To appear in Micro-27, November 1994.

Theoretical Modeling of Superscalar Processor Performance

Derek B. Noonburg*   John P. Shen

Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213
{derekn,shen}@ece.cmu.edu

Abstract

The current trace-driven simulation approach to determine superscalar processor performance is widely used but has some shortcomings. Modern benchmarks generate extremely long traces, resulting in problems with data storage, as well as very long simulation run times. More fundamentally, simulation generally does not provide significant insight into the factors that determine performance or a characterization of their interactions. This paper proposes a theoretical model of superscalar processor performance that addresses these shortcomings. Performance is viewed as an interaction of program parallelism and machine parallelism. Both program and machine parallelisms are decomposed into multiple component functions. Methods for measuring or computing these functions are described. The functions are combined to provide a model of the interaction between program and machine parallelisms and an accurate estimate of the performance. The computed performance, based on this model, is compared to simulated performance for six benchmarks from the SPEC 92 suite on several configurations of the IBM RS/6000 instruction set architecture.

1 Introduction

Current techniques for determining the performance of new superscalar processor designs are usually based on trace-driven simulation. A benchmark is run on existing hardware to generate a trace. A trace-driven, or timing, simulator configured for the new implementation "runs" this trace and generates performance information. In order to obtain accurate performance data, this process must be repeated for many large benchmarks and extremely long traces.

This approach has some shortcomings. The traces are very long — millions of basic blocks, containing tens of millions of instructions for each benchmark — and are thus expensive to store. Simulation of these long traces is very time-consuming. In addition, generating a trace for the first implementation of a new instruction set architecture (ISA) is problematic. A more fundamental problem with simulation is that it neither characterizes nor provides insight into the factors that determine performance. The only way to determine the effect of a design change is to update the simulator configuration and resimulate the entire benchmark suite.

This paper presents a theoretical model of superscalar processor performance. The performance metric used here is instruction-level parallelism (ILP), roughly the speedup over a scalar pipeline. ILP is the result of the interaction of two components: program parallelism and machine parallelism. By examining the parameters associated with specific machines and programs, insight may be gained into the factors that determine performance.

In this theoretical model, a benchmark and a trace are analyzed only once to compute the program parallelism parameters. These parameters are essentially an abstraction of the characteristics of the trace. Each processor implementation is analyzed to compute the machine parallelism parameters. Program and machine parallelisms are then combined to provide an accurate estimate of processor performance. Once the program parallelism parameters are determined for a benchmark, its performance on an arbitrary machine, characterized by a set of machine parallelism parameters, can be computed without reexamining the trace.

Section 2 provides the background for this paper, including previous work and a brief description of our model. Section 3 describes the parallelism distributions and mathematics used by this model. Program paral-

*Supported by a National Science Foundation Graduate Fellowship.

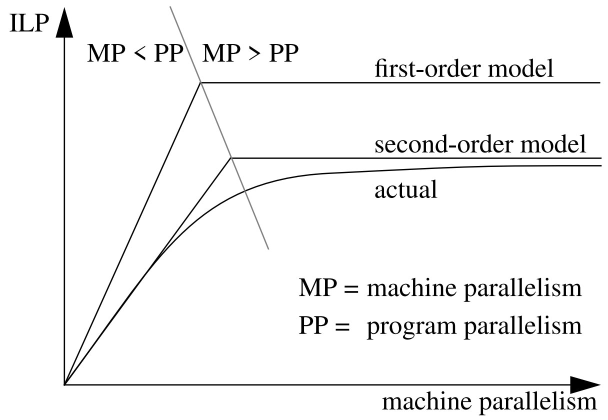


Figure 1: Jouppi's piecewise linear superscalar performance model. The upper curve is his first-order model; the middle curve is his second-order model. The inner curve is the actual performance.

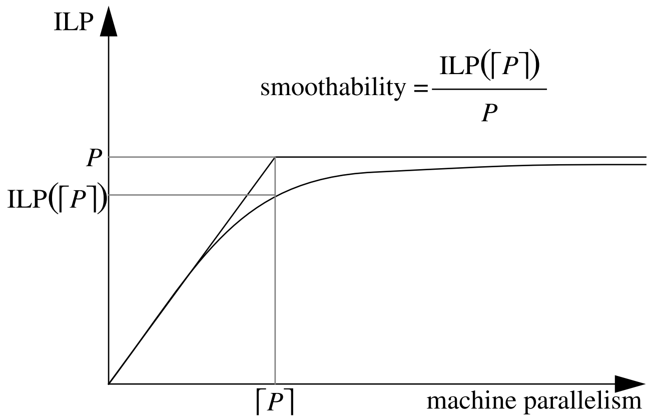


Figure 2: Theobald et al.'s smoothability function.

lelism and machine parallelism are defined in Sections 4 and 5, respectively. Section 6 presents the theoretical model proposed in this work. Finally, Section 7 concludes with remarks about future research.

2 Background

Numerous researchers have proposed methods to determine the maximum program parallelism present in code, using various ideal machine models [Jou89, Wal91, AS92, LW92, TGH92, DAF94]. Most of these are limit studies. Few have attempted to characterize the parallelism which can actually be obtained by realistic processors. Furthermore, most of these are experimental studies that attempt to obtain performance data empirically, usually by simulation. Dubey et al. have recently proposed an analytical model similar to ours [DAF94].

2.1 The piecewise linear model

Jouppi describes a simple theoretical performance model which uses the concepts of machine parallelism and benchmark (i.e., program) parallelism [Jou89]. Machine parallelism is defined as the product of the average degree of superpipelining [Jou89] and the degree of parallel issue. This is effectively the maximum number of instructions "in flight" (in the execution stages) in the processor. Program parallelism is defined as the average speedup when the program is executed on an infinitely parallel superscalar processor, compared to execution on a single-issue processor.

The performance vs. machine parallelism curve is divided into two linear regions. In the first region, the machine parallelism is less than the program parallelism, and the performance is limited by machine parallelism. In this region, performance improves with increased machine parallelism. In the second region, the machine parallelism is greater than the program parallelism, and the performance is limited by the program parallelism. In this region, performance does not change as the machine parallelism is increased. See Figure 1.

Jouppi describes several factors which account for the differences between the modeled and actual behaviors. These include variations in instruction latency, variations in per-cycle instruction parallelism, and variations in parallelism by instruction class. The suggested second-order model [Jou89] takes these nonuniformities into account to explain the lowering of the two performance asymptotes, i.e., the tilting of the machine parallelism-limited asymptote and the dropping of the program parallelism-limited asymptote (see Figure 1). This model does not, however, provide an explanation for the rounding of the knee of the actual performance curve. Effectively, Jouppi's piecewise linear model focuses on the asymptotic behavior of processor performance and not on the interaction of program and machine parallelism in the "transition" region.

2.2 Smoothability

Theobald et al. introduce the concept of smoothability of program parallelism [TGH92] to explain this rounded knee in the transition region. Let P be the average program parallelism of a benchmark. If the program parallelism is perfectly smooth, i.e., evenly distributed over time, then a machine parallelism of $\lceil P \rceil$ is sufficient to achieve maximum performance. In general, program parallelism is not perfectly smooth: more than P instructions are available in some cycles, less than P in others. Theobald et al. define smoothability as the ratio of the performance with a machine parallelism of $\lceil P \rceil$ to the performance with infinite machine parallelism. A perfectly smoothable program would thus have a smoothability of one.

Graphically, smoothability is then a measure of the distance between the "corner" of Jouppi's second-order piecewise linear model and the knee of the actual performance curve (see Figure 2). This distance reflects the inefficiency in the interaction of program parallelism and machine parallelism due to dynamic mismatching of their nonuniformities.

2.3 An ILP model

In our proposed model, ILP is viewed as a result of the interaction of program parallelism and machine parallelism (see Figure 3). When program parallelism is sufficiently high, overall performance is determined almost entirely by machine parallelism. Similarly, if machine parallelism is high, performance is determined by program parallelism. Our model focuses on the region where the two asymptotic regions meet. This region is important because it contains the optimal operating point.

This model is an extension of the work described above. The idea of separately measuring program and machine parallelism is adapted from Jouppi's model [Jou89]. In addition, we provide a method (represented by the "INTERACTION" box in Figure 3) for combining program and machine parallelism to model ILP in the transition region. Theobald et al.'s smoothability experiments [TGH92] indicate that a single average parallelism number is not sufficient to model performance. Our model uses parallelism distributions to account for uneven, or nonuniform, distribution of parallelism. Furthermore, our approach provides an analytical model that facilitates computation of the performance of an arbitrary program on an arbitrary machine, once the component parameters are determined.

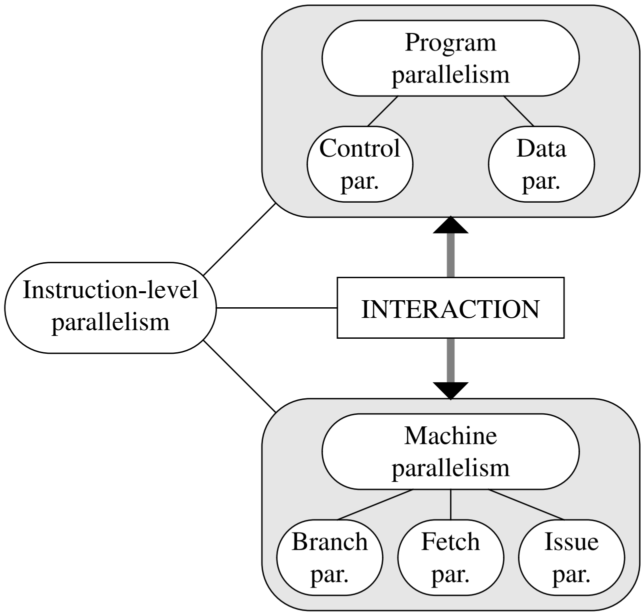


Figure 3: Performance model hierarchy. Performance (ILP) is a result of the interaction of program parallelism and machine parallelism, each of which is broken into components.

3 Parallelism Distributions

In order to account for non-uniform distribution of parallelism, this work uses parallelism distributions rather than single average parallelism values.

Definition: A *parallelism distribution* is represented by a vector (indicated by a boldface variable). This is simply a probability distribution. For example,

$$\mathbf{P} = [\ \mathbf{P}_0 \quad \mathbf{P}_1 \quad \mathbf{P}_2 \quad \ldots\ ]$$

means that a fraction, $\mathbf{P}_i$, of the cycles exhibit parallelism of degree i. Note that $\sum_{i=0}^{\infty} \mathbf{P}_i = 1$. The *average parallelism* is just the weighted average of the parallelism distribution:

$$\langle P \rangle = \sum_{i=0}^{\infty} i\mathbf{P}_i$$

Definition: A *limited distribution* is a simplified model of the effect of constraining a parallelism to stay below some fixed limit. The distribution of $\mathbf{P}$ limited to m is written $\mathbf{P}|_m$. The i^{th} element of this new distribution is:

$$(\mathbf{P}|_m)_i = \begin{cases} \mathbf{P}_i & i < m \\ \sum_{k=m}^{\infty} \mathbf{P}_k & i = m \\ 0 & i > m \end{cases}$$

As an example, consider the parallelism distribution

$$\mathbf{P} = [\ 0.00 \quad 0.05 \quad 0.10 \quad 0.20 \quad 0.50 \quad 0.15 \quad 0.00 \quad \ldots\ ]$$

If this parallelism is limited to a maximum of degree three, the result is

$$\mathbf{P}|_3 = [\ 0.00 \quad 0.05 \quad 0.10 \quad 0.85 \quad 0.00 \quad \ldots\ ]$$

The last non-zero element is the sum of all of the elements of the old distribution which equal or exceed the limit ($0.85 = 0.20 + 0.50 + 0.15$).

Definition: A *parallelism function* specifies the parallelism that can be achieved by a particular stage, as a function of the parallelism available from the previous stage(s). If average parallelism functions and average parallelism values are used (instead of distribution functions and distributions), function application is straightforward. For example, consider an average parallelism function Y. If the average number of instructions available to the stage is $\langle\rangle$, then the average parallelism is just $\langle Y\rangle = Y(\langle\rangle)$. If $\langle\rangle$ is not an integer, linear interpolation on the Y function can be used.

Definition: A *parallelism distribution function* is represented by a matrix (indicated by a hat accent). The ij element of $\widehat{\mathbf{Y}}$, $\widehat{\mathbf{Y}}_{ij}$, is the probability that the output parallelism (the result of applying the parallelism function) is i, given that the input parallelism (the function argument) is j. Thus, column j of $\widehat{\mathbf{Y}}$, denoted $\widehat{\mathbf{Y}}(j)$, is the parallelism distribution which would result if the input parallelism were always j. To apply a distribution function, $\widehat{\mathbf{Y}}$, to an input distribution, $\mathbf{X}$, each column of $\widehat{\mathbf{Y}}$ is weighted by the associated value in $\mathbf{X}$. This is just a matrix-vector multiplication:

$$\mathbf{Y} = \mathbf{X}_0\widehat{\mathbf{Y}}(0) + \mathbf{X}_1\widehat{\mathbf{Y}}(1) + \mathbf{X}_2\widehat{\mathbf{Y}}(2) + \cdots = \widehat{\mathbf{Y}} \times \mathbf{X}$$

Because each of the columns of the matrix is a valid distribution, the result of such a multiplication is also a valid distribution, i.e., its elements sum to one. (Note: vectors are treated as column vectors, though they are written as row vectors to save space.)

Consider a sample parallelism distribution function as follows:

$$\widehat{\mathbf{Y}} = \begin{bmatrix} 1.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & 1.00 & 0.40 & 0.20 \\ 0.00 & 0.00 & 0.60 & 0.60 \\ 0.00 & 0.00 & 0.00 & 0.20 \end{bmatrix}$$

If two instructions are available, the output parallelism distribution would be

$$\widehat{\mathbf{Y}}(2) = [\ 0.00 \quad 0.40 \quad 0.60 \quad 0.00\ ]$$

If the distribution of available instructions is

$$\mathbf{X} = [\ 0.10 \quad 0.20 \quad 0.60 \quad 0.10\ ]$$

then the output parallelism distribution is

$$\mathbf{Y} = \widehat{\mathbf{Y}} \times \mathbf{X} = [\ 0.10 \quad 0.46 \quad 0.42 \quad 0.02\ ]$$

Note the distinction between a parallelism distribution function (matrix), and a parallelism distribution (vector), which is the result of applying (matrix×vector) the function to some input parallelism distribution (vector).

4 Program Parallelism

Program parallelism is the degree to which instructions can be executed in parallel, given the constraints in the object code. This includes effects due to the source language, compiler, and ISA, and properties inherent to the program. The constraints are due to program dependences which can take two forms: control dependences and data dependences [HP90]. Hence, program parallelism is correspondingly decomposed into *control parallelism* and *data parallelism* in our model (see Figure 3).

4.1 Components

Control parallelism is a measure of the number of *useful* instructions which are visible to the processor's instruction-fetch mechanism. Control parallelism is limited by the control dependences caused by branch instructions. Instructions immediately following a branch instruction cannot be executed until the outcome of the branch is known. Starting at the current fetch address, all instructions up to and including the next branch instruction are visible, i.e., can be fetched if there is enough fetch bandwidth. All instructions after that branch must wait for it to be resolved. (These instructions may be fetched, but they are not considered useful because it is not yet known whether they will be executed.) A processor can, however, extract control parallelism from beyond branches by using speculation, such as branch prediction [RF93]. With branch prediction, potentially useful instructions can be fetched from beyond branches.

The *control parallelism function*, $\widehat{\mathbf{P}}_C$, of a program describes the number of visible useful instructions. $\widehat{\mathbf{P}}_C$ is a function of the number of basic blocks visible to the processor (which is determined by the machine parallelism). That is, $(\widehat{\mathbf{P}}_C)_{ij}$ is the probability that i useful instructions are visible, given that j basic blocks are visible. (It is possible for zero useful basic blocks to be visible, if a branch is mispredicted and the processor wastes cycles fetching incorrect instructions; in this case there are clearly zero visible instructions.) The entire $\widehat{\mathbf{P}}_C$ matrix (of some finite size) is computed from the trace so that the trace does not have to be reexamined for every change in the machine parallelism parameters.

Data parallelism is a measure of the number of instructions in a processor which are ready to be executed. Data parallelism is limited by the data dependences caused by register and memory references. Instructions which use a value (register or memory word) must wait until the instruction which generates that value has executed. An instruction can be executed only if it is independent of all other instructions which are still executing. If the processor does not implement out-of-order execution, all previous instructions must also be ready to execute.

The *data parallelism function*, $\widehat{\mathbf{P}}_D$, of a program describes the number of available data-ready instructions.

$\widehat{\mathbf{P}}_D$ is a function of the number of instructions which are available to be examined and issued. $\widehat{\mathbf{P}}_D$ is also dependent on the processor's issue policy: in-order or out-of-order. (The number of available instructions and the issue policy are determined by the machine parallelism.) $(\widehat{\mathbf{P}}_C)_{ij}$ is the probability that i instructions are data-ready, given that j instructions are available to be examined. Two complete $\widehat{\mathbf{P}}_D$ matrices are computed, one for each issue policy (in-order and out-of-order), in order to avoid reexamining the trace.

4.2 Approach

Program parallelism measurements are taken from compiled benchmark code and dynamic traces of that code. This means that program parallelism is a function not only of the benchmark, but also of the ISA, compiler, and typical data set. A more complex model might treat these factors separately. Our model assumes that the object code and the trace have captured their contributions. This subsection defines the program parallelism functions, $\widehat{\mathbf{P}}_C$ and $\widehat{\mathbf{P}}_D$, used in our performance model, and explains how they are obtained.

To generate the control parallelism function, $\widehat{\mathbf{P}}_C$, we first generate a preliminary matrix, $\widehat{\mathbf{P}}'_C$. Assume that j basic blocks are visible. Each instruction in the trace, in turn, is considered as the starting point, i.e., the first instruction to be fetched. The next $j-1$ branches are assumed to be speculated: all instructions in the window starting with this instruction and ending at the j^{th} branch are considered useful. The distribution of window sizes formed by computing the windows for all instructions is $\widehat{\mathbf{P}}'_C(j)$.

This process assumes that each instruction in the trace is equally likely to be a starting point for fetching during any given cycle. However, the processor spends less time in areas with more parallelism, and hence, instructions in these areas are less likely to be fetch starting points. To account for this, the k^{th} element of each distribution (column) in $\widehat{\mathbf{P}}'_C$ is divided by k, and the distribution is rescaled so that it sums to one:

$$\left(\widehat{\mathbf{P}}_C\right)_{ij} = \frac{\left(\widehat{\mathbf{P}}'_C\right)_{ij}/i}{\sum_{k=1}^{\infty}\left(\widehat{\mathbf{P}}'_C\right)_{kj}/k}$$

As an example, consider the short instruction trace in Table 1. The value in the $P_C(1)$ column for each instruction in the table is the number of useful instructions following (and including) that instruction, assuming one visible basic block (zero speculated branches). The distribution of these numbers forms the first column of $\widehat{\mathbf{P}}'_C$:

$$\begin{aligned}\widehat{\mathbf{P}}'_C(1) &= [\ 0/18 \quad 6/18 \quad 6/18 \quad 4/18 \quad 2/18\ ] \\ &= [\ 0.00 \quad 0.33 \quad 0.33 \quad 0.22 \quad 0.11\ ]\end{aligned}$$

cycle	instruction	P_C (1)	P_D (1)	(2)	(3)	(4)
0	`ld a[0]`	3	1	2	2	2
1	`ld b[0]`	2	1	1	1	1
2	`bc a[0] > b[0]` ; taken	1	1	2	3	4
3	`st a[0], c[0]`	4	1	2	3	3
4	`ld a[1]`	3	1	2	2	2
5	`ld b[1]`	2	1	1	1	1
6	`bc a[1] > b[1]` ; not taken	1	1	2	3	4
7	`st b[1], c[1]`	2	1	2	3	4
8	`br`	1	1	2	3	3
9	`ld a[2]`	3	1	2	2	2
10	`ld b[2]`	2	1	1	1	1
11	`bc a[2] > b[2]` ; taken	1	1	2	3	4
12	`st a[2], c[2]`	4	1	2	3	3
13	`ld a[3]`	3	1	2	2	2
14	`ld a[4]`	2	1	1	1	1
15	`bc a[3] > b[3]` ; not taken	1	1	2	3	-
16	`st b[3], c[3]`	2	1	2	-	-
17	`br`	1	1	-	-	-

Table 1: A short dynamic instruction trace for a loop which computes `c[i] = max(a[i], b[i])`.

The first column of $\widehat{\mathbf{P}}_C$ can then be computed:

$$\begin{aligned}\widehat{\mathbf{P}}_C(1) &= [\ 0.00 \quad 0.33/1 \quad 0.33/2 \\ &\qquad\qquad 0.22/3 \quad 0.11/4\ ]/0.60 \\ &= [\ 0.00 \quad 0.55 \quad 0.28 \quad 0.12 \quad 0.05\ ]\end{aligned}$$

The data parallelism function, $\widehat{\mathbf{P}}_D$ is also computed by scanning the trace. Assume that a window of j instructions is available to be issued. Each instruction in the trace, in turn, is considered as the starting point; this instruction is independent by definition (this assumes single-cycle result latencies). Each of the next $j-1$ instructions is examined. For out-of-order execution, an instruction must be independent of all preceding instructions in the window to be marked independent. For in-order execution, the instruction must be independent, and all preceding instructions in the window must also have been marked independent. The distribution of the number of independent instructions forms $\widehat{\mathbf{P}}_D(j)$, the j^{th} column of $\widehat{\mathbf{P}}_D$. Only true, i.e., read-after-write, data dependences are considered. This is effectively the same as assuming perfect renaming. This technique only accounts for register dependences; memory dependences are not considered in this paper. As with control parallelism, each instruction in the trace is considered, and a distribution is constructed.

The raw data parallelism data for the example trace is given in the P_D columns of Table 1. The value in the $P_D(j)$ column is the number of independent instructions in a window of length j, starting at that instruction.

The distribution of these numbers is thus $\widehat{\mathbf{P}}_D(j)$.

$$\widehat{\mathbf{P}}_D = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 18/18 & 4/17 & 4/16 & 4/15 \\ 0 & 0/18 & 13/17 & 4/16 & 4/15 \\ 0 & 0/18 & 0/17 & 8/16 & 3/15 \\ 0 & 0/18 & 0/17 & 0/16 & 4/15 \end{bmatrix}$$
$$= \begin{bmatrix} 1.00 & 0.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & 1.00 & 0.24 & 0.25 & 0.27 \\ 0.00 & 0.00 & 0.76 & 0.25 & 0.27 \\ 0.00 & 0.00 & 0.00 & 0.50 & 0.20 \\ 0.00 & 0.00 & 0.00 & 0.00 & 0.27 \end{bmatrix}$$

4.3 Experimental data

Given the foregoing definitions of control parallelism and data parallelism, we now apply them to a number of real benchmarks. These benchmarks are used throughout this paper for illustrating our model. Six representative benchmarks are chosen:

- compress: file compression utility (SPECint 92);
- eqntott: truth table generator (SPECint 92);
- espresso: boolean minimization (SPECint 92);
- li: Lisp interpreter (SPECint 92);
- hydro2d: astrophysics mathematics (SPECfp 92);
- ora: optical system ray tracer (SPECfp 92);

For the SPEC benchmarks, smaller data sets are used in place of the official reference sets in order to decrease the trace length. The traces range from 0.3 to 3.6 million instructions. The IBM RS/6000 ISA is used for all experiments presented here. All data are generated by software tools which have been implemented for trace generation and analysis, and parallelism parameter extraction.

Figure 4 shows the average control parallelism, $\langle P_C \rangle$, as a function of the number of visible basic blocks. (The average values $\langle P_C \rangle$ and $\langle P_D \rangle$ are easier to show in a graph; distributions are used in the actual model computations.) The control parallelism function is essentially linear, with a slope close to the average dynamic basic block size. That is, speculating one branch doubles the number of instructions available to be fetched. Speculating two branches triples the number, and so on.

Figure 5 shows the average data parallelism $\langle P_D \rangle$, as a function of the number of instructions in the issue window, for the in-order and out-of-order execution models. As the number of examined instructions increases, the number which are independent also increases, up to a limit. This limit, of about three instructions per cycle for the in-order model, can be considered a measure of the intrinsic data parallelism of the benchmark. The in-order and out-of-order curves have the same general shape. However, the intrinsic limit is significantly higher for the out-of-order model.

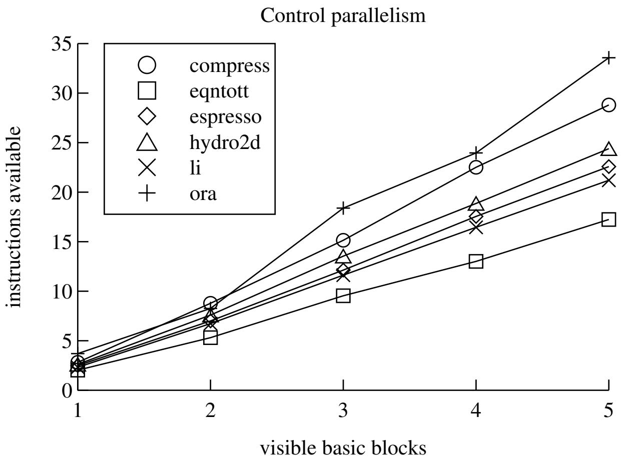


Figure 4: Average control parallelism ($\langle P_C \rangle$) as a function of the number of visible basic blocks.

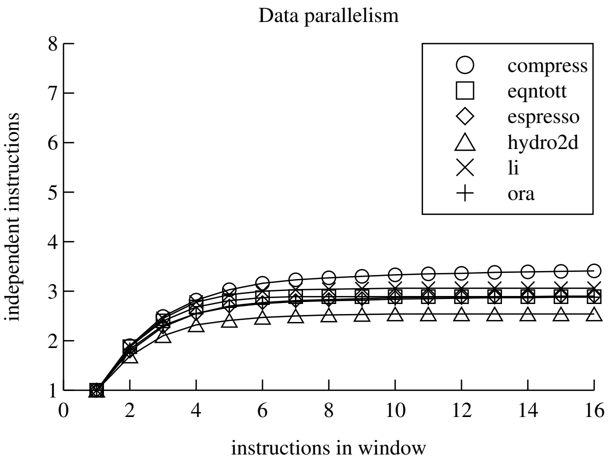
It can be seen from this data that both speculative execution and out-of-order execution are important to achieving high levels of ILP. Speculative execution allows the processor to fetch more instructions. Out-of-order execution (or, equivalently, good instruction scheduling) allows more instructions to be issued.

5 Machine Parallelism

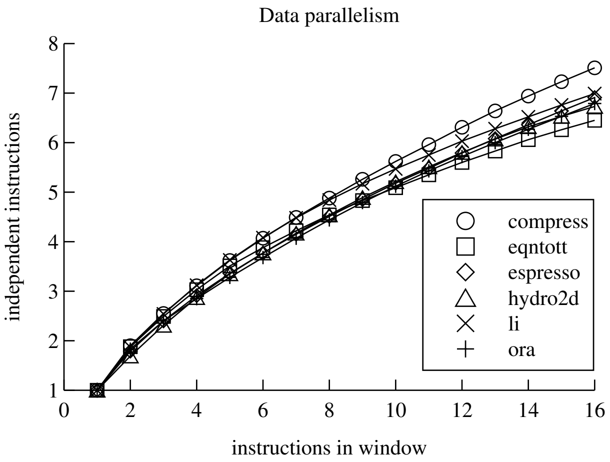
Machine parallelism is the degree to which instructions can be executed simultaneously by the processor, given its inherent implementation constraints. Machine parallelism is the overall capacity for processing instructions. For a scalar pipeline with all the pipeline stages operating in lockstep and no additional buffering other than the single buffers between consecutive stages, a single parameter can characterize this capacity. For superscalar processors, there are constraints that limit the capacity at various intermediate points of instruction processing. In our model, these constraints are divided into three categories — branch constraints, fetch/decode constraints, and issue constraints — to account for potential structural dependences [HP90] in various portions of the machine. Hence, machine parallelism is decomposed into *branch parallelism*, *fetch parallelism*, and *issue parallelism* (see Figure 3).

5.1 Components

Instruction flow through the control path of a superscalar processor is limited by branch instructions. In-



(a)



(b)

Figure 5: Average data parallelism ($\langle P_D \rangle$) as a function of the number of instructions in the issue window, for the (a) in-order and (b) out-of-order execution models.

structions after a branch cannot be fetched until the branch target is known. Techniques such as branch prediction and speculative execution are used to overcome this limitation.

The *branch parallelism*, $\mathbf{M}_B$, of a processor is the distribution of the number of basic blocks which are visible to the fetching mechanism. $\mathbf{M}_B$ is the first stage of the model, so it is a distribution (vector), not a distribution function (matrix). If the processor does not perform speculation and has no branch penalties, the branch parallelism is always one, i.e.,

$$\mathbf{M}_B = [\ 0 \quad 1 \quad 0 \quad 0 \quad \ldots\ ]$$

If the processor speculates past branches, the branch parallelism will sometimes be two (when a branch is predicted correctly), sometimes one (when a branch is mispredicted), and sometimes zero (when the processor mispredicts and fetches useless instructions — this accounts for the misprediction penalty). Note that $\mathbf{M}_B$ is actually a function of certain program parallelism parameters, e.g., the distribution of branch types, although this dependence is not made explicit here.

The data path of a superscalar processor is limited by two things: the fetch/decode rate and the issue rate. Superscalar processors often have buffers between the fetch and issue stages (e.g., an issue buffer), and between the issue and execute stages (e.g., reservation stations). This separates the various stages, allowing some slack between them. Our model accounts for this by separately quantifying the fetch parallelism and issue parallelism.

The *fetch parallelism function*, $\widehat{\mathbf{M}}_F$, describes the number of instructions actually fetched and decoded by the processor per cycle. $\widehat{\mathbf{M}}_F$ is a function of the number of useful instructions which are visible to the fetch unit, i.e., the control parallelism. The fetch parallelism function is determined by the size of the issue buffer, or the number of issue slots, or something equivalent.

The *issue parallelism function*, $\widehat{\mathbf{M}}_I$, describes the number of instructions actually issued to functional units (or reservation stations) per cycle. $\widehat{\mathbf{M}}_I$ is a function of the number (and types) of instructions in the issue buffer, i.e., the data parallelism. The issue parallelism function is determined by the number and types of functional units, their issue latency, and the number of reservation stations associated with each.

5.2 Approach

This work uses a fairly simple model of a superscalar processor to determine machine parallelism. This model is only a starting point. With the increased complexity expected in forthcoming superscalar processors, this model will need to be enhanced. This subsection defines the specific machine parallelism functions, $\mathbf{M}_B$, $\widehat{\mathbf{M}}_F$, and $\widehat{\mathbf{M}}_I$, used in our model, and explains how they are computed.

The RS/6000 uses a simple branch prediction scheme and speculatively issues instructions along the predicted path. On average, the effects of incorrectly predicted conditional branches balance the effects of correctly predicted conditional branches. However, branches based on the count register for loop control can always be predicted correctly. To account for this, the branch parallelism distribution used for the RS/6000 is

$$\mathbf{M}_B = [\ 0 \quad 1-c \quad c \quad 0 \quad \ldots\ ]$$

where c is the fraction of branches which are count register-based. $(\mathbf{M}_B)_2 = c$ means that the processor can always speculate past a count register branch. $(\mathbf{M}_B)_1 = 1-c$ means that the processor can never speculate past other branches. $(\mathbf{M}_B)_0 = 0$ means that there is no penalty for mispredicted branches, i.e., there are no cycles in which the processor is fetching useless instructions. For future processors which have more complex branch prediction and speculation hardware, $\mathbf{M}_B$ will have to be a measure of the average number of successful speculations.

It is assumed that the processor can fetch all useful instructions, up to its fetch bandwidth. If the processor has the capability to fetch b instructions, and i instructions are available, it will actually fetch $\min\{b, i\}$ instructions. The distribution of the number of available instructions is the control parallelism, $\mathbf{P}_C$, so the fetch parallelism is just a limited distribution of $\mathbf{P}_C$, i.e.,

$$\mathbf{M}_F = \mathbf{P}_C|_b$$

The distribution limiting operation can be represented as a matrix, $\widehat{\mathbf{M}}_F$, so that $\mathbf{M}_F = \widehat{\mathbf{M}}_F \times \mathbf{P}_C$:

$$\widehat{\mathbf{M}}_F = \begin{bmatrix} 1 & & 0 & \cdots & 0 \\ & \ddots & & & \\ 0 & & 1 & \cdots & 1 \\ 0 & \cdots & 0 & \cdots & 0 \\ \vdots & & \vdots & & \vdots \\ 0 & \cdots & 0 & \cdots & 0 \end{bmatrix}$$

The issue parallelism function, $\widehat{\mathbf{M}}_I$ is somewhat more complicated. $\widehat{\mathbf{M}}_I$ is a function of the instruction mix, as well as the data parallelism. This function is determined by the processor's issue policy and the number, mix, and issue latency of the processor's functional units.

The concept of *effective functional units* is used to model the dependence of the issue parallelism function on the instruction type mix. Instructions are classified into N_{instr} types, each with an issue latency, l_i. A fraction, t_i, of the dynamically executed instructions are of type i. Similarly, there are N_{FU} FU types. Each FU type k can execute some subset, S_k, of instruction types. The fraction of time spent by FUs of type k issuing instructions of type i ($i \in S_k$) is then:

$$\frac{t_i l_i}{\sum_{j \in S_k} t_j l_j}$$

The number of physical FUs of type k is f_k. The number of effective FUs is really an estimate of the maximum issue rate for that instruction type. If a particular FU type issued only instructions of type i, with issue latency, l_i, there would be $e_i = f_k / l_i$ effective FUs of type i. In general, each FU type can execute multiple instruction types with different issue latencies. To account for this, the f_k FUs are distributed across all instruction types $i \in S_k$, weighted by the fraction of time spent issuing instructions of type i. The number of effective FUs of type i is thus:

$$e_i(t_i) = \frac{t_i l_i}{\sum_{j \in S_k} t_j l_j} \cdot \frac{f_k}{l_i}$$

The issue parallelism function is formulated by assuming that a set of n independent instructions (i.e., the data parallelism is n) contains nt_i instructions of type i. The processor issues $\min\{nt_i, e_i\}$ instructions of type i. The issue parallelism is the sum of the issue rates for all of the instruction types.

Consider an example processor and program. There are five instruction types ($N_{instr} = 5$): int-mul, int-other, fp-mul, fp-div, and fp-other; two FU types ($N_{FU} = 2$): int and fp; and two FUs of each type ($f_{int} = f_{fp} = 2$). Table 2 shows an example instruction type distribution (t_i) and issue latencies (l_i) followed by all of the computations described above. The first five columns show the effective functional unit calculation; the last two show the issue rate calculation, assuming a data parallelism of two ($n = 2$). In this example, the issue rate is limited by program parallelism ($e_i > nt_i$) for integer instructions and machine parallelism ($e_i < nt_i$) for floating point instructions.

This technique produces a single average value, rather than a parallelism distribution. A distribution which is spread between the two closest integers is then constructed. For example, if the computed average issue parallelism for some window size is 2.4, the distribution is spread across 2 and 3. Using linear interpolation, such that the two factors sum to one to form a valid distribution, we get $2.4 = 0.6(2) + 0.4(3)$. The resulting distribution is thus

$$[\ 0 \quad 0 \quad 0.6 \quad 0.4 \quad 0 \quad \ldots\ ]$$

5.3 Experimental data

In this paper, branch parallelism is set as described above for the RS/6000. The baseline RS/6000 has a fetch bandwidth of four, and has four FUs: integer, floating point, branch, and condition register. Machine parallelism is varied by changing only the numbers of integer and floating point FUs. Fetch bandwidth is set to the total number of FUs: (integer) + (floating point) + (1 branch) + (1 condition register). The average issue parallelism data, broken down by instruction type, is presented in Table 3.

i	t_i	l_i	$\frac{t_i l_i}{\sum_{j \in S_k} t_j l_j}$	e_i	nt_i	$\min\{nt_i, e_i\}$
int mul	0.05	5	$0.25/0.70 = 0.36$	$0.36(2/5) = 0.14$	0.10	0.10
int other	0.45	1	$0.45/0.70 = 0.64$	$0.64(2/1) = 1.29$	0.90	0.90
fp mul	0.15	2	$0.30/1.60 = 0.19$	$0.19(2/2) = 0.19$	0.30	0.19
fp div	0.05	20	$1.00/1.60 = 0.63$	$0.63(2/20) = 0.06$	0.10	0.06
fp other	0.30	1	$0.30/1.60 = 0.19$	$0.19(2/1) = 0.38$	0.60	0.38
issue rate ($\langle M_I \rangle$)						1.63

Table 2: An example of issue parallelism computation.

instr type	compress	eqntott	espresso	li	hydro2d	ora
int mul	0.00	0.00	0.00	0.00	0.00	0.00
int div	0.00	0.00	0.00	0.00	0.00	0.00
int other	0.51	0.45	0.56	0.43	0.13	0.26
fp mul	0.00	0.00	0.00	0.00	0.14	0.33
fp div	0.00	0.00	0.00	0.00	0.02	0.02
fp other	0.00	0.00	0.00	0.00	0.25	0.20
load	0.27	0.31	0.24	0.27	0.43	0.24
store	0.15	0.11	0.08	0.17	0.14	0.17
branch	0.28	0.48	0.32	0.35	0.33	0.21
branch unit	0.00	0.01	0.02	0.02	0.01	0.02
trap	0.00	0.00	0.00	0.00	0.00	0.00
$\langle M_I \rangle$	1.20	1.36	1.22	1.25	1.45	1.44

Table 3: Average issue parallelism, broken down by instruction type, for the baseline RS/6000 with one integer and one floating point FU, and a fetch bandwidth of four.

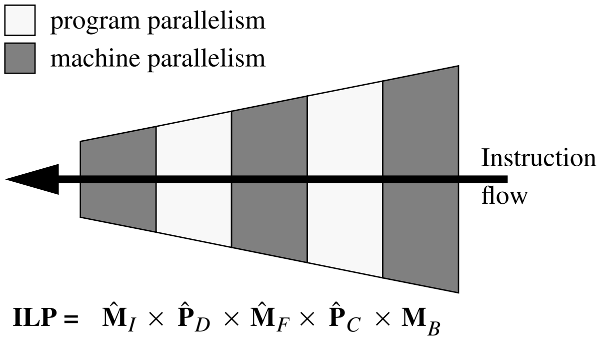


Figure 6: Modeled instruction flow or processor throughput. The boxes represent the restrictions imposed by each stage of the model. Instructions flow to the left; each stage decreases the available parallelism or potential throughput.

6 The Model

This section describes the performance model in detail. The ILP, measured here as average instructions per cycle (IPC), is determined by a composition of the five parallelism distribution functions ($\widehat{\mathbf{P}}_C$, $\widehat{\mathbf{P}}_D$, $\mathbf{M}_B$, $\widehat{\mathbf{M}}_F$, and $\widehat{\mathbf{M}}_I$) described in the previous two sections. Conceptually, instructions flow through the five stages of the model (see Figure 6). At each stage, the rate of instruction flow, or parallelism, is further restricted. The parallelism functions represent estimates of these restrictions. The result is an estimate of the IPC achieved by the processor on the benchmark, or the effective overall throughput of the processor.

6.1 Model definition

As described in Section 3, a distinction is made between a parallelism function and the corresponding parallelism. Program parallelism functions are *determined* by the program and are *functions* of machine parallelism. For example, consider the control parallelism function, graphed in Figure 4. This function is determined purely by the program — specifically, the slope of the line is dependent mainly on basic block size and also on other program characteristics. The control parallelism function is a function of branch parallelism, i.e., the number of speculated branches, represented by the x axis in the graph. To obtain the control parallelism (a value, not a function), the control parallelism function is *evaluated* at a specific branch parallelism. Similarly, machine parallelism functions are *determined* by the machine, and are *functions* of the program characteristics. The program and machine parallelism functions are determined independently; the overall performance is a result of the interactions between these functions.

The branch parallelism distribution is $\mathbf{M}_B$. For this work, it is assumed (see Section 5.2) that

$$\mathbf{M}_B = [\ 0 \quad 1-c \quad c \quad 0 \quad \ldots\ ]$$

Each of the four distribution functions is then applied in turn (see Figure 6):

$$\begin{aligned}
\mathbf{P}_C &= \widehat{\mathbf{P}}_C \times \mathbf{M}_B \\
\mathbf{M}_F &= \widehat{\mathbf{M}}_F \times \mathbf{P}_C = \widehat{\mathbf{M}}_F \times \widehat{\mathbf{P}}_C \times \mathbf{M}_B \\
\mathbf{P}_D &= \widehat{\mathbf{P}}_D \times \mathbf{M}_F = \widehat{\mathbf{P}}_D \times \widehat{\mathbf{M}}_F \times \widehat{\mathbf{P}}_C \times \mathbf{M}_B \\
\mathbf{M}_I &= \widehat{\mathbf{M}}_I \times \mathbf{P}_D \\
&= \widehat{\mathbf{M}}_I \times \widehat{\mathbf{P}}_D \times \widehat{\mathbf{M}}_F \times \widehat{\mathbf{P}}_C \times \mathbf{M}_B
\end{aligned}$$

The overall average parallelism, or performance, is the weighted average of the issue parallelism:

$$IPC = \langle M_I \rangle = \sum_{k=0}^{\infty} k\,(\mathbf{M}_I)_k$$

6.2 Experimental data

Figure 7 compares modeled performance to simulated performance. The benchmarks are simulated with the Explorer trace-driven RS/6000 simulator, which is part of the Visualization-based Microarchitecture Workbench [DSP93]. This simulator is easily retargetable and is cycle-by-cycle accurate; the baseline RS/6000 simulator has been validated with the real system.

The error in the modeled performance ranges from −0.6% to +22%. In general, the error is smaller for wider processors because the performance of these configurations is more dependent on the program parallelism, which is more accurately modeled than the machine parallelism.

The model consistently overestimates performance. This is due to the idealistic assumptions made to simplify the model. These assumptions include perfect register renaming (including the RS/6000's single MQ register, which is a serious bottleneck for numerical benchmarks), single-cycle execution latencies, and ignoring memory dependences. The branch parallelism computation used here is overly simplified: a more accurate version would take into account the prediction of conditional branches, and misprediction penalties.

7 Conclusions and Future Directions

The work presented here demonstrates that a relatively simple model of superscalar performance can achieve reasonably accurate results. Improvements to the model will make it both more accurate and more useful. There are straightforward extensions to our current model that will yield such improvements.

In addition to the RS/6000, this model has been applied to the DEC Alpha AXP, with promising results. However, to achieve acceptable accuracy, the effects of data dependence latencies must be taken into account. There are two aspects to this. First, the machine parallelism functions must be extended to handle multi-cycle result latencies. The current model simply assumes that a result is ready one cycle after issue. Second, the distribution of inter-instruction dependence lengths in the program code must be incorporated into the program parallelism model. Large result latencies have more effect when reads closely follow the corresponding writes; conversely, a compiler can reduce the effects of long result latencies by scheduling instructions between such writes and reads. These issues are especially important for deeply pipelined processors such as the Alpha AXP 21064.

There are several other potential improvements which will make the model more detailed and thus more accurate. The branch parallelism should be made an explicit function of the branching characteristics of the code, i.e., branch type distribution and predictability. The effect of memory dependences should also be considered. Like register dependences, this also has effects on both program and machine parallelisms. A theoretical cache model could be included to represent real memory systems.

Another set of improvements will make the machine parallelism model more accurate, and possibly more useful for experimenting with new processors. Instead of using *ad hoc* computations (e.g., effective functional units) to *estimate* machine parallelism, a set of synthetic benchmarks could be used to *measure* it. The idea is to generate a set of benchmarks which would form a sort of basis, covering all aspects of program parallelism. These benchmarks would be run on the processor or simulator, and their performance used to determine the machine parallelism parameters.

This paper represents only an initial effort toward developing a theoretical model of superscalar processor performance. The ultimate goal is an analytical model that is accurate enough to render the current form of timing simulation unnecessary. It appears that once the modeling parameters are determined, the computation time for determining the resultant performance is negligible compared to the simulation time. There is plenty of room for improving the theoretical model accuracy by increasing its complexity. The current form of trace-driven simulation is analogous to determining the behavior of a circuit by simulating charge movement across doped regions and PN junctions. Our goal is to develop an analytical model for microarchitectures comparable to the SPICE model for a circuit.

References

[AS92] Todd M. Austin and Gurindar S. Sohi. Dynamic Dependency Analysis of Ordinary Programs. In *Proc. ISCA 19*, pages 342–351, 1992.

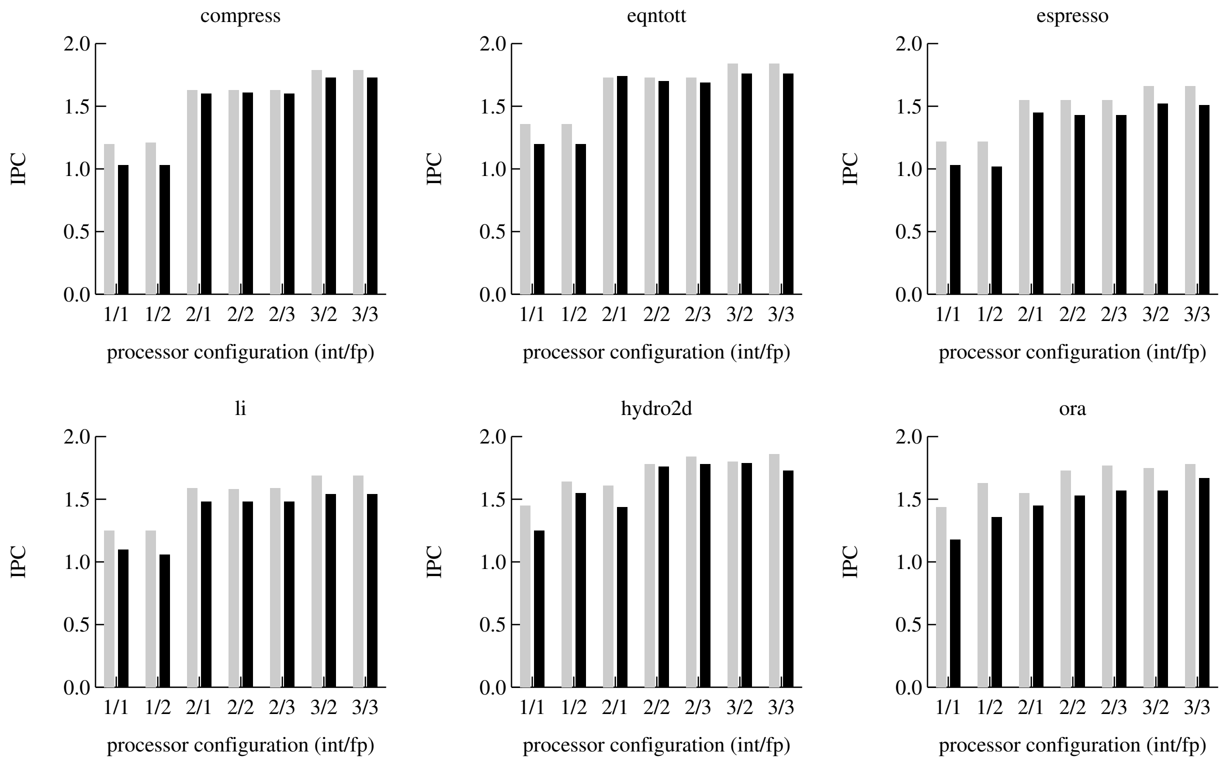


Figure 7: Comparison of performance computed using the model (gray bars) to simulated performance (black bars) for various implementations of the RS/6000. The machine parallelism is varied by adjusting the numbers of integer and floating point functional units.

[DAF94] Pradeep K. Dubey, George B. Adams, III, and Michael J. Flynn. Instruction Window Size Trade-Offs and Characterization of Program Parallelism. *IEEE Transactions on Computers*, 43(4):431–442, April 1994.

[DSP93] Trung A. Diep, John P. Shen, and Mike Phillip. EXPLORER: A Retargetable and Visualization-Based Trace-Driven Simulator for Superscalar Processors. In *Proc. MICRO-26*, pages 225–235, 1993.

[HP90] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., 1990.

[Jou89] Norman P. Jouppi. The Nonuniform Distribution of Instruction-Level and Machine Parallelism and Its Effect on Performance. *IEEE Transactions on Computers*, 38(12):1645–1658, December 1989.

[LW92] Monica S. Lam and Robert P. Wilson. Limits of Control Flow on Parallelism. In *Proc. ISCA 19*, pages 46–57, 1992.

[RF93] B. Ramakrishna Rau and Joseph A. Fisher. Instruction-Level Parallel Processing: History, Overview, and Perspective. *The Journal of Supercomputing*, 7:9–50, 1993.

[TGH92] Kevin B. Theobald, Guang R. Gao, and Laurie J. Hendren. On the Limits of Program Parallelism and its Smoothability. In *Proc. MICRO-25*, pages 10–19, 1992.

[Wal91] David W. Wall. Limits of Instruction-Level Parallelism. In *Proc. ASPLOS-IV*, pages 176–188, 1991.