Modeling Instruction Placement on a Spatial Architecture

Martha Mercaldi Steven Swanson Andrew Petersen Andrew Putnam
Andrew Schwerin Mark Oskin Susan J. Eggers
Computer Science & Engineering
University of Washington
Seattle, WA USA
{mercaldi,swanson,petersen,aputnam,schwerin,oskin,eggers}@cs.washington.edu

ABSTRACT

In response to current technology scaling trends, architects are developing a new style of processor, known as spatial computers. A spatial computer is composed of hundreds or even thousands of simple, replicated processing elements (or PEs), frequently organized into a grid. Several current spatial computers, such as TRIPS, RAW, SmartMemories, nanoFabrics and WaveScalar, explicitly place a program's instructions onto the grid.

Designing instruction placement algorithms is an enormous challenge, as there are an exponential (in the size of the application) number of different mappings of instructions to PEs, and the choice of mapping greatly affects program performance. In this paper we develop an instruction placement performance model which can inform instruction placement. The model comprises three components, each of which captures a different aspect of spatial computing performance: inter-instruction operand latency, data cache coherence overhead, and contention for processing element resources. We evaluate the model on one spatial computer, WaveScalar, and find that predicted and actual performance correlate with a coefficient of -0.90. We demonstrate the model's utility by using it to design a new placement algorithm, which outperforms our previous algorithms. Although developed in the context of WaveScalar, the model can serve as a foundation for tuning code, compiling software, and understanding the microarchitectural trade-offs of spatial computers in general.

Categories and Subject Descriptors

I.6.5 [**Computing Methodologies**]: Simulation and Modeling—*Model Development*; B.8.2 [**Hardware**]: Performance and Reliability—*Performance Analysis and Design Aids*

General Terms

Experimentation, Measurement, Performance

Keywords

dataflow, instruction placement, spatial computing

1. INTRODUCTION

Today's manufacturing technologies provide an enormous quantity of computational resources. Computer architects are currently exploring how to convert these resources into improvements in application performance. Despite significant differences in execution models and underlying process technology, five recently proposed architectures - nanoFabrics [18], TRIPS [34], RAW [23], SmartMemories [26], and WaveScalar [39] - share the task of mapping large portions of an application's binary onto a collection of processing elements. Once mapped, the instructions execute "in place", explicitly sending data between the processing elements. Researchers call this form of computation *distributed ILP* [34, 23, 39] or *spatial computing* [18].

Good instruction placement is critical to spatial computing performance. Our research on WaveScalar indicates that a poor placement can decrease performance by as much as a factor of five. Finding a good placement is hard, because there are an exponential (in the size of the application) number of possible mappings. How can developers, compiler writers, or microarchitects identify the ones that will execute quickly? Searching this enormous space requires a solid understanding of how instruction placement influences performance. In this paper we develop a model of placement performance to study this issue.

To develop the model, we focus on a particular spatial computer, WaveScalar. To accurately predict instruction placement performance, we construct a unified model that considers several factors that contribute to overall performance. Our model comprises three separate components, each of which captures a different aspect of spatial computation: inter-instruction operand latency, data cache coherence overhead, and contention for processing element resources. Our unified model combines these components in proportion to their relative contribution to overall performance.

The model estimates performance using three inputs: (1) the placement in question, i.e., a mapping of instructions in the application to processing elements, (2) a profile of application execution behavior, and (3) the spatial computer's microarchitectural configuration and timing parameters. These inputs are common to all spatial computers, which will allow this approach to generalize beyond WaveScalar.

The paper first develops a model of each component of placement performance in isolation. Using a variety of applications and potential placements, we evaluate each of these component models, using specially configured versions of the WaveScalar microarchitectural simulator. Each configuration accurately simulates the hardware resources of the component in question but idealizes all other resources. We validate each component model by showing that it correlates with its component-isolating simulation.

We then combine these component models to produce a single unified model of placement performance on WaveScalar. The unified model predicts the effect of instruction placement when *all* microarchitectural resources are accurately simulated. The combined model produces performance predictions that correlate to simulation performance with a coefficient of −0.90.

To evaluate our model's predictive power on applications that are not part of our workload, we use a standard machine learning evaluation technique in which we partition our data points into *training* and *test* sets. We derive a model from each of the training sets, and evaluate its predictive capability on its corresponding test set. Evaluated in this way, our model's predicted layout performance correlates to actual performance with a coefficient of −0.82.

The model indicates that PE resource constraints have the greatest effect on placement performance on WaveScalar, followed by inter-instruction operand latency, and finally by cache coherence overhead. These results are useful in several ways. For example, the model provides a quickly calculable objective function that an optimizer could minimize to find an application mapping that maximizes IPC. One could also use the model to design an instruction placement algorithm which is based on the factors that are most important to performance. In Section 6 we do just this and develop an improved placement algorithm by combining two existing algorithms that optimize for the two most important components of placement performance, as dictated by the model. A third strategy is to use the model to guide microarchitectural optimizations or to make the microarchitecture less placement-sensitive.

In the following section we provide an overview of the salient features of WaveScalar. In Section 3 we present the methodology used to develop and validate our placement performance model. Section 4 explains and validates each of the individual components, and Section 5 combines them into a unified model. Section 6 describes an improved instruction placement algorithm we developed that is based on this model. Section 7 explores related work on performance modeling, layout of computation, and spatial computers. Finally in Section 8, we draw our conclusions and discuss future work in this area.

2. BACKGROUND

Before introducing our instruction placement model, we summarize the WaveScalar architecture and its processor implementation. We confine our discussion to those features that provide a context for modeling instruction placement performance, as presented in this paper. A more in-depth description appears in other publications [39, 40, 41].

Instruction set architecture: WaveScalar is a dataflow architecture. Like all dataflow architectures (e.g. [12, 11, 22, 36, 20, 32, 33, 19, 31, 10, 2]), its binary is a program's dataflow graph. Each node in the graph is a single instruction which computes a value and sends it to the instructions that consume it. Instructions execute after all input operand values have arrived, according to a principle known as the *dataflow firing rule* [12, 11].

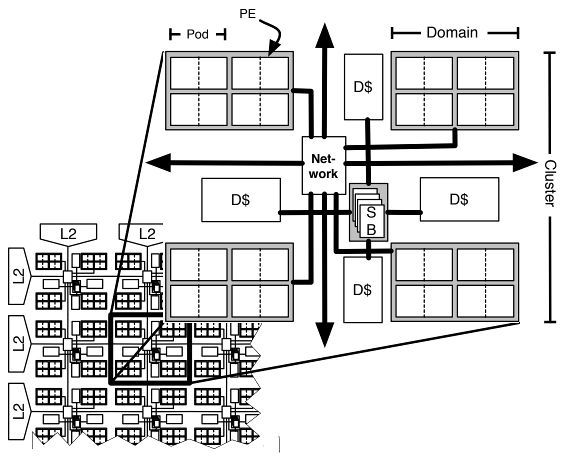


Figure 1: The WaveScalar Processor: The hierarchical organization of the WaveScalar processor.

WaveScalar supports a memory model which commits memory accesses in program order. Equipped with architectural building blocks, called *waves*, which globally order pieces of the control flow graph, and an architectural mechanism, called *wave-ordered memory*, which orders memory operations within a wave, WaveScalar enforces the correct, global ordering of a thread's memory operations. This enables it to execute applications written in imperative languages, such as C or C++. Other work describes the details of this mechanism [39].

Microarchitecture: Conceptually, each static instruction in a WaveScalar program executes in a separate processing element (PE). Building a PE for each static instruction is both impossible and wasteful, so, in practice, WaveScalar dynamically binds multiple instructions to a fixed number of PEs, and swaps them in and out on demand.

The WaveScalar processor is a grid of simple processing elements. Each PE has five pipeline stages and contains a functional unit, specialized memories to hold operands, and logic to control instruction execution and communication. Each PE also contains buffering and storage for several different static instructions, although only one can execute in any given cycle. PEs determine locally when their instructions can execute, contributing to the scalability of the WaveScalar processor design.

To reduce communication costs within the grid, PEs are organized hierarchically, as depicted in Figure 1. Two PEs are first coupled, forming a *pod*; PEs in a pod share bypass logic, so that dependent instructions in a pod can execute back to back. Four pods comprise a *domain*. Within a domain producer-consumer latency is four cycles. Four domains, plus wave-ordered memory hardware and a traditional L1 data cache, make up a *cluster*. A single cluster, combined with an L2 cache and traditional main memory is sufficient to run any WaveScalar program. To build larger machines, an on-chip network connects multiple clusters and a directory-based, MESI-like protocol maintains cache coherence. The coherence directory and the L2 cache are distributed around the edge of the grid of clusters.

PEs per Domain	8 (4 pods)	Domains / Cluster	4
PE Input Queue	16 entries, 4 banks	Network Latency	within Pod: 1 cycle within Domain: 4 cycles within Cluster: 7 cycles inter-Cluster: 7 + cluster distance
PE Output Queue	8 entries, 4 ports (2r, 2w)		
PE Pipeline Depth	5 stages		
L1 Caches	32KB, 4-way set associative, 128B line, 4 accesses per cycle	L2 Cache	16 MB shared, 1024B line, 4-way set associative, 20 cycle access
Main RAM	1000 cycle latency	Network Switch	4-port, bidirectional

Table 1: Microarchitectural parameters of the WaveScalar processor

Table 1 contains the WaveScalar processor microarchitecture configuration parameters used for this study.

3. METHODOLOGY

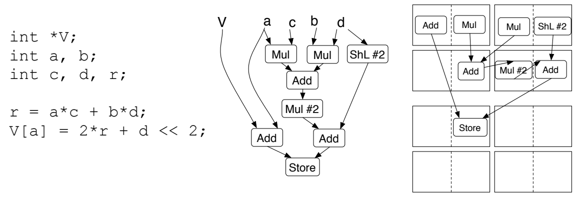
This section describes our methodology for developing an instruction placement model for the WaveScalar processor. The overall goal is to construct a function that takes three pieces of information – (1) a proposed instruction placement for an application, (2) a simple execution profile of the application, and (3) the microarchitectural parameters of the WaveScalar processor – and computes a prediction of placement quality with respect to performance. Once this function is found, we can quickly and easily evaluate the quality of alternative instruction placements.

We begin by separately modeling several factors that affect performance and vary with instruction placement. These factors are: inter-instruction operand latency, the coherence cost of sharing data in memory, and contention for instruction memory in individual PEs. Each becomes a component of the overall instruction placement model. Section 4 describes each in detail.

We exclude other potential factors that we found have little or no effect on the performance of the WaveScalar processor. Examples include main memory latency and network bandwidth. In WaveScalar, the amount of time spent accessing memory is the same no matter where in the grid an instruction is located. Likewise, WaveScalar is sufficiently provisioned in its network bandwidth that this factor also does not affect performance.

We evaluate each of the components using a cycle-accurate, event-driven WaveScalar simulator, whose microarchitecture has been idealized with respect to capacity (infinite), latency (zero), and bandwidth (infinite) for all aspects of execution *except* the hardware that implements the particular component under scrutiny. Isolating a component in this way directly measures its effect on overall performance. This is the same methodology that previous studies used to isolate the effect of a single architectural feature [9]. In that work the feature was finite execution resources. We extend the technique by studying multiple effects in isolation and demonstrating how they can be combined. We show in Section 5 that this approach is ultimately successful – when all components are combined and then tested against a simulator that models the microarchitecture realistically, the combined model accurately predicts placement quality.

In our experiments, we used a set of eight benchmarks comprised of different types of applications: four from SpecInt [38] (*gzip*, *mcf*, *twolf*, *vpr*), two from SpecFP (*equake*, *art*), and two from Splash2 [17] (*fft*, *lu*). For each benchmark we have generated eight possible instruction placements, which produce a range of operand latencies, coherence overhead, and processing element contention.


```
int *V;
int a, b;
int c, d, r;

r = a*c + b*d;
V[a] = 2*r + d << 2;
```

Figure 2: WaveScalar Application Layout: The figure contains two domains.

3.1 Instruction Placements

For spatial processors, the placement phase of instruction scheduling produces an instruction layout that maps units of computation to locations in the computational substrate. In WaveScalar, placement maps WaveScalar instructions onto processing elements in the WaveScalar processor. Figure 2 illustrates an application and a sample layout.

We use eight different instruction placement algorithms in this study:

random: Assigns each instruction randomly to a PE anywhere in the processor.

packed-random: Assigns each instruction to a randomly chosen PE from a restricted set of contiguous domains. The size of this set is the minimum number of domains required to hold all of the program instructions.

static-snake: Assigns instructions to PEs in static program order (i.e., as they appear in the binary). The algorithm fills each domain completely before moving on to the next. Similarly, the algorithm fills all domains in a cluster before moving on to the next cluster. It fills clusters in a "snaking" order: left to right on the first row, right to left on the second, then left to right on the third, and so on.

depth-first-snake: This is a depth-first, search-based algorithm that computes a pre-order traversal of the instructions in the DFG. It then assigns each group of 64 instructions from this pre-order to a PE. The goal of this algorithm is to place data-dependent instructions in the same PE, as these instructions cannot fire in parallel and therefore can share a PE without contending for the execution unit. DEPTH-FIRST-SNAKE fills PEs in the same order as in STATIC-SNAKE.

over-2-DFS, over-4-DFS, over-8-DFS: These are the same as DEPTH-FIRST-SNAKE, except that they assign

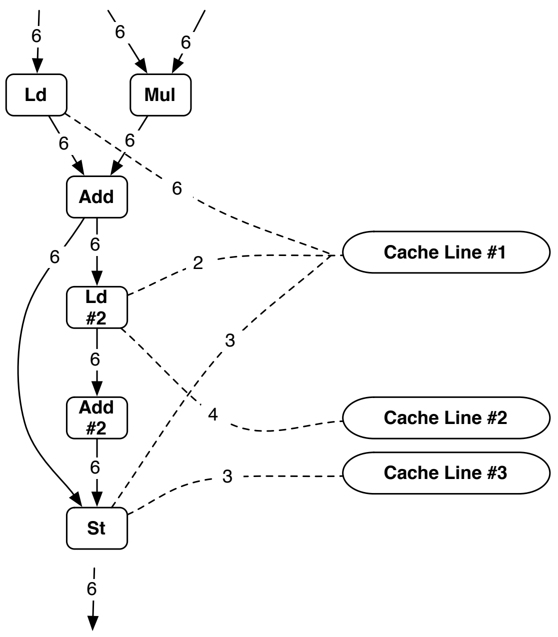


Figure 3: WaveScalar Application Profile

two, four or eight times as many instructions (128, 256, and 512 respectively) to each PE as it can actually hold.

dynamic-snake: Assigns instructions to processing elements in dynamic program order. It fills PEs in the same manner as STATIC-SNAKE.

We selected this set of placement algorithms to represent a variety of approaches to the placement problem. RANDOM and RANDOM-PACKED produce very naive layouts which represent lower bounds on performance. STATIC-SNAKE emulates an instruction prefetching algorithm. DEPTH-FIRST-SNAKE attempts to utilize the sequential nature of dependent instructions in a productive way for placement. The over-subscribed DFS algorithms explore contention effects in the architecture. Finally, DYNAMIC-SNAKE, which uses the dynamic instruction execution order, has a performance advantage over the other algorithms, which use the static instruction order. Any spatial computer could implement the first seven algorithms, while only WaveScalar provides the dynamic instruction placement capabilities required by the last.

3.2 Input Application and Profile

In this work we do not attempt to parallelize or otherwise optimize application binaries for a particular instruction placement or for spatial computing in general. For example all loops were already unrolled by a factor of four. We therefore profiled the binaries as they were to produce an annotated dataflow graph of the program. Figure 3 depicts an example. Profiling annotates edges with the number of times a value flowed along it during profiled execution. Profiling also adds a second type of node, an address node, which represents a cache line address that the application accessed. Address nodes are connected to the instructions which access them. These edges also have weights that indicate the number of profiled accesses.

4. COMPONENTS OF PERFORMANCE

This section presents the three components of the instruction placement model that most affected performance: operand latency, coherence overhead, and contention for PE resources. We describe each in isolation and present simulation data that illustrates their value in predicting execution performance from an application's instruction layout, profile and the processor's microarchitectural parameters. Where appropriate, we will briefly describe some of the alternative components which we had hypothesized would affect performance, but turned out not to. Section 5 combines the three individual components into a unified instruction placement model.

Throughout this paper we use the following common set of variables:

- n is the number of static instructions in the application. i refers to the i'th instruction, j refers to the j'th instruction.
- C_i is a cluster, expressed as an x, y coordinate that contains instruction i ($C_i = (C_{x_i}, C_{y_i})$). Similarly, D_i, P_i, and p_i are the locations of the domain, the pod, and the processing element, respectively, where instruction i is placed.
- $T_{i,j}$ refers to the amount of communication (or traffic) between instructions i and j. This is simply the profile edge weight.
- a refers to a particular cache line address.
- C_a is the number of clusters that contain copies of the cache line with address a.
- N_a is the total number of accesses to address a across all clusters throughout execution.
- I_P is the number of instructions assigned to PE p.
- $PeCapacity$ is the maximum number of instructions that can be resident at a PE at any moment. In this study $PeCapacity = 64$.

We use the Manhattan-distance for all distance calculations in our work. For example, the cluster distance, $||C_i - C_j||$ is $|Cx_i - Cx_j| + |Cy_i - Cy_j|$.

The results present data both graphically and in tabular form. For the graphs, we computed each component model's output (the X-axis) from an instruction layout, an application profile, and the WaveScalar microarchitectural configuration. We then compared this to the results of a cycle-level simulation executing the application with the same layout (this is the Y-axis). As previously explained, when testing a component, we idealized the microarchitecture, except for the hardware that implements it. For the unified model evaluation, we simulated all hardware realistically.

The tabular data presents two metrics: *correlation* and *contribution*. Correlation is the standard statistical correlation, computed between the model's output for a layout and the measured performance of its corresponding simulation. It indicates how closely the model of a component matches its simulation. We use correlation, because our goal is to estimate the relative quality of layouts instead of their absolute performance. Because the models are structured as cost functions, the correlations are negative (with increased

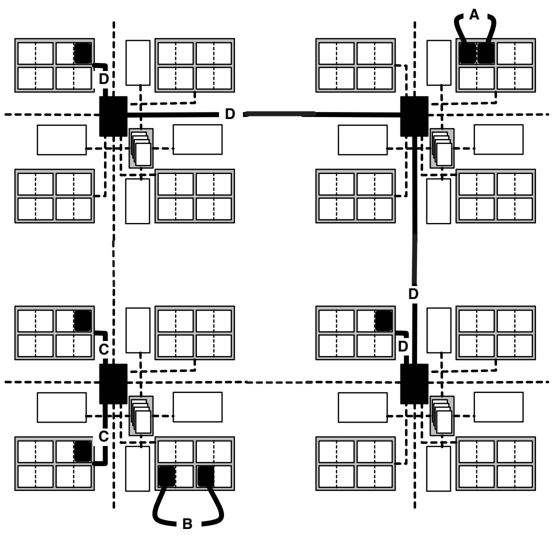


Figure 7: Operand Latency: We expect operand communication to have varying latencies depending on how instructions are placed. This figure illustrates the four types of WaveScalar operand communication.

cost translating into lower performance in IPC). Contribution represents the importance of a component to overall performance; it captures the degree to which overall performance varies with the component value. We calculate contribution by taking the variance of the simulated IPC and dividing it by the average IPC. Components with high contribution values will influence overall performance more than those with lower contribution values.

4.1 Operand Latency

With shrinking process sizes, on-chip communication is expected to increasingly dominate performance [1]. Operands traveling from instruction to instruction in spatial computers is one type of on-chip communication. The latency of this communication is dependent almost exclusively on the instruction layout. For example, two data-dependent instructions placed on opposite sides of the WaveScalar processor will require many cycles to communicate. If this pair of instructions executes frequently, performance suffers even more.

Figure 7 illustrates the four different types of communication in WaveScalar's hierarchical communication structure: long distance, inter-cluster communication (labeled **D**); intra-cluster but inter-domain communication (labeled **C**); intra-domain but inter-pod communication (**B**); and intra-pod communication (labeled **A**). Each type of communication has a different latency. Intra-pod communication occurs over shared bypass buses and is zero cycles. Within a domain the latency from the execution pipeline stage in one PE to the execution stage in another is 4 cycles. Within the same cluster, but across domains, communication travels through the cluster switch and requires 7 cycles. Finally, long distance communication, requires a time proportional to its distance: 7 cycles for level **C** communication, plus 1 cycle for each inter-cluster network link traversed.

We model operand latency based on the distance between the producer and consumer of the operand. Considering any two instructions, i and j, the latency between them is dependent on their locations:

$$Latency_{i,j} = \begin{cases} 0 & \text{if } P_i = P_j, \\ 4 & \text{if } D_i = D_j, \\ (||C_i - C_j|| + 7) & \text{otherwise.} \end{cases} \quad (1)$$

The total latency that operand traffic incurs is therefore the summation of this value for each pair of instructions multiplied by the frequency of communication between them:

$$Latency = \sum_i \sum_j T_{i,j} \times Latency_{i,j} \quad (2)$$

We can compute this estimate quickly from profile data and an instruction layout. Figure 4 contains the results of isolating operand latency. The table on the left is the correlation between *Latency* and application performance. The average correlation coefficient is -0.88. The figure on the right shows raw performance versus operand latency for a representative application; the X-axis is the latency metric and the Y-axis is simulated performance (IPC)

The high correlations across all applications indicate that the operand latency model captures the effect of actual operand latency induced by an application layout: higher latency leads to lower performance. (*Art* on the right side of Figure 4 typifies this strong inverse correlation.) An average contribution factor of 0.84 indicates that operand latency is a significant contributor to overall application performance. Taken together, the correlation and contribution results indicate the extent to which operand latency affects overall performance and that it is accurately reflected by our model.

This model examines only operand latency over the network and ignores any limitations on bandwidth. We originally included an additional component in the model to capture network bandwidth. However, with 4 ports per cardinal direction for each of the inter-cluster network switches, we found that the WaveScalar design provided sufficient bandwidth capability and consequently bandwidth did not constrain performance.

4.2 Data Cache Coherence

The second component of placement performance is coherence overhead. Each WaveScalar cluster has an L1 data cache. When an instruction accesses memory, it first checks its local L1 cache. If the line is present, the operation can continue; if not, the directory-based coherence system fetches it, either from another L1 cache or from the L2 cache.

From the application profile, we can identify which instructions access which addresses in memory. Using this information, we evaluate an instruction layout, based on how the memory accesses might tax the distributed cache system. Ideally, a placement algorithm would assign all of the instructions which access a common cache line to the same cluster, thereby allowing the instructions to share a single cache line in the local L1 cache. However, when two instructions in *different* clusters write the same cache line, the coherence system will transfer ownership of the cache line with each write. This transfer of ownership occurs both for addresses that are accessed by multiple threads (as happens on traditional shared memory systems) and *within a single thread.*

The number of ownership transfers depends on the temporal order of the cache accesses [16]. Because the profiles

Benchmark	Correlation Coefficient	Contribution
art	-0.90	0.56
equake	-0.92	0.41
fft	-0.88	1.52
gzip	-0.93	0.52
lu	-0.86	1.45
twolf	-0.89	0.65
vpr	-0.84	0.85
mcf	-0.80	0.78
Average	-0.88	0.84

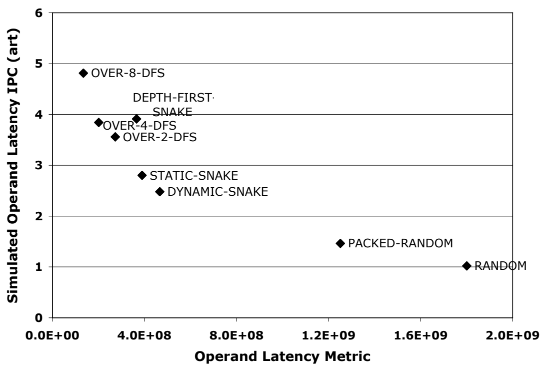


Figure 4: Operand Latency Metric Evaluation: On the left is the correlation between simulated performance and operand latency for our application suite. Since higher operand latency should lower IPC, a perfect correlation would be -1.0. On the right is a graph depicting results for a single application, *art*. Each of the eight data points represents one layout. The X-coordinate is the operand latency metric, while the Y-coordinate is the simulated operand latency IPC. Other applications' correlations are qualitatively similar.

Benchmark	Correlation	Contribution
art	-0.92	0.49
equake	-0.99	0.18
fft	-0.33	0.35
gzip	-0.95	0.33
lu	-0.64	0.79
twolf	-0.92	0.45
vpr	-1.00	0.03
mcf	-0.97	0.12
Average	-0.84	0.34

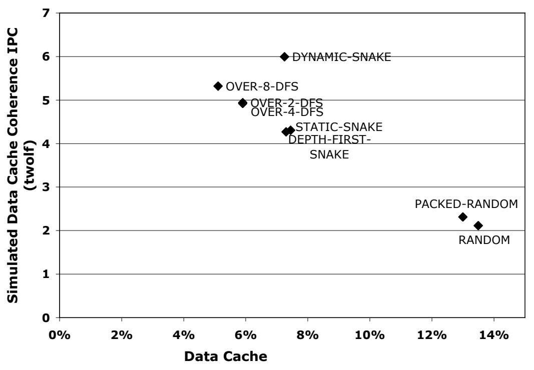


Figure 5: Coherence Overhead Evaluation: On the left is the correlation of the estimated shared data miss rate to simulated IPC for each of the benchmarks and the contribution of this metric to overall performance. Because an increased miss rate should degrade memory performance, an ideal correlation would be -1.0. On the right is a graph that depicts simulated memory system performance versus the coherence overhead metric for each instruction layout for *twolf*.

Benchmark	Correlation	Contribution
art	-0.69	1.22
equake	-0.84	0.28
fft	-0.74	2.43
gzip	-0.83	0.76
lu	-0.65	2.58
mcf	-0.83	0.65
twolf	-0.79	1.02
vpr	-0.67	0.73
Average	-0.76	1.21

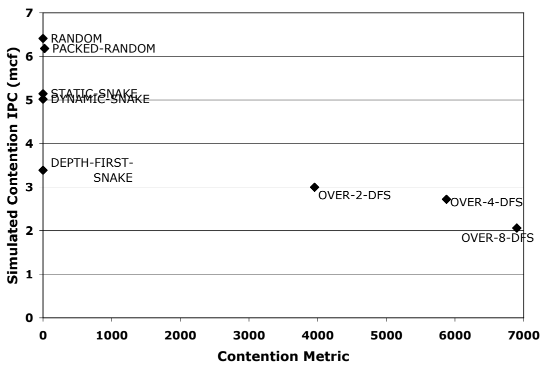


Figure 6: PE Contention Evaluation: The table on the right shows the correlation of our PE contention model to the simulated IPC. The graph shows the detailed data for one application: *mcf*

do not capture this information, any model we build must make an assumption about how ownership of a cache line changes over time. We assume that all intra-thread sharing is migratory, meaning that writes to a line from different clusters are not interleaved over time. We build the model of the coherence system, shown in Equations 3 and 4, based on this assumption. The model assumes that the coherence system migrates each cache line a between L1 caches only once for each of the C_a clusters that accesses it.

$$Misses_a = \begin{cases} 1 & \text{if } C_a == 1, \\ C_a & \text{if } C_a > 1 \end{cases} \tag{3}$$

$$Hits_a = \begin{cases} N_a - 1 & \text{if } C_a == 1, \\ N_a - C_a & \text{if } C_a > 1 \end{cases} \tag{4}$$

To test this migratory model, we simulated with no operand latency, bandwidth constraints, or execution resource constraints; only the effect of the data caches throttles performance. We measured the correlation between the miss ratio predicted by the coherence model and the resulting application performance. The results, the table in Figure 5, support the use of a migratory sharing scenario to model distributed data cache behavior. The actual and predicted performance correlate with an average coefficient of -0.84. However, at 0.34, its contribution to performance is lower than operand latency.

An alternate sharing model assumes inter-cluster write contention, where an instruction in one cluster writes a cache line, followed by an instruction in a second cluster, followed by a third, and so on. Each write will require action from the coherence system. While in theory a situation such as this can occur, in practice it does not. Fewer than 1% of the cache lines used by an application exhibit any significant contention. Consequently, the contention-based coherence model did not correlate as well as the migratory model.

We did not include common cache effects, such as conflict and capacity misses, in our instruction placement model because our experiments show that the principle change in performance between two instruction layouts is attributable to sharing misses, rather than these other types of cache misses.

4.3 Processing Element Contention

The final component of the instruction placement model is contention for instruction storage at the processing elements (PEs). PEs in the WaveScalar processor can hold a limited number of instructions. If the number of instructions assigned to a PE exceeds this limit, the processor stores the excess instructions in memory and swaps them in when required during execution. PEs use an LRU algorithm to decide which instruction to evict. In all of the experiments presented in this paper, the PEs hold 64 instructions and each instruction swap takes 32 cycles.

We model the contention among instructions at a PE by counting the number of excess instructions at each PE:

$$PeContention = \sum_p \begin{cases} |I_p| - PeCapacity & \text{when } |I_p| > PeCapacity \\ 0 & \text{otherwise.} \end{cases} \tag{5}$$

This straightforward model is highly effective at capturing contention for instruction storage. Figure 6 shows the experimental results. As with the other comnponents, perfect correlation would be -1.0, as more contention should translate to lower performance. The average correlation across all benchmarks and all layouts is -0.76. The average contribution is 1.21, indicating that contention, like operand latency, is an important element of overall performance.

Examining the table in Figure 6, we see that the PE instruction contention metric correlates well with IPC when all aspects of execution except for the PE resources are idealized. The exception is when the model predicts there will be no contention (the points on the Y-axis of the graph in Figure 6). Other statistics gathered from the microarchitectural simulator indicate that in these cases contention that is not modeled for other PE resources, such as the operand input queue, the instruction dispatch queue, and the ALU, produces variations in actual contention. Furthermore, metrics for these other types of contention correlate very closely with simulated IPC. For example, for *mcf*, the correlation coefficient of the average instruction scheduling queue length to IPC, is -0.93. This indicates a potentially fruitful area for future instruction placement model refinement.

5. COMBINED

Finally, we combine each of the component models into a unified model of instruction placement. Our goal is to predict relative WaveScalar performance for a particular instruction layout when we simulate all parts of the microarchitecture realistically. A tension between data sharing, which encourages that instructions be placed close together, and contention for hardware resources, which encourages physical dispersion to balance the load more evenly, is inherent in the unified model. Operand latency and coherence costs for shared data capture the first concern, while contention for instruction storage in processing elements encapsulates the second.

The unified model combines these three metrics, balancing the packing-dispersion trade-off by weighting each model according to its relative *contribution*. The overall model is simply a weighted sum of each of the three sub-models, as shown in Equation 6. The coefficient for a component, such as $Contribution_{Latency}$ for operand latency, is the average contribution across all benchmarks.

$$\begin{aligned} &PredictedPerformance = \\ &Contribution_{Latency} \times Latency + \\ &Contribution_{Data} \times Data + \\ &Contribution_{PeContention} \times PeContention \end{aligned} \tag{6}$$

For each of the sixty-four data points (eight benchmarks, eight layouts each), we calculated a combined model output according to Equation 6. We then simulated each layout with all microarchitectural parameters modeled realistically. Because each benchmark has a different amount of inherent parallelism, we normalized the IPCs for each application to between 0 and 1. We also normalized the component model metrics. Because some of the component models produce metrics that are different in scale from the others (e.g., as a weighted sum of edges, the latency metric might be very large, while the data cache miss rate will always be between 0 and 1), these values must be normalized before they can be combined meaningfully. Ultimately, the component model outputs are normalized twice, once per component model and once per application. Finally we adjust the contribution values so that they sum to 1. The resulting values are: $Contribution_{Latency} = 0.35$, $Contribution_{data} = 0.14$, and $Contribution_{PeContention} = 0.51$.

The graph in Figure 8 shows the correlation between predicted and measured performance of the instruction placement model derived from all data. The average correlation coefficient for all of the applications is -0.90.

An important caveat about this correlation is that we evaluated the model on the same data points from which we derived it. Because we intend the model to be predictive, a more rigorous evaluation of its quality is to measure how well it predicts the performance of a new application. Separating data into *training* and *test* sets is common in the field of machine learning: One develops a model from *training* data and then evaluates it on the corresponding *test* data.

Taking this approach, we divided our benchmark data into eight pairs of training and test data sets. For instance, one training set consists of all benchmarks except *fft* and the corresponding test set contains only *fft*. We then built a model based on each set of training data and evaluated it on its corresponding test data. The results, which appear in Figure 8, show that the instruction placement model is reliably capable of predicting the performance of new applications, with a -0.82 average correlation. Given this strong correlation, instruction placement algorithms that rely upon this model to gauge placement decisions can have confidence in those decisions.

6. AN IMPROVED PLACEMENT ALGORITHM

There are many uses for an instruction placement model. One is to help build better instruction placement algorithms. Among the eight instruction placement algorithms used in this study, two were particularly good at optimizing certain components: DEPTH-FIRST-SNAKE outperformed all of the others from an operand latency perspective, while DYNAMIC-SNAKE was best with respect to contention for instruction space in the PEs. The clear dominance of these factors, together accounting for 86% of instruction layout performance according to the model, as well as the fact that they address opposing instruction placement concerns, suggests that a blend of these two algorithms will be even better.

We examine DEPTH-FIRST-SNAKE first. This algorithm uses a depth first search of the dataflow graph to find dependent chains of producers and consumers. By placing instructions in a chain on the same PE, we ensure that those instruction operands use the low-latency local bypass network. Our simulation results confirm this intuition, with OVER-8-DFS which forms extremely long chains outperforming the nearest placement by 20% when measuring only operand latency performance.

However, with respect to PE contention, DYNAMIC-SNAKE, performs best. DYNAMIC-SNAKE packs the WaveScalar processor grid with instructions most efficiently, by filling PEs in dynamic execution order, thereby guaranteeing that PEs contain only instructions that are actually executed.

Following the intuition that our modeling work provided, we have designed a new placement algorithm, DYNAMIC-DEPTH-FIRST-SNAKE, which combines the best features of both of DEPTH-FIRST-SNAKE and DYNAMIC-SNAKE. Like DEPTH-FIRST-SNAKE, it fetches instructions in chains of producers and consumers. Like DYNAMIC-SNAKE, it places instruction chains in PEs in execution order. The example in Figure 9 explains the details of this algorithm further.

On average, DYNAMIC-DEPTH-FIRST-SNAKE improves on DEPTH-FIRST-SNAKE and DYNAMIC-SNAKE by 28% and 7%, respectively, with peak application gains of 60% and 43%, respectively. DYNAMIC-DEPTH-FIRST-SNAKE is currently the instruction placement algorithm we are using for ongoing WaveScalar research.

7. RELATED WORK

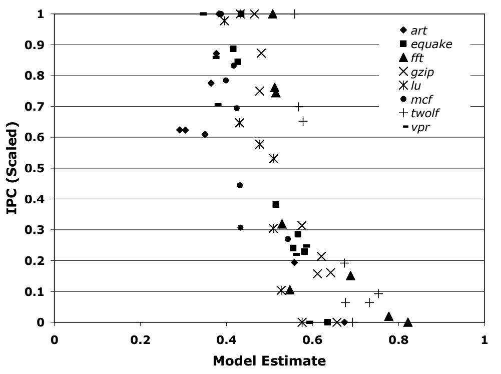
Performance modeling has a rich history. The placement model we present here leverages this body of work but also blends it with aspects from other domains, such as FPGA/ASIC CAD synthesis. Here we place our placement model in context between these related fields.

Related architectures: There are several recently designed spatial architectures, in addition to WaveScalar, which we believe could be modeled using the methodology described in this paper: nanoFabrics [18], TRIPS [34], RAW [23], and SmartMemories [26]. The nanoFabric [18] work is the most closely related. Operand communication latency is critical for efficient operation in nanoFabrics. TRIPS [34] is more centralized than WaveScalar but still has many distributed aspects. In particular our contention model should be applicable to contention for execution resources in physical TRIPS "frames". The RAW [23] microprocessor could use a similar model. In this case contention would be less of an issue because the static scheduler could resolve many temporal resource conflicts at each tile. Finally, the SmartMemories [26] processor should have similar issues of coherence sharing, latency, and contention even though it utilizes a coarser-grained thread.

Uniprocessor models: There is a wealth of previous work in the area of processor performance modeling in general. For superscalars the approaches have been statistical [28], [29], [30], trace-based [14], or both [14]. Because they focus on uniprocessors, these models treat computation sequentially – there is a single data cache hierarchy, and a single processor core. Hence, there is no "placement effect", as all computation occurs in one component. Issues such as operand latency simply do not appear in these uniprocessor models.

Multiprocessor models: Some models of parallel machines have focused on features that are central to distributed ILP architectures, such as parallel processing elements [8], [15] and distributed memory [44]. The parallelism in distributed ILP systems is significantly more fine grained than in these previous models. Each instruction in a distributed ILP system is conceptually a thread, and consequently the contention for resources among threads is significantly higher. Furthermore, in WaveScalar, there is a new form of migratory sharing that occurs within the *same* thread.

Place & Route: The problem of instruction placement for a distributed ILP architecture closely resembles FPGA place and route [3], [13], [37] and ASIC synthesis [6]. However, there are a number of subtle differences, the most significant of which is the scheduling of communication. With FPGAs and ASICs, communication and execution resources are scheduled statically, and once set, communication latencies are known and guaranteed. With distributed ILP archi-



		Correlation Coefficients	
Training Set	**Test Set**	**Training Data**	**Test Data**
all except art	art	-0.82	-0.76
all except equake	equake	-0.81	-0.89
all except fft	fft	-0.87	-0.74
all except gzip	gzip	-0.79	-0.83
all except lu	lu	-0.85	-0.77
all except mcf	mcf	-0.81	-0.95
all except twolf	twolf	-0.84	-0.76
all except vpr	vpr	-0.81	-0.89
Average			-0.82

Figure 8: Combined Performance Model Evaluation: On the left is a graph of instruction layout performance as predicted by the complete model (X-axis) versus realistic simulated performance (Y-axis). The correlation coefficient is -0.90. In the table, each row represents a different training set. The first column of data shows the average correlation of predicted to actual performance for a training data. The second column shows its correlation to the test set.

tectures, execution is far more dynamic. While inspired by place and route solutions, the placement model presented in this work handles dynamic effects, such as contention for instruction storage resources.

Our work builds upon a previous effort [9] which made an ideal dataflow execution more realistic by placing a limit on the number of available processing elements. We take this methodology a step further, adding other practical concerns (namely operand communication latency and distributed data cache coherence), and introducing a technique for combining multiple component models accurately.

Compiler optimizations: Any software transformations or optimizations that serve to increase or manage parallelism, such as k-loop bounding [7], memory access optimizations, such as tiling [5], target-specific optimizations, such as cyclic data decomposition and tiling [25], or specially designed languages [27, 43, 4, 24, 21], are complementary to this work. Such transformations serve to improve the potential performance of the application. The analytical model we present here inputs the compiled application, and its aim is to realize as much of the software-exposed parallelism as possible at execution time. Any hardware improvements one might apply [42, 35] are also complementary. As with the software transformations, such hardware optimizations can help performance, but are orthogonal to the analytical model.

8. CONCLUSION

This paper has presented an instruction placement model that drives instruction scheduling on WaveScalar's spatial microarchitecture. The model has three major components: operand latency, the coherence cost of sharing data, and contention for instruction memory in individual PEs. We developed and validated each of these components before combining them to produce a unified instruction placement model. The unified model predicts performance for different instruction layouts with a correlation of -0.90. Standard machine learning testing of the model achieves a correlation of -0.82.

The model provides empirical evidence that instruction layout heuristics for spatial processors, in this case WaveScalar, should balance operand latency and migratory cache sharing with contention for processing element instruction storage. Based on these results, we designed a new scheduling algorithm for instruction placement, that drew upon characteristics of two previous algorithms that exploited these features individually. The new algorithm outperformed the old ones by averages 28% and 7%.

We expect that our model and the methodology used to develop could apply similarly to the development of scheduling algorithms for other spatial computing fabrics, including TRIPS [34], RAW [45], SmartMemories [26], and nanoFabrics [18]. More important than how these individual architectures implement instruction placement, is the question of how to make intelligently guided decisions about the process. The model presented in this paper can serve as that guide.

9. ACKNOWLEDGMENTS

This work has been made possible through the generous support of an NSF CAREER Award (ACR-0133188), ITR grant (CCR-0325635), and doctoral fellowship (Swanson); Sloan Research Foundation Award (Oskin); Intel Fellowships (Swanson, Mercaldi); ARCS Fellowships (Putnam, Schwerin); and support from Intel and Dell.

10. REFERENCES

[1] 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.

[2] Arvind and R. Nikhil. Executing a program on the mit tagged-token dataflow architecture. *IEEE Transactions on Computers*, 39(3).

[3] V. Betz and J. Rose. VPR: A new packing, placement and routing tool for FPGA research. In W. Luk, P. Y.

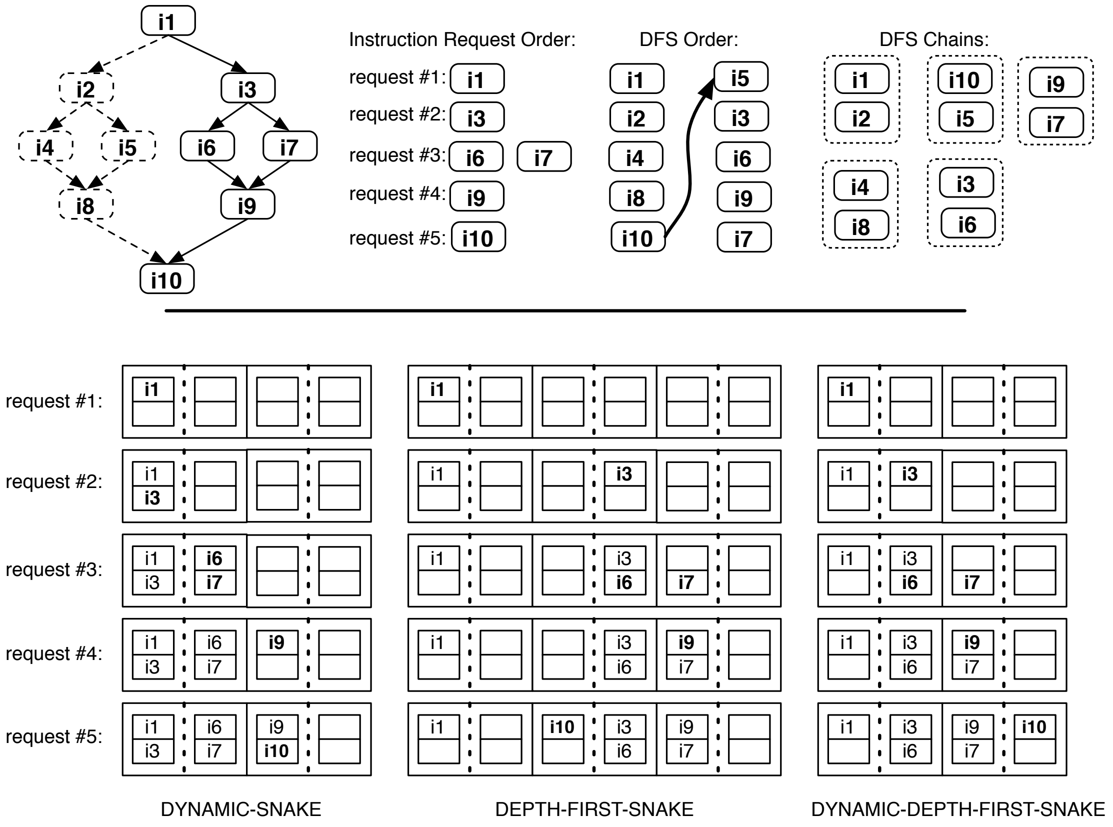


Figure 9: DYNAMIC-DEPTH-FIRST-SNAKE Example: Here is a small example that illustrates the differences between dynamic-snake,depth-first-snake and dynamic-depth-first-snake. On the top is a sample dataflow graph, in which $i1$ is a data steering instruction. Imagine that for this particular run the right path is executed. Just to the right of the graph is the dynamic order in which the instructions to be executed are loaded into the processor. The bottom part of the figure depicts instruction loads over time for the three instruction placement algorithms, dynamic-snake on the left, depth-first-snake in the middle, and dynamic-depth-first-snake on the right. The WaveScalar processor depicted has only 2 pods, and only 2 instructions fit in each PE. Therefore dynamic-snake will form chains of two instructions. For dynamic-snake, the first instruction required is $i1$, followed by a request for $i3$, which is loaded into the same PE as $i1$ since there is room. All other required instructions are loaded in a similar fashion. depth-first-snake places instructions in the same dynamic request stream, but PEs are populated according to the static instruction order. dynamic-depth-first-snake groups instructions in the same way as depth-first-snake, but the the selection of PEs is made dynamically, as in depth-first-snake. Note that the blank spaces left behind by depth-first-snake are gone in dynamic-depth-first-snake. It is this improved utilization of instruction space, coupled with depth-first-snake's deliberate use of the fast local bypass network, which allows dynamic-depth-first-snake to outperform dynamic-snake and depth-first-snake. (In this example, whose purpose is to illustrate the workings of the different instruction placement algorithms, depth-first-snake and dynamic-depth-first-snake have identical operand latency costs; however, in general, i.e., in our applications, that cost was less for dynamic-depth-first-snake.

Cheung, and M. Glesner, editors, *Field-Programmable Logic and Applications*. Springer-Verlag, Berlin.

[4] A. P. Bohm and J. Sargeant. Efficient dataflow code generation for sisal. Technical Reports UMCS-85-10-2, Department of Computer Science, University of Manchester, Oct. 1985.

[5] L. Carter, J. Ferrante, and S. F. Hummel. Hierarchical tiling for improved superscalar performance. In *IPPS '95: Proceedings of the 9th International Symposium on Parallel Processing*, 1995.

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

[7] D. E. Culler and Arvind. Resource requirements of dataflow programs. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, Honolulu, Hawaii, May 30–June 2, 1988. IEEE Computer Society TCCA and ACM SIGARCH. *Computer Architecture News,* 16(2), May 1988.

[8] 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*.

[9] D. E. Culler and G. K. Maa. Assessing the benefits of fine-grain parallelism in dataflow programs. In *Supercomputing '88: Proceedings of the 1988 ACM/IEEE conference on Supercomputing*, 1988.

[10] D. E. Culler, A. Sah, K. E. Schauser, T. von Eicken, and J. Wawrzynek. Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine. In *Proceedings of the4th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1991.

[11] 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, April 3–5, 1978. IEEE Computer Society and ACM SIGARCH.

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

[13] C. Ebeling et al. Mapping applications to the rapid configurable architecture. In *Symposium on FPGAs for Custom Computing Machines*, Napa Valley, California, April 1997.

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

[15] S. Fortune and J. Wyllie. Parallelism in random access machines. In *STOC '78: Proceedings of the tenth annual ACM symposium on Theory of computing*, 1978.

[16] N. Gloy, T. Blackwell, M. D. Smith, and B. Calder. Procedure placement using temporal ordering information. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, Research Triangle Park, North Carolina, December 1–3, 1997. IEEE Computer Society TC-MICRO and ACM SIGMICRO.

[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*, 24(1), Jan. 1991.

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

[19] 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*, 1989.

[20] J. R. Gurd, C. C. Kirkham, and I. Watson. The manchester prototype dataflow computer. *Communications of the ACM*, 28(1), 1985.

[21] C. A. R. Hoare. Communicating sequential processes.

[22] 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*, 1983.

[23] 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.

[24] C. Lin. ZPL Language Reference Manual. UW-CSE-TR 94-10-06, University of Washington, 1996.

[25] J. Lo, S. Eggers, H. Levy, and D. Tullsen. Compilation issues for a simultaneous multithreading processor, 1996.

[26] 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.

[27] R. Nikhil. *ID Version 88.1, Reference Manual*. MIT, Laboratory for Computer Science, Cambridge, MA, 90.1 edition, 1991.

[28] D. B. Noonburg and J. P. Shen. A framework for statistical modeling of superscalar processor performance.

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

[30] M. Oskin, F. T. Chong, and M. K. Farrens. HLS: combining statistical and symbolic simulation to guide microprocessor designs. In *ISCA*.

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

[32] G. M. Papadopoulos and K. R. Traub. Multithreading: A revisionist view of dataflow architectures. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, Toronto, Ontario, May 27–30, 1991. ACM SIGARCH and IEEE Computer Society TCCA. *Computer Architecture News,* 19(3), May 1991.

[33] 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*, 1989.

[34] 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.

[35] F. I. Scalability. Design and analysis of routed inter-alu networks.

[36] 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*, 1986.

[37] A. Singh, G. Parthasarathy, and M. Marek-Sadowska. Interconnect resource-aware placement for hierarchical fpgas. In *ICCAD '01: Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, 2001.

[38] SPEC. Spec CPU 2000 benchmark specifications. SPEC2000 Benchmark Release, 2000.

[39] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin. WaveScalar. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, page 291, 2003.

[40] S. Swanson, A. Putnam, M. Mercaldi, K. Michelson, A. Petersen, A. Schwerin, M. Oskin, and S. Eggers. The wavescalar architecture.

[41] S. Swanson, A. Putnam, M. Mercaldi, K. Michelson, A. Petersen, A. Schwerin, M. Oskin, and S. Eggers. Area-performance trade-offs in tiled dataflow architectures. In *Proceedings of the 33rd Annual Symposium on Computer Architecture*, 2006.

[42] 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)*, 2003.

[43] W. Thies, M. Karczmarek, and S. P. Amarasinghe. Streamit: A language for streaming applications. In *Computational Complexity*.

[44] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8), 1990.

[45] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Argarwal. Baring it all to software: Raw machines. *IEEE Computer*, 1997.