All these machines share a common model of placement: modules are placed at sites in a fixed grid, and iterative improvements in the form of module interchanges are attempted to optimize some metric of placement, usually aggregate wire length.

A prototype implementation of a module interchange placement machine is discussed in [3]. The machine solves placement as a quadratic assignment problem, and incorporates a 4-stage arithmetic pipeline to evaluate the incremental change in wiring cost after a trial interchange of modules. Speedups here can be attributed to the decreased time to evaluate the cost metric after an interchange.

Two similar 2-dimensional mesh machines are proposed in [2, 9]. The idea here is to map the placement grid directly onto a large array of neighbor-connected processors. Instead of single pair-swaps, multiple parallel pair-swaps are attempted between neighboring groups, such as rows or columns. The speedup here occurs because parallel swaps provide much larger decreases in wiring cost in essentially the same time as that taken for a single swap.

Gate array placement is accelerated by a microprogrammable engine in [8]. A conventional annealing algorithm is rewritten in microcode to execute on a fast microengine attached to a microprocessor-based host. Speedups here are directly attributable to the speed of the underlying hardware and the direct microcode implementation.

Most similar to our own work is the multiprocessor placement algorithm outlined in [7]. The modules and nets of a placement task are divided into groups across a shared-memory multiprocessor. After a proposed interchange, recalculation of the cost metric proceeds in parallel on the objects stored in each processor. We note here that this is not the only feasible partitioning scheme for multiprocessor placement (see Section 3).

In contrast to [3, 8], the work described in this paper concentrates on speedups obtainable through parallelism alone, not increases in the speed of the underlying hardware. In contrast to all previous work, our approach is based upon the observation that annealing is not a static process. Partitioning strategies can be designed to exploit the dynamic behavior of simulated annealing algorithms.

3. Multiprocessor Decompositions of the Placement Problem

The success of a parallel algorithm depends largely on how it is partitioned across processors. This section describes a taxonomy of strategies to partition annealing-based placement tasks for shared-memory machines.

As an idealized model of sequential annealing, we assume that the time to propose, evaluate, and accept/reject each move is roughly constant. Then as shown in Figure 2a, the annealing task is characterized as a long sequence of accept/reject decisions on moves; the length of the sequence, and the per-move computing time determine the total solution time. Seen this way, we can conceive of two orthogonal approaches to accelerating annealing algorithms. The first approach is simply to reduce the per-move computing time (Fig. 2b) by dividing the task of computing a single move into several cooperating subtasks that execute in parallel on a multiprocessor. The second approach is to compute several complete moves in parallel (Fig. 2c). Annealing is still characterized as a long sequence of accept/reject decisions, but in each time step we evaluate many moves in parallel. We refer to these two basic partitioning schemes as *move decomposition* (Fig. 2b) and *parallel moves* (Fig. 2c). Figure 3 shows a taxonomy of strategies derivable from these two approaches. The remainder of this section examines the variations in parallel placement algorithms motivated from this starting point.

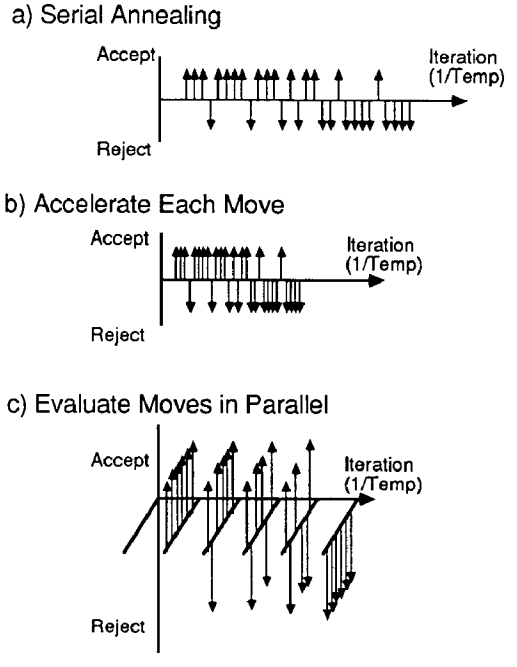

Figure 2: Approaches to Parallel Annealing

Note that move decomposition and parallel moves strategies are not mutually exclusive. With sufficient parallel resources, one could evaluate several moves in parallel and also divide each of these parallel moves into cooperating subtasks. Since these two approaches can appear in combination, we define two parameters to characterize a partitioning strategy:

- *Move-Granularity:* The number of parallel subtasks into which the evaluation of individual moves is divided.
- *Move-Parallelism:* The number of complete moves evaluated in parallel.

Different parallel placement algorithms will have different move-granularity and move-parallelism, but their product is bounded by the number of parallel processors required to execute the algorithm.

3.1 Move Decomposition Strategies

The work of a move consists of selecting a feasible perturbation in the move-set, evaluating the cost change (in this case wire lengths and cell overlaps), deciding to accept or reject, and perhaps updating a global database. Some, but not all these activities can proceed in parallel. Parallelism is strictly limited here: when moving an average module, we perturb only a limited number of connected cells and nets. Maximum move-granularity is roughly proportional to the size and terminal count of the cells being placed.

In Figure 3 we distinguish two types of move decomposition: decomposition by *object*, and by *function*. An object is typically a data structure; a decomposition by object delegates responsibility for a particular set of objects to a processor. A function is an operation on a data structure; a decomposition by function delegates responsibility for some set of operations to a particular processor. As an example, [7] suggests dividing modules and nets into groups and assigning them to individual processors. All necessary computations involving those objects are performed by the processors containing them: this is an object decomposition. If on the other hand we store all information in shared memory, and dedicate processors to individual tasks such as wire length evaluation after a move, the strategy is a functional decomposition. Note that the line between object and functional decompositions is fine, and should in practice be considered a matter of degree.

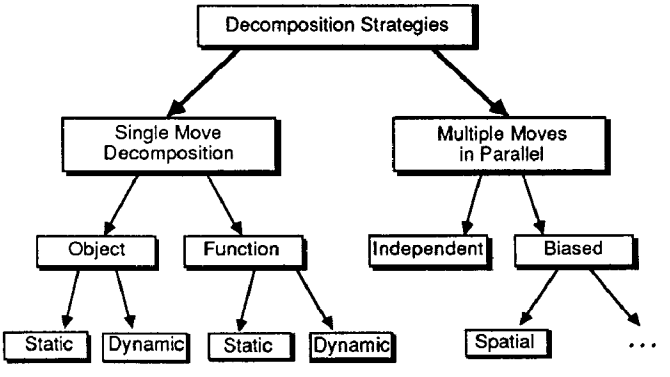

Figure 3: Taxonomy of Decompositions

As a further clarification, we further divide object and function decompositions into *static* and *dynamic* cases. If the distribution of objects to processors occurs at initialization and never changes, the strategy is *static*. If processors are still responsible for particular objects, but the objects can be reassigned during execution, the strategy is *dynamic*. Static and dynamic functional schemes are similarly defined.

Object and function decompositions have the advantage of small shared memory requirements, since most of the problem state may reside in the private memory of each processor. The disadvantage here is that as move-granularity increases, synchronization overhead also increases, reducing the expected speedups from greater parallelism.

3.2 Parallel Move Evaluation

Parallel moves decompositions are characterized by how a processor chooses moves to evaluate. Each processor could choose moves *independently* from the entire set of available moves. Alternatively, we could bias each processor's move choices. An example of biased move selection would be a spatial decomposition where we partition the chip *spatially* into regions, and assign a processor to perform moves within each region. In either case, the major problem is that moves interact. The interaction of concurrently evaluated moves limits the effective parallelism of the calculation.

When moves are evaluated in parallel it is important to control how moves that have been accepted by the normal annealing criterion are accepted and applied to the problem database. At the very minimum, moves which are accepted in parallel cannot be contradictory (e.g., moving the same cell to two different locations). Furthermore, erroneous accept/reject decisions are possible when moves are evaluated concurrently.

Suppose a set S of moves is evaluated in parallel. One way to avoid erroneous decisions is to restrict the set of moves that are accepted in parallel to some set of moves that do not interact with each other. Let S^{ni} be a set of non-interacting moves in S. Note that by concurrently applying all the moves in S^{ni} to some initial problem configuration, we reach the same configuration as if we serially applied the moves in S^{ni}, *in any order*. The moves in S^{ni} are *serializable*. Thus our initial requirement of non-interacting moves can be restated as a requirement that any set of accept/reject decisions made in parallel can be arrived at by some *serialization* of these moves. By maintaining the serializability of the processed moves, we guarantee that all of the statistical properties of simulated annealing are maintained. An option not discussed here is the deliberate introduction of uncertainty by accepting non-serializable sequences of moves. It is unclear what effect this will have on either the quality of the solution or the rate of convergence.

This notion of serialization is central to the parallel moves scheme. Annealing proceeds at each successive fixed temperature until thermal equilibrium is attained, usually determined in practice by evaluating some large fixed number of moves. If moves are evaluated in parallel, and some moves are accepted, some rejected, and some simply ignored, the question is how to count these moves toward the goal of reaching thermal equilibrium. Our solution is to find some serializable subset (an ordered subset of moves which, if evaluated serially, would produce the same accept and reject decisions), S^{ss}, of the parallel moves, S, and only count those toward equilibrium. We identify two extremes among approaches to finding good serializable subsets:

- **Optimum Serializable Subset:** Trying to find the largest serializable set of moves appears to be very difficult, at least as difficult as evaluating the moves in the first place.
- **Simplest Serializable Subset:** Note that the sequence formed by taking all rejected moves, and appending one accepted move, is always serializable. This scheme takes the accepted move found first during the parallel evaluation of moves, and aborts all other accepted moves.

A simple model illustrates the performance of the simplest serializable scheme. Assume that the evaluation of moves is synchronous: all processes simultaneously evaluate one move each time step. Let N be the number of processors and m be the number of moves accepted during a particular time step. At a temperature T, the probability of accepting a move is given by $a(T)$ which is a characteristic of the annealing algorithm. Since each processor either accepts a move with a probability of $a(T)$ or rejects a move with probability $1 - a(T)$, m is a random variable with a binomial probability distribution.

To find the simple serializable subset, S^{ss}, at each time step we will abort all but one of the accepts found. If there are no accepted moves ($m = 0$), then we will count N moves as progress towards equilibrium. If $m > 0$, then we will count the $N - m$ rejected moves and the one accepted move as progressing toward equilibrium. The size of S^{ss} is another random variable; its expected value, $E[|S^{ss}|]$ is the average number of parallel moves counted toward equilibrium, which is approximately the speedup expected from an algorithm implementing this policy.

This policy depends on $a(T)$, and should work well when $a(T)$ is small, in the cold regime. Note that for $a(T)$ small, $E[|S^{ss}|]$ is roughly N: few moves are accepted, so most move evaluations are counted toward equilibrium. For $a(T)$ large, $E[|S^{ss}|]$ is roughly 1: in the hot regime all processes try to accept a move, but because of serializability, only one succeeds, and we progress only one move toward equilibrium. Figure 4 confirms these observations, and shows predicted speedups for this idealized model using the data for $a(T)$ from figure 1.

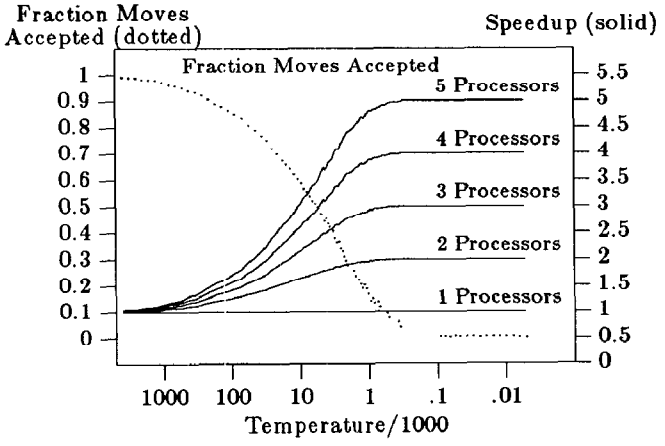

Figure 4: Performance Model for Simple Parallel Moves

3.3 Annealing is not Static: The Effects of Temperature

The parallel moves strategy is dependent on temperature; models suggest it performs best in the cold regime. The move-decomposition schemes are temperature independent; the number of parallel tasks into which a move can be divided (move-granularity) is roughly constant across temperature. The fact that the speedup of some algorithms varies as a function of temperature suggests that we need not employ just a single algorithm for placement, but should instead choose the best algorithm for each temperature regime. We call such approaches *adaptive* strategies.

Adaptive strategies can be viewed as temperature-varying tradeoffs between move-granularity and move-parallelism. In practice, the points at which we change strategies need not be determined with great precision, and can be found by tracking the fraction of accepted moves as annealing proceeds [5].

4. Implementation and Performance of Parallel Placement

Three strategies described in the previous section have been implemented and tested on an experimental multiprocessor. Reflecting the strategies employed, these implementations are called *DynamicFunction*, *StaticFunction*, and *ParallelMoves*. This section describes the parallel computing environment, the three implementations, and their measured performance.

4.1 Parallel Computing Environment

Placement experiments were performed on a DEC VAX 11/784, running an experimental version of the *Mach* operating system [1]. The 11/784 consists of four VAX-11/780 processors connected to a shared 8 Mbyte memory. *Mach* is a multiprocessor operating system being developed at Carnegie-Mellon for ultimate use on a larger multiprocessor system having on the order of 50 processors. All software developed as part of this work will port transparently to a larger system when it becomes available.

Perhaps the most important feature of *Mach* for this research is that it provides for the direct sharing of memory among multiple processes. A portion of physical memory is designated as shareable. Processes can allocate memory objects in shared memory and share them with other processes. Conventional synchronization primitives (e.g., semaphores), and interprocess communication (IPC) facilities provide for synchronizing communicating processes.

4.2 Common Implementation Characteristics

Two important characteristics of our parallel placement implementations are the synchronization mechanisms employed, and the allocation of critical data structures, both public and private, to shared memory.

Mach provides synchronization mechanisms with a wide range of speed and features. However, many of the events on which we synchronize are of such short duration that the overhead of having the operating system mediate the synchronization is unacceptably high. Therefore, our current implementations use simple spin locks relying on atomic VAX instructions and shared memory variables. As long as the critical regions over which we are performing exclusion are assuredly short, this is not excessively inefficient.

The data structures for the parallel implementations are very similar to *SerialPlace*. Some modifications have been made to remove some bottlenecks to parallelism. For example, in *ParallelMoves*, all temporary variables required to do one move are duplicated in each parallel process. This eliminates some of the need for mutual exclusion to maintain the integrity of these variables when several processes are working within the same parts of the global database. In the *StaticFunction* and *DynamicFunction* implementations, summary data structures containing concise information about the interaction of cells, nets and terminals are computed during initialization and stored within the global database. This simplifies the dynamic assignment of work to processors. For *all* implementations, the entire database for the placement problem is maintained in shared memory.

4.3 Implementation Descriptions

The structure of the *DynamicFunction* implementation is shown in figure 5. Each move is dynamically divided into several (up to about 15) jobs that are inserted into a shared *work queue* structure; processors compete to pull jobs out of the queue and execute them. A subtask can only be executed if all dependencies have been satisfied. All processes wait while one process makes the decision about accepting or rejecting the move, after which the update is performed in parallel.

The division of labor in the *StaticFunction* implementation is shown in figure 6. Moves are statically divided into three parallel subtasks. The first process chooses the N+1st move, calculates some penalties which contribute to the cost function, then waits for the other two processes to complete their portion of the Δcost calculation. When evaluation of Δcost has been completed by all processes, Process 1 decides whether to accept or reject the move, and signals the other two processes of the decision. If the move is accepted, all three processes handle the update, then proceed to the next move. The choice of three processes for this decomposition is essentially arbitrary, decided mainly because of its straightforward implementation. Different static decompositions using more than three processors are possible.

In the *ParallelMoves* implementation, processors concurrently propose and evaluate complete moves. In contrast to the simplified model for expected speedup presented in section 3.2, processors actually evaluate moves asynchronously, and

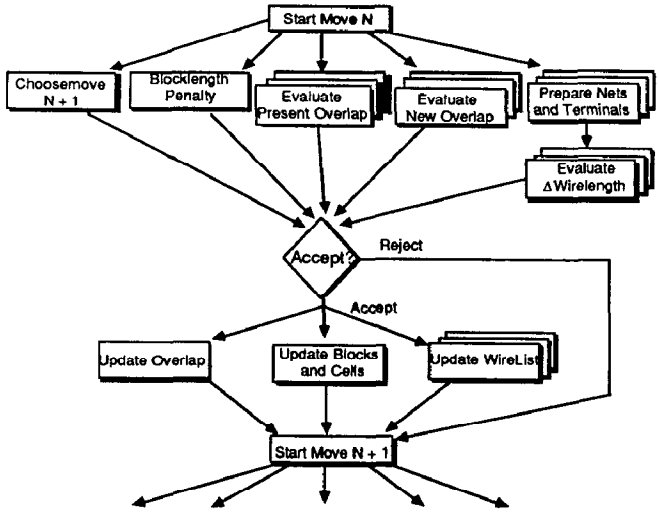

Figure 5: *DynamicFunction* Implementation

synchronize only when an accepted move is found. In practice, this requires several locking operations to maintain the integrity of the data structures. We implement the simplest serializable policy for evaluating progress toward equilibrium at each temperature.

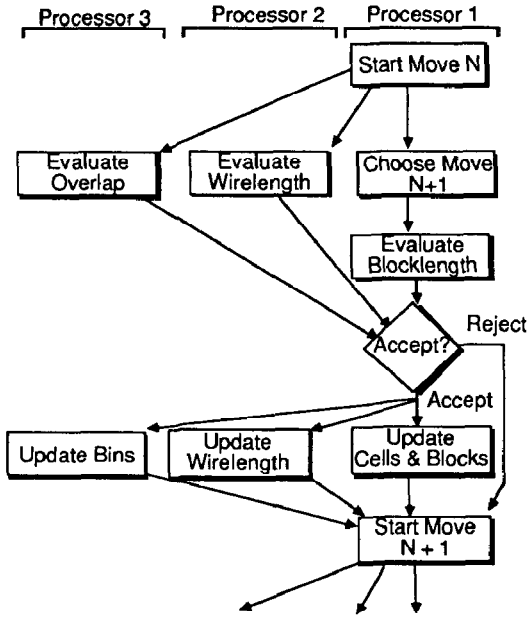

Figure 6: *StaticFunction* Implementation

4.4 Performance Results

Performance results for these three algorithms are quite promising. All parallel algorithms converge to solutions comparable in quality to those of *SerialPlace*. We measure at each temperature during annealing the time to reach thermal equilibrium, i.e., the time to perform some constant number of moves. We refer to the process of reaching equilibrium as an *iteration*. Data is displayed as time/iteration versus temperature. Time is measured as elapsed real time.

Performance data for the *DynamicFunction* code running on 1-4 processors is shown in figure 7. Although significant speedups over the single processor time are achieved for 2, 3, and 4 processors, the algorithm saturates at about 4 processors and is never much faster than *SerialPlace*. Some closer analysis reveals that this implementation spends considerable time simply waiting to satisfy the task dependencies shown in Figure 5. In addition, we have observed what amounts to enforced serialization in accessing jobs in the shared work queue. Mutual exclusion, necessary to maintain the integrity of the queue, permits only one process to access the queue at a time. There are several feasible solutions to this problem. For example, since the work to compute a move is completely determined when evaluation of the move begins, processes could simply remove multiple jobs from the queue in each access to reduce this conflict.

Results for the 3 processor *StaticFunction* implementation versus *SerialPlace* running on a single processor are shown in figure 8. The *StaticFunction* scheme yields about a 2x speedup over the serial algorithm, and further acceleration of this algorithm is possible. Note that *DynamicFunction* divides a move into about 10 pieces of work, but fails to provide much speedup because of the overhead of dynamically distributing the work. A static scheme can exploit just as much parallelism while avoiding this overhead. We conjec-

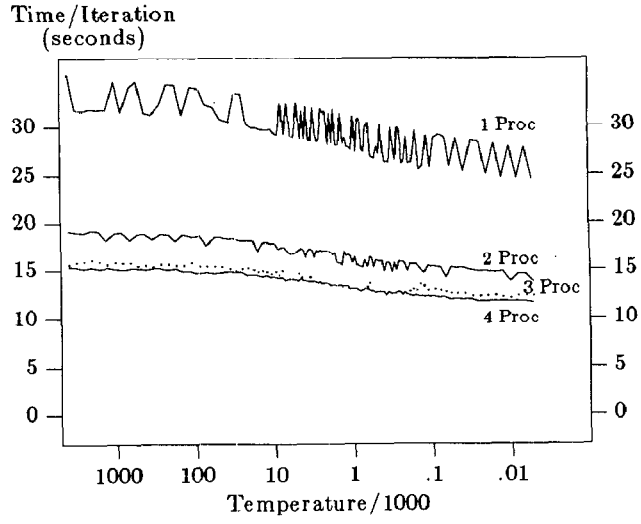

Figure 7: Performance of *DynamicFunction*

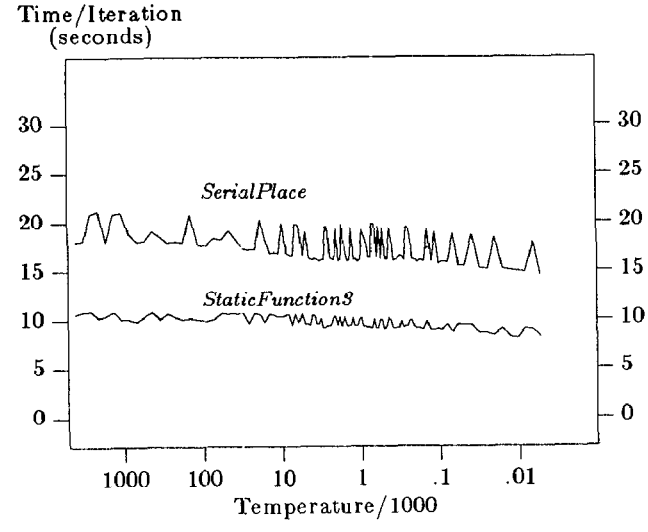

Figure 8: Performance of *StaticFunction*

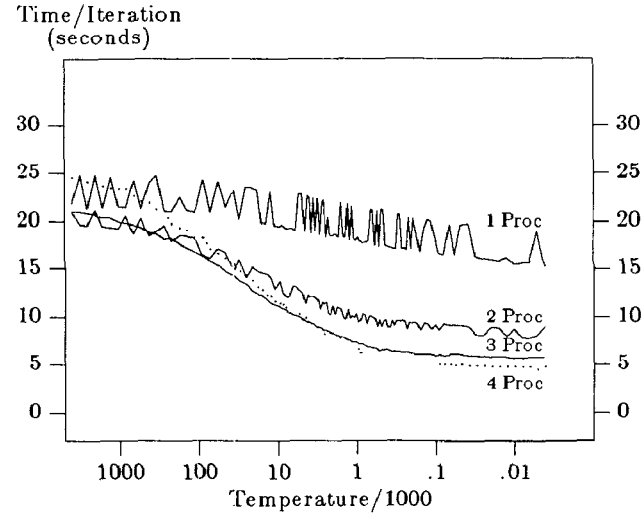

Figure 9: Performance of *ParallelMoves*

ture that this scheme will exhibit speedups scaling up to not more than 10 processors.

Performance of the *ParallelMoves* code running on 1-4 processors is shown in figure 9. Dramatic speedups occur, but only in the cold regime. It is interesting to note that in the hot regime, speedups actually decrease with the addition of processors. Since the *ParallelMoves* strategy requires synchronization between processes following every move accept decision, performance is affected in the hot regime where almost every move is accepted. Although hot regime performance can be improved with some low-level tuning, it is unclear if this is worthwhile to pursue, since in this case we are mainly interested in *ParallelMoves* in the cold regime.

Performance of the two best implementations expressed as speedup relative to *SerialPlace* is shown in figure 10. This data confirms our earlier conjecture about the utility of employing adaptive strategies: *StaticFunction* is clearly superior in the hot regime, and *ParallelMoves* superior in the cold regime. Switching from one strategy to the other midway through placement will produce the greatest speedup.

Speedup measured over the course of a complete annealing run will be deceptive, since all of the dependence of performance on temperature is masked. An unimpressive speedup curve does not necessarily imply that an algorithm attains

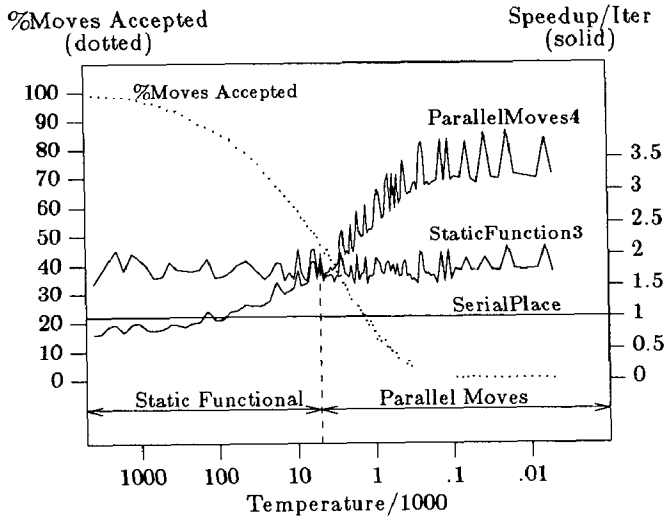

Figure 10: Comparative Speedup versus Temperature

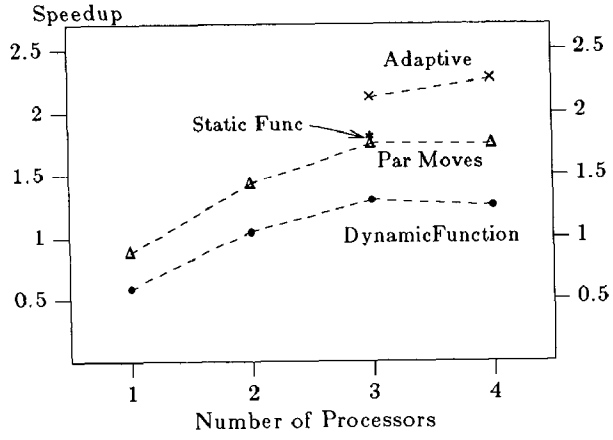

Figure 11: Speedup for Single and Adaptive Strategies

unimpressive speedups at all temperatures. In multiprocessor studies it is common to measure speedup of a given algorithm as a function of the number of processors. These traditional speedup curves for all implementations are shown in figure 11. A new artificial strategy *IdealAdaptive* is shown which switches from *StaticFunction* on 3 processors to *ParallelMoves* on 4 processors at the optimal temperature. Note that *IdealAdaptive* is significantly better than any single-strategy approach, and shows signs of continued increase in speedup as more processors are added. In practice, choosing the switching temperature will be based on runtime observations of the fraction of accepted moves, and need not be very precise.

5. Conclusions and Future Work

This paper presents new insights into the dynamics of annealing algorithms for standard cell placement, and demonstrates their impact on strategies for implementing placement on shared memory multiprocessors. We introduce the notion of adaptive strategies to produce optimal speedups over different temperature regimes. Implementations of three strategies on an experimental multiprocessor show practical speedups in a manner consistent with our observations of how annealing tasks change as they proceed. All these parallel algorithms will extend gracefully to larger multiprocessors, where they promise to provide even greater speedups.

Based on these experiments, we are currently refining our placement algorithms to remove the few remaining bottlenecks to parallelism. The success of these studies encourages us to develop other annealing-based parallel layout tools. A multiprocessor-based floorplanner is currently under development. Finally, we note that many of the methods presented here apply not just to placement, but should provide a basis for faster parallel implementation of a variety of related annealing-based tasks.

Acknowledgements

We acknowledge the helpful support of Zary Segall, Bob Baron, and Richard Rashid of the CMU CS Department.

Bibliography

[1] R. Baron *et al.*, "MACH-1: An Operating System Environment for Large-Scale Multiprocessor Applications", *IEEE Software*, July 1985.

[2] D. Chyan, and M. Breuer, "A Placement Algorithm for Array Processors", *Proc. of the DA Conf*, IEEE, 1983.

[3] A. Iosupovici, C. King, and M. Breuer, "A Module Interchange Placement Machine", *Proc. of the DA Conf*, IEEE, 1983.

[4] S. Kirkpatrick, C.D. Gelatt, M.P. Vecchi, "Optimization by Simulated Annealing", *Science*, 220(4598):671-680 , May 1983.

[5] S. Kravitz, "Multiprocessor-Based Placement by Simulated Annealing", Research Report CMUCAD-86-6, Dept. of ECE, Carnegie-Mellon University, February 1986.

[6] C. Sechen, A. Sangiovanni Vincentelli, "The Timberwolf Placement and Routing Package", *IEEE Journal of Solid-State Circuits*, SC-20(2):510-522 , April 1985.

[7] R. Smith, "Accelerator Plans for Iterative Improvement Placement", Presentation at IEEE Physical Design Workshop, January 1985.

[8] P. Spira and C. Hage, "Hardware Acceleration of Gate Array Layout", *Proc. of the DA Conf*, IEEE, June 1985.

[9] K. Ueda, T. Komatsubara, and T. Hosaka, "A Parallel Processing Approach for Logic Module Placement", *IEEE Trans. on CAD*, CAD-2(1):39-47 , January 1983.