Development of a WaveScalar Performance Model

Martha Mercaldi
mercaldi@cs.washington.edu
University of Washington

Abstract:

Faced with the challenge of effectively translating an exponentially growing number of transistors into improved performance, architects have begin to explore partitioned and distributed architectures. Such designs, called distributed ILP, require that a program be correspondingly partitioned to execute on sucn an architecture. The quality of this partition is important as it greatly influences the ultimate performance of the system. The search space of possible partitions is exponentially large, necessitating a heuristic search. In practice, simulation is much too slow to use as a guide for the heuristic, and an analytical model is more appropriate.

In this paper we develop a framework for constructing models of distributed ILP systems and present an analytical model for one such system: WaveScalar. The model has four primary components: latency, data migration, bandwidth and execution resource contention. We begin by modelling each component individually to determine the relative influence of each factor on the system and use the resulting information to construct a single, holistic view of system performance. The predictions of the resulting model show a -.85 correlation with actual performance. Such accuracy makes the model a fitting guide in a heuristic search for the best application partition.

1 Introduction

Moore's Law, which states that the number of transistors available to computer architects doubles every eighteen months, has held for the past decade, and is predicted to remain accurate for at least the next decade. However, system performance will face new challenges keeping up with this blistering pace. The challenge architects face is managing an ever-growing set of resources and effectively translating them into improved performance. Studies have shown that there is more instruction-level parallelism (ILP) available in programs [1] than processors have yet managed to exploit.

Traditional architectures rely on several centralized structures, such as the instruction queue, inter-functional unit bypass logic, and load-store queue, which limit their ability to scale. The size of such structures grows exponentially number of underlying resources, or functional units, they serve. This prevents processors from simply adding more resources to further exploit the available application parallelism.

Distributed ILP systems are one response to this challenge. They achieve scalability by partitioning centralized structures, or eliminating them entirely where possible thus avoiding their exponential growth. Several recently proposed systems, including nanoFabrics [2], TRIPS [3], RAW [4], SmartMemories [5], and WaveScalar [6] belong to this class.

However, in order to execute an application on such a partitioned and distributed architecture, the application must be correspondingly partitioned and distributed. As there are an exponential number of ways to partition an application, exhaustively exploring them all becomes infeasible. Thus, we must rely on heuristics to selectively explore the search space. This raises the question: With what do we guide our search? Our aim is to find the application partitioning that performs best, so the ideal approach would be to simulate each partition we want explore and to to measure how it performs. Unfortunately, simulation is much too slow to make this a viable possibility. A more practical approach is to develop an analytical model of the system that accurately predicts performance but is simple enough to be a suitable guide for our search. The development of such a model is the contribution of this work.

Here we develop a model of ne specific distributed ILP architecture, WaveScalar, which is described in more detail in Section 2. While this particular model was developed for one system, we believe the framework is generalizable to other distributed ILP architectures and can be applied with the same degree of success. Because WaveScalar shares the core features of any distributed ILP architecture, we expect any architecture-specific differences between models based on this framework to be superficial only.

Our model predicts performance of the WaveScalar system quite accurately, with predicted and actual performance correlating with a coefficient of -.85. Given this high correlation, the model provides a quick and accurate indicator of ultimate performance. It can thus be used to guide optimization of the application placement. Here we present, in detail, the model's development and evaluation.

The remainder of the paper is organized as follows: Section 2 describes the salient details of WaveScalar. Section 3 presents the methodology used to develop the model. Sections 5 through 7 detail the development and evaluation of the model. In Section 4 we clarify our relation to other similar research projects and conclude in Section 8.

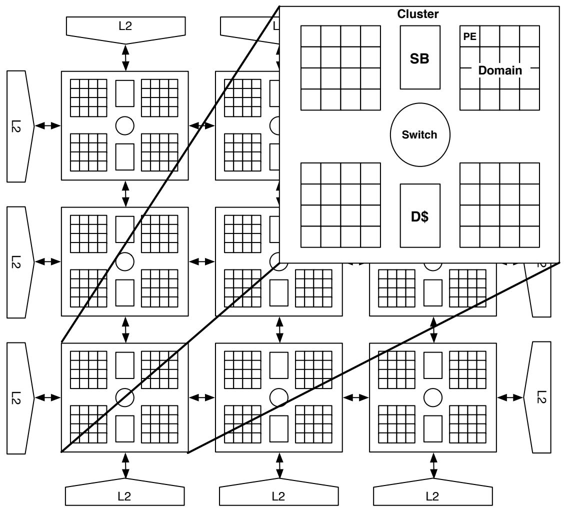


Figure 1: **The WaveCache and cluster:** A 3x3 WaveCache with nine clusters, and details of one cluster.

2 Background

2.1 WaveScalar Architecture

Before describing the details of the placement problem for WaveScalar [6] it is helpful to briefly present the details of the architecture. The reader is referred to Swanson, et al [6] for a more complete discussion of the processor and its dataflow execution model. In WaveScalar, as in all dataflow architectures (e.g. [7, 8, 9, 10, 11, 12, 13, 14]) the application is represented to the processor as a dataflow graph, with control dependencies translated into data dependencies. The binary is simply an encoding of this graph.

The architecture itself is called the WaveCache and is shown in Figure 1. As can be seen, the design is heirarchical and highly regular. At the lowest level of the heirarchy is the *Processing Element* (PE) which contains an ALU capable of executing any instruction in the instruction set. Program instructions are stored at the same PE in which they are executed. In addition, PEs are virtualizable, meaning that multiple instructions can be stored at a PE at one time. PEs are grouped into *domains*. The size of a domain is limited by the following restriction: all intra-domain communication must happen within a single cycle. Currently there are 16 PEs per domain. Four domains, together with a piece of the L1 data cache, a store buffer [1], and a network switch make up a *cluster*. A cluster is the smallest component that can execute a program. This design can be replicated as many times as space allows or as performance requires.

Instructions communicate explicitly via an interconnect network rather than via a register file as in traditional architectures. The network switches are used for both intra- and inter-cluster communication. Each network switch can communicate with its vertical and horizontal neighbors. As can be inferred from the preceding description and from Figure 1, the architecture's complete L1 data cache is distributed across all clusters. The L2 cache and coherence directory surround the computation elements.

For large applications, the WaveCache might not be large enough to hold all of the instructions at once. Instead, the grid acts as an instruction cache, caching the application's working set. Hence the name WaveCache.

[1]This component implements Wave-Ordered Memory, the mechanism which preserves total load-store ordering, allowing C-style imperative code to execute. See Swanson, et al [6] for details of this algorithm

2.2 WaveScalar Placement

As mentioned earlier, an application is represented as a graph in which the nodes represent instructions, and the edges indicate an operand dependency. In WaveScalar we refer to the task of partitioning the application *placement*. Placement is essentially a mapping of instructions onto PEs. There are two independant dimensions along which a mapping can be classified: creation time (either static or dynamic) and mutability (either mutable or immutable). Static mappings are created before a program is executed and unlike dynamic mappings do not consider any runtime properties of the application. Along the second axis, immutable mappings are mappings for which an instruction's assignment to a PE cannot be changed once it is initially set, and so an instruction will always execute in the same PE. Mutable mappings allow the mapping to change as the program executes and therefore allow instructions to migrate from PE to PE.

This work is applicable to both dynamic and static mappings. With respect to mutability we restrict ourselves to immutable mappings. Before addressing the more challenging task of evaluating mutable mappings one must first understand the simpler case of immutable mappings.[2]

Though simpler, discerning the best immutable mappings is no small task. For an application with n instructions, and a WaveCache with p processing elements, there are $O(n^p)$ possible mappings. This makes any sort of exhaustive search infeasible, therefore we need to use a heuristic search. The task of the heuristic is illustrated in Figure 2. Which is the best of mappings (a), (b), and (c) (or any of the other possibilities for that matter)? On the one hand instructions which communicate should be close, but not so densely packed that the local communication overwhelms the network. This is just one example of the often conflicting concerns, which must be balanced in the ideal placement. Simulation allows us to accurately evaluate a mapping but is unfortunately too slow to be practical. Thus we must turn to an analytical model of the system.

In the following section we present our framework for finding a model which balances these tradeoffs.

[2]While the WaveCache is multithreaded [15], here we use only single-threaded applications.

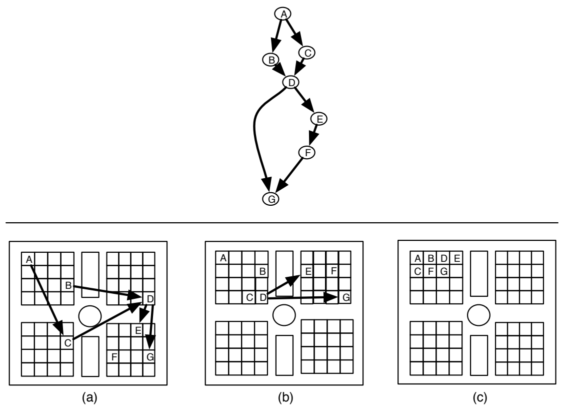


Figure 2: **WaveScalar Application Layout:** Several possible layouts of a sample application graph are proposed. Only intra-domain communication is indicated by arrows in the layouts. For a graph with n nodes and a WaveCache with p processing elements there are $O(n^p)$ possible layouts. The task of a placement heuristic is to distinguish those that will perform best.

2.3 Model Inputs

The model will use three pieces of input: a proposed assignment of instructions to PEs, an application profile, and the WaveCache parameters.

The application profile used here is a very simple frequency profile. Like the application itself, it is represented as a graph, shown in Figure 3. The nodes represent program instructions, and the directed edges that connect them indicate operand producer-consumer relationships. These edges are annotated with the number of times an operand was passed from producer to consumer during profiled execution. The profile includes a second type of node, an address node, each of which represents an address (of cache-line size granularity) in memory that was accessed by the application. Address nodes are connected to the instructions which access them by undirected edges. These edges also have weights indicating the number of profiled accesses.

The WaveCache parameters are used to tell the heuristic what hardware resources are available. Table 1 details the parameters used for experiments presented in this paper.

3 Methodology

We develop a heuristic to estimate the quality of a placement by first modeling separately the architectual components which come to bear on the quality of an instruction placement. The four components with which we begin are: operand *latency*, distributed *data* cache behavior, operand *bandwidth* demands, and *contention* amongst instructions for residence in the WaveCache. These components are described in detail in Section 5.

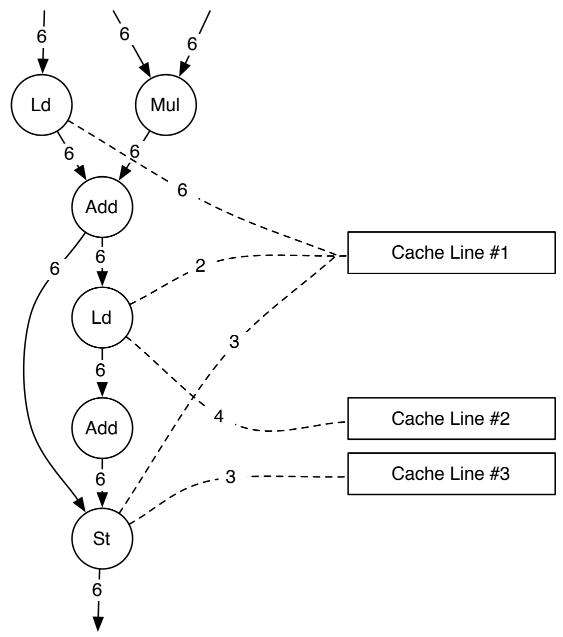


Figure 3: **WaveScalar Application Profile**

We evaluate each component model using a cycle-accurate, event-driven WaveScalar simulator. We use a set of six benchmarks: two from SPEC [16] Int (*gzip*,*mcf*), two from SPEC Float (*equake*, *art*), and two from Splash2 [17] (*fft*, *ocean*). For each benchmark, eight possible instruction mappings are produced. We used eight relatively simple algorithms to produce these mappings for each benchmark. The algorithms, described briefly now, were chosen as simple, easily implementable algorithms which exercise the four architectural components on which our model will be based. This selection of mappings produces a range of operand latencies, data cache demands, bandwidth demands and processing element contention.

random Each instruction is assigned at random to any instruction in the WaveCache.

packed-random Each instruction is assigned randomly to one of a restricted set of processing elements. The set of PEs to which instructions can be assigned is restricted to a physically contiguous region large enough to hold the complete program, but no larger.

static-stripe Each instruction is assigned in static program order to each processing element. Processing elements are filled to capacity first, then processed row by row. Domains are filled next, followed by the entire cluster, and finally spilling into the next cluster.

dynamic-stripe Processing elements are filled in the same order as **static-stripe** but instructions are assigned in dynamic program order.

WaveCache Capacity	$16,384$ static instructions (16 per PE)		
PEs per Domain	16	Domains per Cluster	4
PE Input Queue	8 entries, 2 ports (1r, 1w)	Network Latency	PE→Switch: 1 cycle
PE Output Queue	8 entries, 2 ports (1r, 1w)		Switch→Switch: 1 cycle
L1 Caches	16KB, 4-way set associative, 64B line, 4 accesses per cycle	L2 Cache	16 MB shared, 128B line, 4-way set associative, 20 cycle access
Main RAM	1000 cycle latency	Network Switch	4-port, bidirectional

Table 1: Microarchitectural parameters of a proposed WaveCache

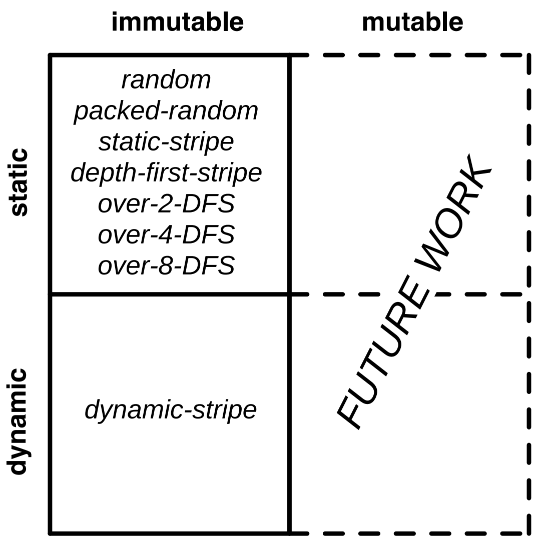


Figure 4: **Classification of Application Mappings:**

depth-first-stripe Processing elements are filled in the same order as **static-stripe** but instructions are assigned in the order they are encountered in depth-first pre-order traversal of the dataflow graph.

over-2-DFS Like **depth-first-stripe** except processing elements, instead of being filled to capacity, are filled to double capacity before moving on to the next one.

over-4-DFS Like **over-2-DFS** except PEs are filled to quadruple their capacity.

over-8-DFS Like **over-4-DFS** except PEs are filled to eight times their capacity.

In Figure 4 we classify these algorithms according to the classification laid out in Section 2. To evaluate a component model we ran each of these forty-eight (six benchmarks, eight mappings each) layouts through the model to obtain a prediction. We then simulated each layout on a specially configured simulator that isolated just the architectural feature we sought to model. For example, to measure only the effects of operand *latency*, we idealized the other three resources (*bandwidth*, *data* and *contention*) by providing infinite network ports, infinite store buffer ports and queue length, idealized data caches, and infinite instruction capacity at each processing element. With this simulator configuration the only thing that could limit performance was operand latency (and, of course, the application itself, but this can be safely ignored as it is not affected by the instruction placement). With this data, we can calculate the correlation between the predicted and measured performance.

These individual component models enable us to determine the relative influences of each component. Given the competing nature of these components (e.g. latency v. contention) these relative influences will allow us to assign proper weights to each component when we combine them into a unified model. To do this we quantify the average *contribution* of each component according to Equation 1. Intuitively, the larger the variance seen across a wide range of placements the greater the performance improvement that can be gained by properly optimizing a given component. We divide by the average over all placements in order to determine the relative magnitude of this potential improvement.

$$contribution_{component} = \frac{variance(IPC) \text{ of all placements}}{average(IPC) \text{ over all placements}} \tag{1}$$

4 Related Work

At first glance, the instruction placement problem appears similar to the existing and well-studied problem of place and route in FPGA and ASIC synthesis ([18],[19]). However, these problems fundamentally differ in the dynamicism the underlying computation substrates elicit during execution. The dynamic behavior of distributed ILP processors makes predicting, and hence optimizing, application performance far more challenging.

Another project that is similar in appearance is that in mapping a neural network onto multiprocessor computers [20]. Modeling of distributed ILP systems must take into account several other components and the resulting interrelationships that bear on ultimate performance.

Any software transformations or optimizations [21] that serve to increase or manage parallelism are complementary to this work. Such transformations serve to improve the potential performance of the application. The analytical model presented here is applied to the resulting application, and its aim is to enable the computing resources to realize as much of the software-offered parallelism as possible.

Also complementary is the recent work on on chip interconnect implementations, [22]. As with the software transformations, such hardware optimizations improvements can also help performance, but are orthogonal in purpose to an analytical model.

Performance models have proved extremely valuable in other contexts and have been pursued for many architectures. Many are based on statistical simulation [23], [24], [25], program traces [26], or both [27]. However these focused on superscalars, which have a dramatically different set of performance considerations from a distributed ILP architecture. Looking for models closer to this paradigm, one finds many for parallel programs [28], parallel sequential machines [29], and multithreaded machines [30], but these still do not address distributed ILP.

Finally the existence of several ongoing research efforts to design distributed ILP architectures illustrates the need for an effective method of placing instructions. In addition to WaveScalar these efforts include nanoFabrics [2], TRIPS [3], RAW [4], and SmartMemories [5]. Since these architectures all share the same core features, the performance model framework and methodology presented hear should apply to those systems as well.

5 Components of Performance

The model comprises four component models, each one designed to model one factor that bears on the ultimate quality of an instruction mapping. Here we present these four models. They have been divided into two classes, features which favor densely packed mappings, and those that favor more evenly dispersed mappings. The goal of the ultimate model is to find a suitable tradeoff between these competing concerns.

Before beginning, we define the variables and terms which are used in these models:

n is the number of instructions in the application.

i refers to the i'th instruction, j refers to the j'th instruction.

C_i is the x, y physical location in the WaveCache of the cluster containing instruction i. Cx_i, and Cy_i are the components individually. Similarly, D_i and P_i are the location of the domain and processing element, respectively, where instruction i is placed.

$S_{(x,y)}$ is the network switch in the x, y cluster.

$T_{i,j}$ refers the amount of communication (or traffic) between instructions i and j.

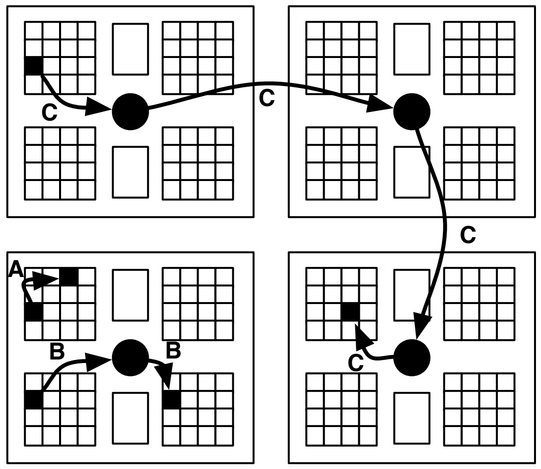


Figure 5: **Operand Latency:** We expect operand communication to have varying latencies depending on how instructions are placed.

A is the set of cache-line sized addresses the application uses.

a refers to a particular cache-line size address.

$M_{i,a}$ refers to the number of memory accesses between instruction i and memory address a.

$M_{C,a}$ refers to the number of memory accesses between all instructions in cluster C and memory address a. $M_{C,a}$ can be defined in terms of $M_{i,a}$: $M_{C,a} = \sum_{i \in C} M_{i,a}$.

Several times we refer to the distance between two items in the WaveCache. We use the Manhattan-distance in terms of clusters, domains or processing elements. This distance, $||C_i - C_j||$ is simply calculated as $|Cx_i - Cx_j| + |Cy_i - Cy_j|$.

5.1 Latency

With shrinking process sizes, on-chip communication is expected to increasingly dominate performance [31]. One obvious determinant of performance in spatial architectures is how much of this communication is local versus long-distance. Given a pair of producer and consumer instructions, it would be much better to place them nearby each other instead of on opposite sides of the WaveCache, particularly if they execute frequently. If not, all the time spent waiting for the operands to traverse the WaveCache will serve only to increase execution time.

The operand latency model is a simple weighted sum of all operand communication latencies. An operand is indicated by an edge in the application's dataflow graph. Figure 5 illustrates three different classes of communication in the hierarchical communication model the WaveCache uses.

All intra-domain communication, labeled 'A' in Figure 5, requires only a constant single cycle. As this is the lowest possible communication latency it is considered "free" and contributes nothing to the final calculation (shown in Equation 2). If a pair of producer-consumer instructions are not in the same domain ('B' and 'C' in Figure 5), the latency estimate then depends on the distance between their respective clusters. The latency is calculated as shown in Equation 2. The extra two cycles are added to account for the operand traveling from its source domain to the local cluster's network switch, and from the destination cluster's network switch to the destination cluster.

$$Latency_{i,j} = \begin{cases} 0 & \text{if } D_i = D_j, \\ T_{i,j} \cdot (||C_i - C_j|| + 2) & \text{otherwise.} \end{cases} \tag{2}$$

The total latency estimate for a mapping is the summation of all of the operand latencies, each weighted by the expected volume of traffic that travels that route. This frequency comes directly from the application profile. Equation 3 shows the full calculation.

$$Latency = \sum_i \sum_j (T_{i,j} \times Latency_{i,j}) \tag{3}$$

For each of the forty-eight mappings described in Section 3 we calculated this latency value. We then simulated each layout with the simulator configured such that only operand latency would limit performance (with infinite bandwidth, idealized data caches, and infinite WaveCache capacity). The graph in Figure 6 shows the correlation between these two values (latency metric and normalized IPC) for one benchmark, *fft*. Each point in the graph is one of the eight instruction mappings detailed in Section 3.

The first column of the chart in Figure 6 gives the coefficient of correlation for each benchmark. A correlation coefficient of 0 means there is no correlation between the two data values, while 1 or -1 indicates perfect, linear correlation and inverse correlation respectively. As we can see, in this set of benchmarks the latency model nearly perfectly predicts the measured performance limited only by latency. We also see an average contribution of.62 which indicates that by we should be able to improve latency by roughly 62% via careful placement.

5.2 Distributed Data Cache Behavior

In addition to operands, instructions share data indirectly through memory. Two instructions might have a producer-consumer relationship in which the producer writes a value to a cache line, which is subsequently read by the consumer. It is best to place such pairs of instructions in the same cluster. The design of the cache-heirarchy is the reason for this. As described in Section 2, the L1 cache is distributed across the WaveCache with one portion in each cluster. When an instruction accesses memory it first checks the local L1 cache of the cluster in which it is resident. If two instructions in the same cluster are accessing the cache line it will be there locally available to each of them. However if those two instructions are not in the same cluster, for one of the instructions it is considered an L1 cache miss. Because only one copy of the cache line is maintained in the L1 cache, it must be copied back and forth between any clusters from which it is accessed.

Using the profile data we can calculate directly the number of times each address is accessed from each cluster, for each proposed instruction placement. What we cannot tell from the profile information is how these accesses are distributed over time. For example, suppose two instructions i_A and i_B, resident in clusters C_A and C_B, were sharing a cache line a. If i_A completed all of its accesses before i_B began its accesses, there would be only two L1 misses: a compulsory miss when the line is first loaded from the L2 cache into C_A's L1 cache, and a sharing miss when i_B first accesses a and the line is copied from C_A to C_B. This access pattern is illustrated in Figure 7.

At the opposite extreme is the case in which i_A and i_B's accesses are interleaved over time: i_A accesses it first from C_A, then i_B from C_B, then i_A from C_A, and so on. In this case all of the accesses would cause an L1 miss: first the compulsory miss followed by a sequence of sharing misses. This is illustrated in Figure 8.

For each of these extreme cases we can construct a simple model which calculates the number of L1 cache hits and misses for a cache line a. Equations 4 and 5 calculate the number of estimated misses and hits respectively under the *minimal-sharing-migration* model. Equations 6 and 7 show the calculations for the *maximal-sharing-contention* case.

Minimal Sharing Migration

$$Misses_a = \sum_C \begin{cases} 1 & \text{if } M_{C,a} > 0, \\ 0 & otherwise. \end{cases} \tag{4}$$

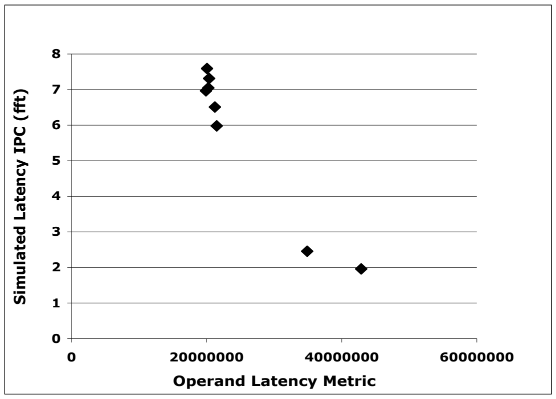
$$Hits_a = \left(\sum_C M_{C,a} \right) - Misses_a \tag{5}$$

Maximal Sharing Contention

$$Misses_a = \left(\sum_C M_{C,a} \right) - Hits_a \tag{6}$$

$$Hits_a = Max_C\{M_{C,a}\} - SecondMax_C\{M_{C,a}\} \tag{7}$$

Using the totals shown in Equations 9 and 8 we can calculate an expected miss rate from each model.



Benchmark	Correlation Coefficient	Contribution
fft	-0.97	0.87
ocean	-0.88	0.78
equake	-0.99	0.25
art	-0.97	0.71
gzip	-0.99	0.55
mcf	-0.98	0.56
Average	-0.96	0.62

Figure 6: **Operand Latency Metric Evaluation:** On the left is a graph depicting simulated application performance versus our latency metric for a single application, *fft*. Other applications are qualitatively similar. On the right is the correlation between simulated performance and the latency metric for our application suite. Since higher latency should lower IPC, a perfect correlation would be -1.

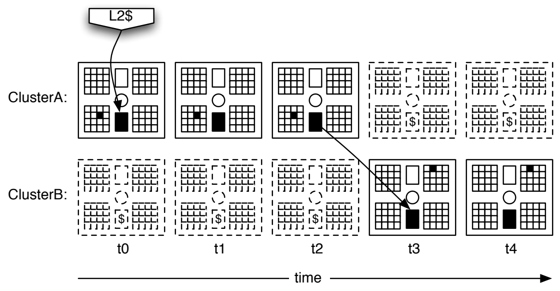


Figure 7: **Minimum Sharing Migration:** Distributed cache access pattern that minimizes L1 coherence misses.

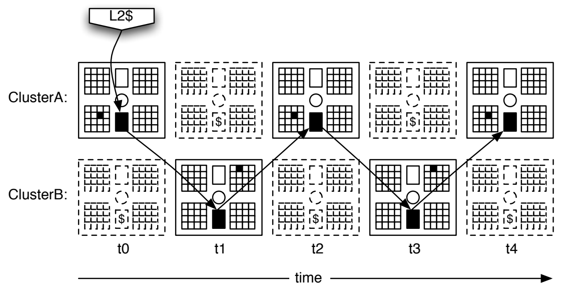


Figure 8: **Maximum Sharing Contention:** Distributed cache access pattern that maximizes L1 coherence misses.

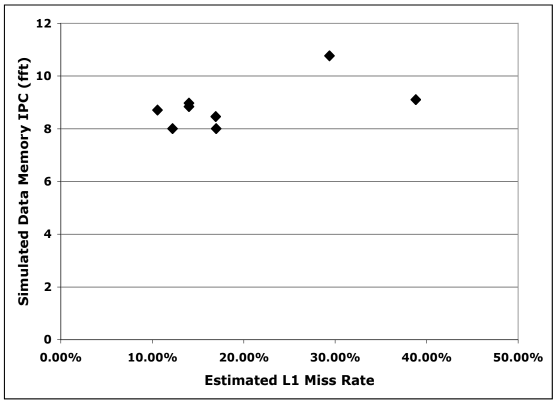


Figure 10: *fft* **Data Model Detail**

$$Misses = \sum_a Misses_a \tag{8}$$

$$Hits = \sum_a Hits_a \tag{9}$$

While data cache capacity does influence performance, we have not included it in the data cache component because variations in instruction placement do not affect the size of its influence.

Observations from simulations suggest that approximately 80% of misses are compulsory and set-conflicts, while the rest are sharing. Of the sharing misses less than 1% of cache-lines exhibited severe contention. This suggests the minimum-sharing-migration model will more accurately model performance. To test this hypothesis we simulated the WaveCache with no operand latency, bandwidth constraints, or execution resource constraints. Only the data caches throttle performance. The resulting application performance was correlated against the miss ratio ($Misses/(Misses+Hits)$) predicted by the model. The results are shown in the table in Figure 9.

Because a increased miss rate should yield poorer memory performance, an ideal correlation would be -1. Here we see reasonable correlation most benchmarks. Also note that the average contribution is significantly lower than that of latency (.08 versus .62). This means we can effect performance relatively less through the optimization of data cache behavior when compared to operand latency.

5.3 Operand Network Bandwidth

Network bandwidth is the first of the hardware concerns that would argue for more distrubuted instructions. While we would like most communication to be local, if all of it is restricted to the same physical section of the grid, all of the concentrated network traffic will overwhelm the available hardware and slow execution to a crawl. To build this concern into the heuristic, we develop a component which characterizes the expected bandwidth behavior of a proposed application layout. The WaveCache currently uses a dynamic routing algorithm by which a message is routed in the direction of its destination with the path chosen randomly from anomg the potential routes.

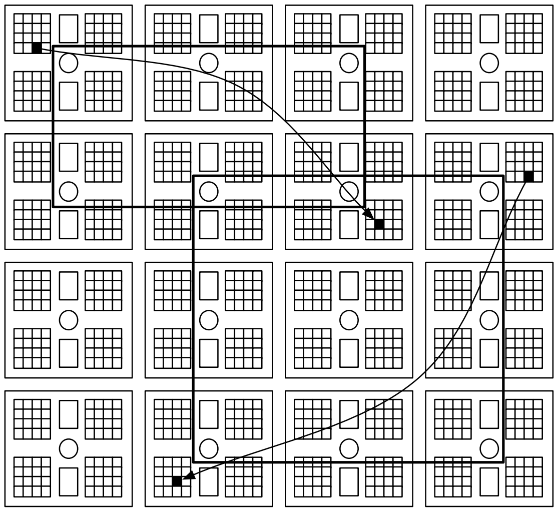
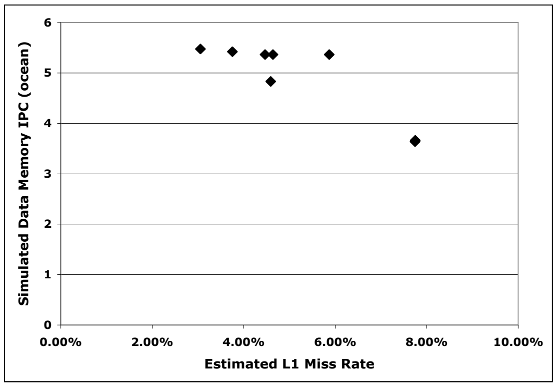


Figure 11: **Overlapping Operand Bandwidth Demands:** Each operand may require bandwidth from a subset of the network links.

We estimate the bandwidth demands at each network link in the following way: Given the random routing algorithm, when instruction i sends an operand to instruction j it will take one of many possible paths from the source cluster, C_i, to the destination cluster, C_j. All of the possible paths are contained within the rectangular region defined by C_i and C_j. We refer to this region, shown for two sample communication arcs in Figure 11, as $BoundingBox(C_i, C_j)$. Overlaying all of these rectangles for all operand communication in the application, we can estimate the total expected bandwidth requirements at each switch.

Across many messages, the aggregate bandwidth utilization across these links is a predictable function of the layout and profile data. The expected load at $S_{(x,y)}$ due to communication between instruction i and j, $Load_{x,y,i,j}$, is the sum of all incoming link loads.



Benchmark	Correlation Coefficient Minimum Sharing Migration	Correlation Coefficient Maximum Sharing Contention	Contribution
fft	0.56	-0.02	0.09
ocean	-0.89	-0.65	0.13
equake	-0.82	-0.77	0.02
art	-0.90	-0.88	0.12
gzip	-0.91	-0.71	0.10
mcf	-0.96	-0.42	0.02
Average	-0.65	-0.57	0.08

Figure 9: **Data Memory Metric Evaluation:** On the left is a graph depicting simulated memory system performance versus our data memory metric: expected miss rate. The graph shows the data points for each instruction layout for *ocean*. On the right is the correlation of data metric to simulated IPC for the other benchmarks.

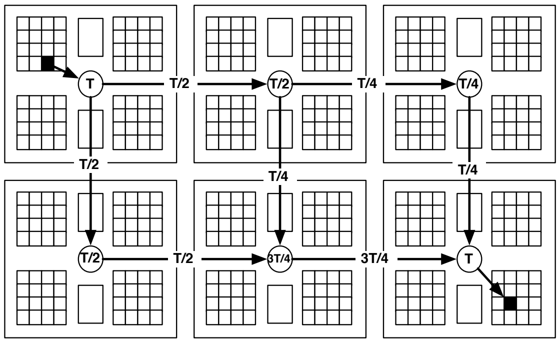


Figure 12: **Bandwidth Demand Estimation:** Estimated bandwidth demands made by a single operand which is sent C times.

$$Load_{x,y,i,j} = \begin{cases} 0 \\ \quad \text{when} \quad x,y \notin BoundingBox(C_i,C_j) \\ T_{i,j} \\ \quad \text{when} \quad x,y = C_i \vee x,y = C_j \\ Load_{x-1,y,i,j} + Load_{x,y-1,i,j}/2 \\ \quad \text{when} \quad x = Cx_j, y \neq Cy_j \\ Load_{x-1,y,i,j}/2 + Load_{x,y-1,i,j} \\ \quad \text{when} \quad x \neq Cx_j, y = Cy_j \\ Load_{x-1,y,i,j}/2 + Load_{x,y-1,i,j}/2 \\ \quad \text{when} \quad x \neq Cx_j, y \neq Cy_j \end{cases} \tag{10}$$

The details of the calculation are showin in Equation 10 and illustrated in Figure 12. We have simplified things slightly by assuming that C_i is the upper-left and C_j is the lower right corner of this box. A similar set of equations (not shown) are used to compute the load when this is not

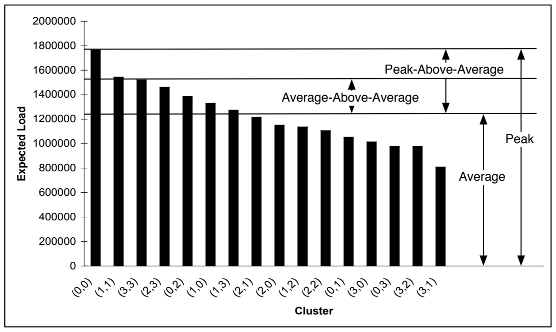


Figure 13: **Total Bandwidth Demands:** Total estimated bandwidth demands for each network link in Figure 11.

the case.

Using this model we can then estimate the total amount of network traffic that passes through any network switch $S_{(x,y)}$:

$$Load_{x,y} = \sum_{i,j} Load_{x,y,i,j}$$

Once we have an estimate of the total load for each network switch, we must find a meaningful way to summarize the information. Figure 13 illustrates the challenge. Here we show the estimated load for each network switch for the **depth-first-stripe** placement of *FFT*. It is not obvious from this raw information how the betwork will behave from a bandwidth perspective. We tested many four ways to interpret this graph. They are described here:

Average Average load of all network switches: $AverageLoad = (\sum_{x,y} Load_{x,y})/NumberSwitches$.

Peak Maximum load of all network switches: $\max_{x,y}(Load_{x,y})$.

Average-Above-Average The average amount by which the expected workloads exceed the average workload: $\sum_{x,y}(Load_{x,y} - AverageLoad)/NumberSwitches$ when $Load_{x,y} > AverageLoad$

Peak-Above-Average The maximum amount by which any one expected workload exceeds the average workload: $max_{x,y}(Load_{x,y}) - AverageLoad$

We computed each of these metrics for each layout, and correlated them against another specially configured WaveCache simulator. In this simulation, latency, data, and contention for execution resources are free, but contention on network switches is not. The results are shown in Table 2. We see that the correlations are not as strong as in the other models. Notice that as this component varies, the performance of the system is typically not affected, as evidenced by the fact of contributions of 0. The only time that the correlation is meaningful is when the contribution is non-zero as it is in *fft* where it is .13.

Thus, we base our decision on the only situation where changing the layout affects bandwidth. As with the other component models, the aim is to select a metric with the strongest negative correlation. In this respect the metric that predicts bandwidth performance best is Peak-Above-Average. Notice that the Average metric correlates strongly in the opposite of the desired direction. At first this appears to be an error, however, one can show through a derviation that $AverageLoad \approx k \times Latency$, where k is a constant! This is to say that Latency directly correlates with AverageLoad which in turn correlates inversely to Bandwidth performance. This exposes one of the tradeoffs mentioned previously and discussed further in Section 7. The Latency and Bandwidth models conflict in that a layout that is better with respect to Bandwidth is worse from the perspective of Latency and vice versa.

6 Processing Element Contention

The fourth and final component of the placement problem which we model is contention at the processing elements. The PEs in the WaveCache have a limit on the number of instructions they are able to hold. If the number of instructions assigned to a processing element exceeds this limit, the excess instructions must be stored in memory and swapped in when required by execution. PEs use an LRU algorithm to decide which instruction to replace. Like all cache misses, performing these swaps incurs a performance penalty. For the experiments presented in this paper PEs could hold up to 16 instructions at a time.

We use a simple model to estimate the amount of contention there will be among instructions for space in the WaveCache. The estimate is the number of instructions which cannot fit in the PEs to which they are assigned. In the following equations $PECapacity$ is the number of instructions which any PE can hold, and I_P is the set of instructions assigned to processing element P.

$$PeContention = \sum_P \begin{cases} |I_P| - PECapacity \\ \quad \text{when} \quad |I_P| > PECapacity \\ 0 \\ \quad \text{otherwise} \end{cases} \tag{11}$$

Despite this model's simplicity, it is highly effective at modeling grid space contention in the WaveCache. Figure 14 shows the experimental results. As with the other metrics, perfect correlation would be -1, as greater contention should equate to lower performance. The average correlation across all benchmarks and all layouts is $-.61$. The average contribution was 0.91, indicating that contention has a significant influence on performance.

Examining in more detail a graph of one of the more weakly correlating benchmarks *equake* (also in Figure 14), we see that by and large the contention metric correlates extremely well, but breaks down when the contention metric is equal to 0. Here we see that despite having no contention for space in the WaveCache, performance suffers. Recall that component IPC was measured with all other components of the simulator idealized (immediate operand delivery, infinite interconnect bandwidth resources, and perfect distributed cache coherence). However, if there was no contention for PEs, there must have been some remaining resource that restricted performance. One possibility that remains is contention for the ALU resources at the PE. Since the ALU can execute only one instruction per cycle, naive placements that do not attempt to exploit this are subject to performance degradation. This is the case with the *equake* benchmark and the STRIPE placement (the lowest point on the left of Figure 14).

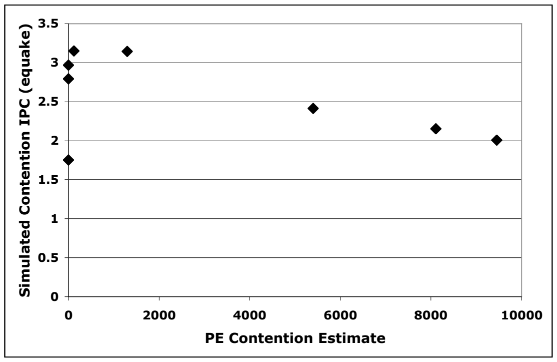
When we remove the zero-contending points, also shown in the table in Figure 14, the correlation increases dramatically. This, along with the high contribution of performance values suggest that tools should be careful to segment use of PEs across disparate temporal sections of application execution in order to minimize swapping at PEs.

7 Unifying the Model

In order to develop the richest, most complete model with which to guide an instruction placement optimization, we must weigh all four of thses models. However there is a fundamental tension in instruction placement, between sharing, which exhorts us to keep instructions close together, and finiteness of hardware resources, which encourages physical

	Correlation Coefficient				
Benchmark	**Average**	**Peak**	**Average-Above-Average**	**Peak-Above-Average**	**Component Contribution**
fft	0.68	-0.35	-0.37	-0.44	0.13
ocean	0.02	0.56	0.65	0.40	0.00
equake	-0.77	0.81	0.80	0.84	0.00
art	-0.21	-0.06	-0.02	0.00	0.01
gzip	-0.40	0.12	-0.12	0.32	0.00
mcf	-0.66	0.30	0.28	0.42	0.01
Average	-0.22	0.23	0.20	0.26	0.03

Table 2: **Bandwidth Metric Correlations:** Correlation coefficients of each of the four potential bandwidth metrics to IPC.



	Correlation Coefficient		
Benchmark	**All Layouts**	**Layouts with non-zero contention**	**Contribution**
fft	-0.67	-0.98	2.20
ocean	-0.65	-0.65	1.27
equake	-0.54	-0.99	0.12
art	-0.62	-0.56	0.81
gzip	-0.54	-0.56	0.64
mcf	-0.65	-1.00	0.43
Average	-0.61	-0.79	0.91

Figure 14: **Processing Element Contention Metric Evaluation:** The table on the right shows the correlation of our PE contention model to the simulated IPC. The graph shows the detailed data for one application: *equake*

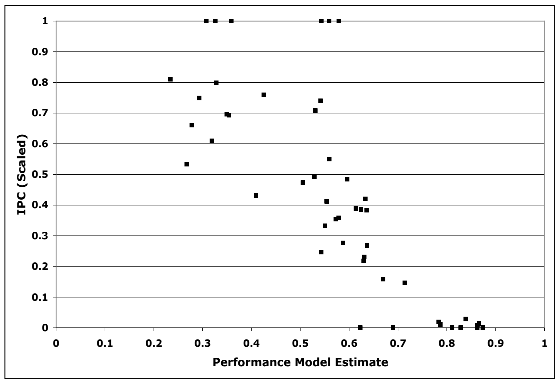


Figure 15: **Performance Model v. Simulated IPC** The performance predicted by the complete model (x-axis) correlates with actual, simulated performance (y-axis) with a coefficient of -.82.

dispersion to balance the load more evenly. *Latency* and *data* capture the first concern while *bandwidth* and *contention* encapsulate the second.

Here we unify these four metrics, balancing the packing-dispersion tradeoff by weighting each model according to its relative *contribution*. The overall model is simply a weighted sum of each of the four sub-models as shown in Equation 12. The coefficient for a component, such as $Contrib_{latency}$ for *latency*, is the average contribution across all benchmarks.

$$\begin{aligned} EstimatedPerformance \quad = \quad & \\ & Contrib_{latency} \times Latency \\ & Contrib_{data} \times Data \\ & Contrib_{bandwidth} \times Bandwidth \\ & Contrib_{PeContention} \times PEContention \end{aligned} \tag{12}$$

For each of the forty-eight data points (six benchmarks, eight layouts each) we calculated a combined model score according to Equation 12. We then simulated each layout on a fully-featured, non-idealized simulator. The graph in Figure 15 shows the correlation between predicted and measured performance. The correlation coefficient for this data is -.82.

Because each benchmark has a different amount of inherent parallelism, we must normalize the IPCs for each application. A normalized IPC of 0 is simply the lowest observed IPC for some benchmark, while 1 is the highest. Similarly we must normalize the component model values before combining them into a single model. This is because each component produces an output with a different order of magnitude. For example raw *latency* scores, as they are a sum over all of the dataflow edges in the application, are quite large, on the order of millions, while raw *data* scores are miss rates ranging from 0 to 1. The component model outputs are normalized, per model, per application.

While this combined model performs well, it is not exactly fair to evaluate the model on the very same data from which it was derived. Because the model is meant to be predictive, a more meaningful evaluation of its quality would be to measure how well it predicts the performance of an application to which it has not yet been exposed. This practice of separating data into *training* and *test* sets is commonplace in the field of machine learning. A model must be developed based on the *training* data and evaluated using the *test* data.

		Correlation Coefficients	
Training Set	**Test Set**	**Training Data**	**Test Data**
all except fft	fft	-0.86	-0.83
all except ocean	ocean	-0.89	-0.68
all except equake	equake	-0.83	-0.95
all except art	art	-0.84	-0.88
all except gzip	gzip	-0.85	-0.87
all except mcf	mcf	-0.84	-0.90
Average			-.85

Table 3: **Combined Performance Model Evaluation** The first column of data in shows the average correlation of predicted to actual performance for the training data. The second column shows its correlation to the test set.

With this evaluation technique in mind, we divided our benchmark data into six pairs of trianing and test data sets. For instance, one training set consists of all benchmarks except *fft* and the corresponding tests set contains only *fft*. We then built each model based only on the training data. Deriving a model from certain benchmarks means that the average contributions were calculated based only on those benchmarks. The resulting model was then evaluated on the test data. The first column of data in Figure 3 shows the average correlation of predicted to actual performance for the training data. The second column shows its correlation to the test set.

As can be seen in Table 3, the model's prediction still correlates quite strongly with actual performance. The average coefficient of correlation is -.85. We believe that this is an extremely positive result, indicating that the model developed here provides an accurate and quickly calculable heuristic with which to guide layout optimization.

8 Conclusion

This paper demonstrates a framework by which an accurate performance model for a distributed ILP architecture can be developed. The predicted performance from model developed herein correlates to actual performance with a -.85 correlation. This accuracy makes the model not only a fast but effective tool with which to evaluate an instruction placement on the WaveScalar architecture. Such a tool enables a heuristic search through the space of application placements for the one which will result in the best application performance.

References

[1] M. S. Lam and R. P. Wilson, “Limits of control flow on parallelism,” in *Proceedings of the 19th Annual International Symposium on Computer Architecture*, (Gold Coast, Australia), pp. 46–57, ACM SIGARCH and IEEE Computer Society TCCA, May 19–21, 1992. *Computer Architecture News*, 20(2), May 1992.

[2] S. C. Goldstein and M. Budiu, “Nanofabrics:spatial computing using molecular electronics,” in *Proceedings of the 28th annual international symposium on Computer architecture*, pp. 178–191, 2001.

[3] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore, “Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture,” in *Proceedings of the 30th annual international symposium on Computer architecture*, 2003.

[4] W. Lee *et al.*, “Space-time scheduling of instruction-level parallelism on a Raw machine,” in *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems ASPLOS-VIII*, October 1998.

[5] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. Dally, and M. Horowitz, “Smart memories: A modular reconfigurable architecture,” in *International Symposium on Computer Architecture*, 2002.

[6] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin, “WaveScalar,” in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, p. 291, 2003.

[7] J. B. Dennis, “A preliminary architecture for a basic dataflow processor,” in *Proceedings of the 2nd Annual Symposium on Computer Architecture*, 1975.

[8] A. L. Davis, “The architecure and system method of DDM1: A recursively structured data driven machine,” in *Proceedings of the 5th Annual Symposium on Computer Architecture*, (Palo Alto, California), pp. 210–215, IEEE Computer Society and ACM SIGARCH, April 3–5, 1978.

[9] S. Sakai, y. Yamaguchi, K. Hiraki, Y. Kodama, and T. Yuba, “An architecture of a dataflow single chip processor,” in *Proceedings of the 16th annual international symposium on Computer architecture*, pp. 46–53, ACM Press, 1989.

[10] T. Shimada, K. Hiraki, K. Nishida, and S. Sekiguchi, “Evaluation of a prototype data flow processor of the sigma-1 for scientific computations,” in *Proceedings of the 13th annual international symposium on Computer architecture*, pp. 226–234, IEEE Computer Society Press, 1986.

[11] J. R. Gurd, C. C. Kirkham, and I. Watson, “The manchester prototype dataflow computer,” *Communications of the ACM*, vol. 28, no. 1, pp. 34–52, 1985.

[12] M. Kishi, H. Yasuhara, and Y. Kawamura, “Dddp-a distributed data driven processor,” in *Conference Proceedings of the tenth annual international symposium on Computer architecture*, pp. 236–242, IEEE Computer Society Press, 1983.

[13] V. G. Grafe, G. S. Davidson, J. E. Hoch, and V. P. Holmes, “The epsilon dataflow processor,” in *Proceedings of the 16th annual international symposium on Computer architecture*, pp. 36–45, ACM Press, 1989.

[14] G. Papadopoulos and D. Culler, “Monsoon: An explicit token-store architecture,” in *Proceedings of the 17th International Symposium on Computer Architecture*, May 1990.

[15] A. Schwerin, “Multithreading the wavecache,” Master’s thesis, University of Washington, 2003.

[16] SPEC, “Spec CPU 2000 benchmark specifications.” SPEC2000 Benchmark Release, 2000.

[17] M. Gokhale, W. Holmes, A. Kopser, S. Lucas, R. Minnich, D. Sweeney, and D. Lopresti, “Building and using a highly parallel programmable logic array,” *Computer*, vol. 24, Jan. 1991.

[18] V. Betz and J. Rose, “VPR: A new packing, placement and routing tool for FPGA research,” in *Field-Programmable Logic and Applications* (W. Luk, P. Y. Cheung, and M. Glesner, eds.), pp. 213–222, Springer-Verlag, Berlin, 1997.

[19] C. Chang, J. Cong, D. Pan, and X. Yuan, “Multilevel global placement with congestion control,” 2003.

[20] J. Ghosh and K. Hwang, “Critical issues in mapping neural networks on message-passing multicomputers,” in *Proceedings of the 15th Annual International Symposium on Computer Architecture* [32], pp. 3–11. *Computer Architecture News*, 16(2), May 1988.

[21] D. E. Culler and Arvind, “Resource requirements of dataflow programs,” in *Proceedings of the 15th Annual International Symposium on Computer Architecture* [32], pp. 141–150. *Computer Architecture News*, 16(2), May 1988.

[22] M. B. Taylor, W. Lee, S. Amarasinghe, and A. Agarwal, “Scalar operand networks: On-chip interconnect for ilp in partitioned architectures,” in *Proceedings of the The Ninth International Symposium on High-Performance Computer Architecture (HPCA’03)*, p. 341, IEEE Computer Society, 2003.

[23] D. B. Noonburg and J. P. Shen, “A framework for statistical modeling of superscalar processor performance,” pp. 298–309.

[24] S. Nussbaum and J. Smith, “Modeling superscalar processors via statistical simulation,” 2001.

[25] M. Oskin, F. T. Chong, and M. K. Farrens, “HLS: combining statistical and symbolic simulation to guide microprocessor designs,” in *ISCA*, pp. 71–82, 2000.

[26] L. Eeckhout, K. D. Bosschere, and H. Neefs, “Performance analysis through synthetic trace generation.”

[27] T. Conte, M. Hirsch, and K. Menezes, “Reducing state loss for effective trace sampling of superscalar processors,” pp. 468–477.

[28] V. S. Adve and M. K. Vernon, “The Influence of Random Delays on Parallel Execution Times,” in *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp. 61–73, May 1993.

[29] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken, “LogP: Towards a realistic model of parallel computation,” in *Principles Practice of Parallel Programming*, pp. 1–12, 1993.

[30] A. Agarwal, “PERFORMANCE TRADEOFFS IN MULTITHREADED PROCESSORS,” Tech. Rep. MIT/LCS/TR-501, 1991.

[31] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger, “Clock rate versus IPC: The end of the road for conventional microarchitectures,” in *27th International Symposium on Computer Architecture*, 2000.

[32] IEEE Computer Society TCCA and ACM SIGARCH, *Proceedings of the 15th Annual International Symposium on Computer Architecture*, (Honolulu, Hawaii), May 30–June 2, 1988. *Computer Architecture News*, 16(2), May 1988.