8-1-1991

Exploiting Fine-Grain Concurrency Analytical Insights in Superscalar Processor Design

Pradeep K. Dubey
Purdue University

George B. Adams III
Purdue University

Michael J. Flynn
Purdue University

EXPLOITING FINE-GRAIN CONCURRENCY: ANALYTICAL INSIGHTS IN SUPERSCALAR PROCESSOR DESIGN

Pradeep K. Dubey

George B. Adams III

Michael J. Flynn *

School of Electrical Engineering

Purdue University

West Lafayette, Indiana 47907

Purdue University

TR-EE 91-31

August 1991

* Department of Electrical Engineering, Stanford University

ACKNOWLEDGEMENTS

The authors thank Prof. Henry Dietz, Prof. Jose Fortes, Prof. Mike Atallah, Prof. Arup Bose, and Raymond Kamin, all of Purdue University.

We also thank Dr. James T. Kuehn at the Supercomputing Research Center in Bowie, Maryland, and the management at the University Computing Center at the California State University, Sacramento, for making their Multiflow computers available for this research. Finally, we thank Mr. Kent G. Fielden of Intel Corporation, Santa Clara for making available technical documentation crucial to the experiments.

TABLE OF CONTENTS

Page

Page

LIST OF TABLES

Table | Page

Appendix
Table

LIST OF FIGURES

Figure Page

Figure Page

Figure Page

Figure Page

Figure Page

Appendix
Figure

Figure Page

Figure Page

Figure Page

LIST OF SYMBOLS

Symbol	Explanation
T	Latency of the instruction execution logic tree with no pipelining (in time units)
Δt	Clock period with pipelining (in time units)
g	Latency of the instruction execution logic tree with no pipelining (in units of gate delays) also referred to as operation gate delay
$g_{pipelined}$	Latency of the logic tree with pipelining (in units of gate delays)
n	Total number of operations
T_n	Time to execute n operations
G	Throughput in terms of number of operations per time unit
s	Number of stages in a pipeline
k	Number of pipelines
N	Number of processors in a multiprocessor system
W	Size of an instruction window
κ	Dynamic overhead coefficient for a pipeline
c	Constant overhead for a pipeline
s_{opt}	Optimum number of segments for maximizing pipeline throughput
$G_{nominal}$	Pipeline throughput at nominal values of all parameters
G_{norm}	Pipeline throughput, normalized relative to $G_{nominal}$

G_{opt}	Pipeline throughput at $s = s_{opt}$ (normalized with respect to $G_{nominal}$)
G_{subop}	Pipeline throughput at $s = s_{subop}$ (normalized with respect to $G_{nominal}$)
G_{ovrop}	Pipeline throughput at $s = s_{ovrop}$ (normalized with respect to $G_{nominal}$)
G_{nom}	Pipeline throughput at $s = s_{nom}$
ΔG_{opt}	Pipeline throughput gain at $s = s_{opt}$ (normalized with respect to G_{nom})
ΔG_{subop}	Pipeline throughput gain at $s = s_{subop}$ (normalized with respect to G_{nom})
ΔG_{ovrop}	Pipeline throughput gain at $s = s_{ovrop}$ (normalized with respect to G_{nom})
u	Resource utilization
$u_{\max}$	Maximum possible resource utilization
v	First-order coefficient in the utilization equation
r	Second-order coefficient in the utilization equation
u_{ns}^{sp}	Utilization factor for the parallelizable code for pipeline stages
u_{ns}^{ss}	Utilization factor for the parallelizable code for complete pipelines
u_{ns}^{mp}	Utilization factor for the parallelizable code for processors
K	Average branch delay
I^{+}	Average number of wasted instructions per branch
α	Fraction of code that must be serially executed on one pipeline
β	Fraction of code that must be serially executed on one processor
$k_{\max}$	Maximum degree of operation (pipeline) level parallelism
$N_{\max}$	Maximum degree of iteration (processor) level parallelism
l	Distance between dependent instructions as a fraction of the size of loop body

π_b	Fraction of branch delays overlapped with execution delays
π_o	Fraction of operand fetch delays overlapped with execution delays
b	Branch frequency
x	Two-clock instruction frequency
$(1-h)$	Cache miss probability
m	Frequency of memory reference per operation
y	Fraction of data accesses to shared variables
d_b	Fraction of operation gate delay required for branch resolution
d_m	Multiple of operation gate delay required for cache miss processing
d_c	Multiple of operation gate delay required for on-chip cache access
D_i	Average delay for wrongfully executed instructions per incorrect branch prediction
μ	Number of cycles for undoing the damage of a wrongfully executed instruction
$I_{i,j}$	Event that instructions I_i and I_j are mutually independent
$P(x)$	Probability of the event x
$P(I_j{:}y)$	Scheduling probability of instruction I_j with I_0
$P_k(I_1, I_2, ..., I_W)$	Probability of having exactly k instructions dispatchable with I_0, in a window of size W
$P_{\geq}(I_1, I_2, ..., I_W)$	Probability of having k or more instructions dispatchable with I_0, in a window of size W
g_k	Throughput of a k-pipeline processor ignoring any dependency constraints
G_k	Throughput of a k-pipeline processor under dependency constraints
δ	Dynamic distance between dependent instructions, also used for inter-iteration dependency distance

p_{δ}	Probability of conditional independence of instructions at a distance of δ
p_{ω}	Probability of scheduling instructions past ω basic blocks

ABSTRACT

Dubey, Pradeep Kumar. Ph.D., Purdue University, August 1991. Exploiting Fine-Grain Concurrency: Analytical Insights in Superscalar Processor Design. Major Professor: George B. Adams III.

This dissertation develops analytical models to provide insight into various design issues associated with *superscalar*-type processors, i.e., the processors capable of executing multiple instructions per cycle. A survey of the existing machines and literature has been completed with a proposed classification of various approaches for exploiting fine-grain concurrency. Optimization of a single pipeline is discussed based on an analytical model. The model-predicted performance curves are found to be in close proximity to published results using simulation techniques. A model is also developed for comparing different branch strategies for single-pipeline processors in terms of their effectiveness in reducing branch delay. The additional instruction fetch traffic generated by certain branch strategies is also studied and is shown to be a useful criterion for choosing between equally well performing strategies.

Next, processors with multiple pipelines are modelled to study the tradeoffs associated with deeper pipelines versus multiple pipelines. The model developed can reveal the cause of performance bottleneck: insufficient resources to exploit discovered parallelism, insufficient instruction stream parallelism, or insufficient scope of concurrency detection. The cost associated with *speculative* (i.e., beyond basic block) execution is examined via probability distributions that characterize the inherent parallelism in the instruction stream. The throughput prediction of the analytic model is shown, using a variety of benchmarks, to be close to the measured static throughput of the compiler output, under resource and scope constraints. Further experiments provide misprediction delay estimates for these benchmarks under scope constraints, assuming beyond-basic-block, out-of-order execution and run-time scheduling. These results were derived using traces generated by the Multiflow TRACE SCHEDULING™ compacting C and FORTRAN 77 compilers.

TRACE SCHEDULING is a trademark of Multiflow Computer, Inc.

A simplified extension to the model to include multiprocessors is also proposed. The extended model is used to analyze combined systems, such as superpipelined multiprocessors and superscalar multiprocessors, both with shared memory. It is shown that the number of pipelines (or processors) at which the maximum throughput is obtained is increasingly sensitive to the ratio of memory access time to network access delay, as memory access time increases. Further, as a function of inter-iteration dependency distance, optimum throughput is shown to vary nonlinearly, whereas the corresponding optimum number of processors varies linearly. The predictions from the analytical model agree with published results based on simulations.

CHAPTER 1
INTRODUCTION

1.1 Motivation

Ever since the advent of first computer, while one group of designers concentrated on achieving an equivalent performance at a lower cost, the other group endeavored to deliver higher performance at affordable cost. In the world of microprocessors, some of the latter group of designers are trying to gain a better understanding of the performance achievable by *concurrent execution of scalar instructions*. Processors capable of such execution are referred to as *superscalar* in recent literature. Because most of the current microprocessors (CISC or RISC) achieve an execution rate of one assembly level instruction per clock, there is a keen interest in exceeding this rate by executing multiple instructions per clock. Contribution to this ongoing research is the primary motivation of this dissertation.

Performance studies can be broadly classified as either *simulation-based* or *analytical*. Most of the work on processor performance has concentrated on the simulation-based approach. The research presented in this dissertation seeks to complement previous work by providing an alternative approach to study processor performance based on relatively simple analytical models. Such models, when validated through correlations with existing simulation-based performance predictions and with empirical data when available, can provide valuable additional insights into performance potential at a fraction of the time needed to run typical simulations or conduct experiments to gather performance measurements.

Architects of next generation processors are often required to provide an as-far-as-possible accurate performance estimate of the proposed design. Common performance estimates these days for microprocessors are numbers such as, SPECmark, which is the geometric mean of the SPEC ratios for the 10 CPU-intensive benchmarks that comprise the SPEC suite. A SPEC ratio is the ratio of execution time for a given benchmark relative to the execution time of that benchmark on a VAX 11/780 running the ULTRIX 3.1B operating system. (SPEC stands for the Systems Performance Evaluation Cooperative.) The traditional performance modelling approach is to run traces of the benchmarks on a low level model of the proposed next generation machine. While this low level model of the new machine must be detailed enough to provide a sufficiently

accurate prediction, it has to be abstract enough to yield feasible simulation run time.

The simulation predicament can be resolved by noting that in moving to a next generation machine design it is the machine architecture that is being redefined and the benchmark traces being used as input may be unchanged. Therefore if the benchmarks can be abstracted or characterized, there would be no need to go through tedious clock-by-clock simulation. There are several ways to characterize a machine architecture, such as the number of pipelines and processors, cache size, branch delay and so on, that can be useful for estimating its performance potential. An analogous set of program features that can serve as an indicator of its performance potential has been almost absent in the published literature. The need for program characterization has also served as a counterpoint motivation for this work.

1.2 Review of Concurrency Representation, Detection and Scheduling Techniques

Given a certain end-user task, the most obvious performance measure is the amount of real time spent in performing the task. Consider the often-repeated question: *Where does the time go* ? This total time is clearly the basis of perceived performance by the end-user and is spent in following transformation stages:

(User level) Algorithm -> HLL -> Assembly -> Micro-instruction (Implementation level)

The sequential nature of above transformation clearly implies that *time lost at any stage is lost for ever*. In other words, an inefficiency introduced at the algorithm level can never be recovered at the microcode level. Also, the representation at each level is mostly sequential in nature (such as the line-by-line program representation in FORTRAN or any assembly language). Most of the time this sequentiality is not essential from correctness point of view but is simply imposed due to representational syntax or resource constraints. Before considering ways of detecting and exploiting the hidden concurrency, consider a concurrency representation framework that is not only generic enough to be common to all these levels, but also specific enough to account for the major time consuming phases at any given level of program transformation.

Assume that at any given stage, a program description consists of a set, $\{\phi\}$, of uniquely numbered operations. Each operation ϕ is a member of a set $\{\gamma\}$ which defines the *instruction set* architecture at that level. An *interface space* between any two levels consists of the complete set of parameter values and storage contents shared between the two levels for communicating the program transformation. For example, at the interface of assembly and microcode, the interface space consists of the set of visible machine registers, shared flag bits for assembly level conditional jump implementation as well as the modifiable user memory space.

Program representation at each level can be described in terms of a generalized AND/OR graph. Such graphs have been used in the past for search space representation [Nil80]. The graph consists of nodes, $i \in \{\phi\}$, such that there is a node corresponding to each program operation. There is a directed arc from node i to node j if the operation corresponding to node j is *dependent* on that corresponding to node i. Simply stated, this implies that operation j can not be initiated until the completion of operation i. Assume that given any two operations, i and j, the boolean relation, $D(i,j)$ which is *true* if j is dependent on i, can always be evaluated. Two nodes m and n are considered mutually independent if neither m is dependent on n nor n is dependent on m. Nodes with two or more children are classified into AND nodes and OR nodes. A parent node with mutually independent children nodes is called an AND node. Each OR node represents a choice point, i.e., a conditional branch is made to one of the arcs depending on the user input available at run time. For simplicity, define a node with a single child as an AND node also. A *solution subgraph* is defined as a graph consisting of the (unique) start node, such that if a node is an AND node, all the arcs originating from it are part of the graph, whereas, if a node is an OR node, *only one* of the arcs originating from it is part of the graph. Thus, at any level of representation there are multiple solution subgraphs. A node at a given level can be considered a compact representation for a similar graph at the following level.

All the operations linked by an AND arc from some node i, can be executed concurrently following the evaluation of the parent node. Exploiting such concurrency at higher levels (such as the algorithm level or the HLL level) is referred to as *coarse grain* parallelism, whereas that at the lower levels (such as the assembly or microcode level) is referred to as *fine grain* parallelism. The time spent in a path is simply the cumulative sum of the time spent in evaluating each node along the path. The time spent at the user level is the time spent in the longest path in the corresponding solution subgraph. This longest path is referred to as the *critical path* in the discussion to follow. The representation permits backward arcs for loop identification. Consequently, the critical path in the solution subgraph is not necessarily a simple path.

In the next section, a wide variety of techniques available for exploiting fine grain parallelism are examined. A brief introduction of data structures commonly used for concurrency representation is discussed next, followed by a survey of a broad spectrum of available design choices and implementation tradeoffs.

1.2.1 Representing Concurrency

A variety of data structures have been suggested for concurrency representation, that is, modelling the inherent concurrency in a given computation sequence. All such models must satisfy the conditions of *determinacy* and *termination* [KaM66]. Informally

stated, determinacy implies that the results that appear in the interface space are invariant under the particular sequence (if any) in which the concurrent operations are executed. In other words, a machine that is capable of simultaneously executing some or all of the mutually independent operations should yield the same result as a purely sequential machine executing the independent operations in some order. Termination means that there are identifiable terminating conditions that occur after a finite number of steps. Next consider some commonly used data structures.

Computation Graphs. Directed graphs, in spite of their irregular structures, have received considerable attention because of their theoretical properties. Computation graphs refer to labeled directed graphs that were first devised by Karp and Miller [KaM66]. Each node of the graph represents an operation, whereas, interpretation of each edge is extended to represent a first-in first-out queue of data directed from one node to another. A node evaluation involves taking a certain number of operands off the incoming edge(s) and placing certain number of results on the outgoing edge(s). Thus, a node can be fired (evaluated) only if the expected number of operands are available. To keep track of the queue of data on each edge, a specific parameter tuple is associated with each edge. This simplified model was later expanded [BBE70] to include conditional branching facilities.

Precedence Matrices. Another data structure that has been studied for concurrency representation is the matrix [TjF73]. Unlike graphs, matrices have a regular structure that makes them better suited for VLSI implementation. However, the number of matrix elements required for a certain task representation grows as the square of the number of operations in the task. Graphs, on the other hand, have a more space efficient implementation. Each operation in a task I_i corresponds to the ith row and i^{th} column of a matrix M. This matrix M is defined as a *precedence* matrix if it is a boolean matrix such that, $M_{ij} = true$ if and only if $(D(i,j)\ OR\ D(j,i))$ is true i.e., if the instructions i and j are not mutually independent. The precedence matrix is symmetric and hence can be considered a triangular matrix with a zero diagonal, because an instruction does not depend upon itself. In the case of graphs, a chain of dependency (for example, operation k depends on j which in turn depends on i) is represented by a path from node i to node k through node j, rather than a direct edge from i to k. Similarly, the precedence matrix also does not contain entries to represent such dependency chains. For example, in this case, although the entries, M_{ij} and M_{jk} are set *true* there is no entry for M_{ik}. To get a complete picture of the dependencies between, say, n operations, one needs to use the matrix M_n, which is given by, $M^1+M^2+.....+M^n$. M^r indicates the matrix M raised to the power r and + refers to the boolean OR operation.

Petri Nets. A Petri net is also a graphical representation of program behavior but with directed edges between two different types of nodes. A node represented as a 'O' is called a *place* and a node represented as a '|' is called a *transition*. The places having edges directed into a transition are called *input* places and those having edges directed

out of the transition are called *output* places. The places have the ability to hold *tokens*. A transition having a token on each of its input places is considered *active* and can *fire*, (i.e. be evaluated). The firing results in removal of a token from each input place and adding a token to each output place. A Petri net with every place having exactly one transition entering the place and only one transition leaving the place is called a *marked graph* and is directly representable as a computation graph.

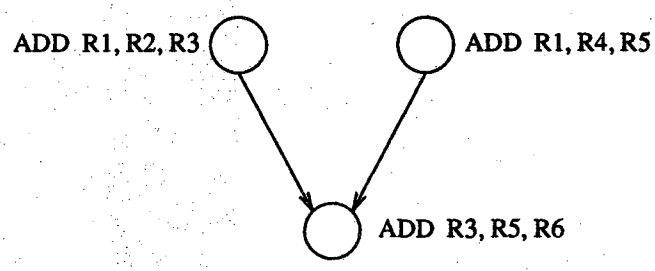
This is illustrated in Figure 1.1 by using these data structures to model the concurrency available in the assembly code sequence given in Figure 1.1.a. While the graph representation in Figure 1.1.b is concise, the corresponding matrix representation in Figure 1.1.c has more elements but is quite regular in shape. A comparison of computation graphs and Petri nets can be found in [Mil73]. This paper also discusses in detail a more general model called parallel program schemata, which includes the notion of a random access memory accessible to the operation nodes for reading and writing. The *presence* of an *arc* between two nodes in the graph representation indicates the *absence* of concurrency, i.e., a *dependency* between the operations. Therefore, maximizing concurrency implies minimizing dependency, which in turn may result in reducing the critical path of the solution subgraph, and lowering the execution time.

1.2.2 Dependencies

An operation can be defined as a function, $\phi(.)$, with source operands $s_1, s_2, \cdots$, also known as the function domain, and the result destinations $\phi(1), \phi(2), \cdots$, also referred to as the range of the function. Thus, a typical operation evaluation consists of reading the source operands, applying the specified function, and writing the result. Consider two operations, ϕ_i and ϕ_j, where ϕ_i precedes ϕ_j in a purely sequential execution. *Data dependency* between these operations results from overlapping ranges and/or domains. If the range of ϕ_i is same as that of ϕ_j, then ϕ_j is said to be *output dependent* on ϕ_i. It is also called *Write after Write* dependency. If the range of ϕ_i is the same as the domain of ϕ_j, and there is no operation ϕ_k, which follows ϕ_i but precedes ϕ_j, such that ϕ_k is output dependent on ϕ_i then ϕ_j is said to be *essentially dependent* on ϕ_i. Such a dependency is also referred to as *Read after Write* dependency or *flow dependence*. On the other hand, if the range of ϕ_j is the same as the domain of ϕ_i, ϕ_j is said to be *order dependent* on ϕ_i, since the dependency exists only if the order specified by the proper sequential execution is reversed. This is also known as *Write after Read* dependency or *anti-dependence*. Finally, if the domain of ϕ_j is the same as the domain of ϕ_i, ϕ_j is said to be *input dependent* on ϕ_i. Input dependence does not imply a lack of concurrence unless there is a limit on simultaneous distribution of input to a number of different operation evaluations.

1. ADD R1, R2, R3 ; R3 := R1 + R2
2. ADD R1, R4, R5 ; R5 := R1 + R2
3. ADD R3, R5, R6 ; R6 := R3 + R5

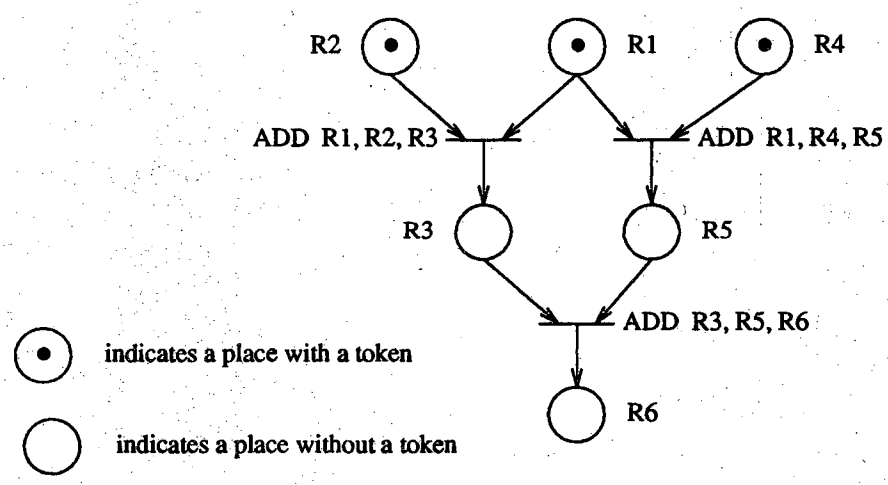
(a)



(b)

	1	2	3
1	0	0	1
2	0	0	1
3	1	1	0

(c)



(d)

Figure 1.1 Computation graph (b), Precedence matrix (c) and Petri net (d) for the sample code sequence in (a)

Resource dependency which is sometimes referred to as *operational* dependency, among operations ϕ_i and ϕ_j may exist if both correspond to the same function, ϕ. Thus, data-independent operations may not be executed concurrently if they must use the same resource for evaluation. Concurrent evaluation of resource dependent operations is possible if the resource can be shared by multiple operations with the same function or if there are multiple resources. In other words, resource dependency implies lack of concurrency only when the number of permissible concurrent operations is exceeded. Generally speaking, resource dependency also includes the lack of requisite interconnection paths to transmit source operands and/or results.

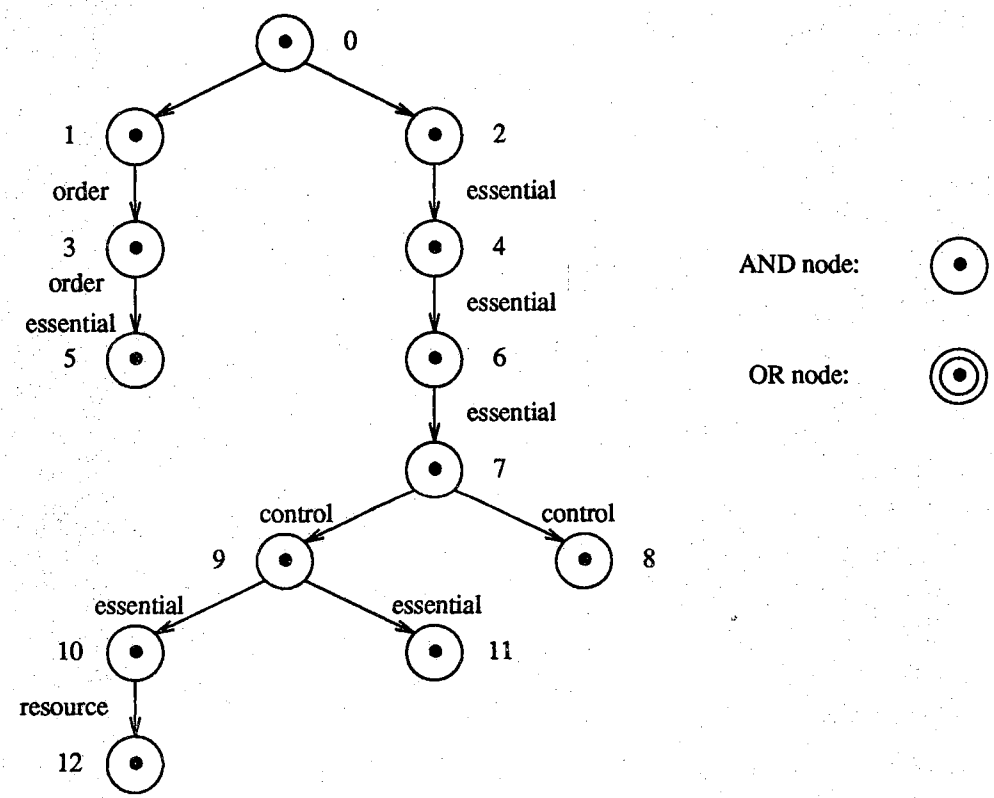
An operation node having an OR node as an ancestor may never need to be evaluated if it is not part of the solution subgraph. In other words, an operation is considered *procedurally* or *control* dependent on all the preceding conditional branch instructions. Thus, while data and resource independence of two operations implies that the operations are *executable* concurrently, procedural dependence tells whether they even need to be executed. Figure 1.2.a lists a typical assembly code sequence, whereas, the corresponding AND/OR graph is given in Figure 1.2.b illustrating the dependencies explained above.

To further explore the nature of dependencies, consider an ideal machine with infinite resources. On such a machine, it should always be possible to remap the range of operation ϕ_j such that it does not overlap the domain of any of the preceding operations. For example, in Figure 2.a R2 can be mapped to, say, R11 so that X3 is no longer order dependent on X1. Similarly, it should always be possible to remap the range of any operation such that it does not overlap the range of any of its followers. For example, the destination register R2 can be mapped to, say, R12 so that X5 is no longer output dependent on X3. Resource dependency becomes a non-issue in such an ideal environment. Further, assume that whenever a choice point is encountered, resources can be replicated such that different OR arcs can be concurrently explored until the control dependency is resolved, at which time the incorrect paths can be discarded. As a result, for an environment with infinite resources, procedural dependency does not imply lack of concurrency either. In this environment only one type of dependency remains, *essential* dependency. Hence, all dependencies other than the essential data dependency can be considered different variations of *resource* dependency.

A solution subgraph may have cycles resulting from *loop* structures in the program, consequently the critical path may consist of one or more iterations of certain sets of nodes. Operations that belong to the same cycle but are data independent in different iterations of the cycle are called *cyclically independent*. These operations may still be resource or control dependent across different iterations and they may be data-dependent in the same iteration. Next take a look at some important design choices for concurrency detection and scheduling.

```
X0:   BEGIN
X1:   MUL  R1, R2, R3      ; R3 := R1 * R2
X2:   MUL  R4, R5, R6      ; R6 := R4 * R5
X3:   ADD  R7, R8, R2      ; R2 := R7 + R8
X4:   SUB  R6, R7, R9      ; R9 := R6 - R7
X5:   SUB  R2, R10, R2     ; R2 := R2 - R10
X6:   CMP  R9, 0           ; set zero flag if R9 = 0
X7:   JMPZ X9              ; Jump if zero flag set
X8:   INC  R10             ; R10 := R10 + 1
X9:   DIV  R1, R2, R3      ; R3 := R1 ÷ R2
X10:  ADD  R4, R3, R5      ; R5 := R4 + R3
X11:  ADD  R6, R3, R7      ; R7 := R6 + R3
X12:  ADD  R8, R3, R9      ; R9 := R7 + R3
```

(a)



(b)

Figure 1.2 AND/OR graph (b) for the code sequence in (a). Assuming a machine with two add/subtract and two multiply/divide units. Input dependence ignored.

1.2.3 Detecting, Dispatching, and Scheduling Concurrent Operations

Concurrency *detection* implies examining a certain number of operations for isolating the data-dependent pairs, whereas *dispatching* refers to issuing some or all of these operations detected concurrent for scheduling. For example, say 16 instructions at a time may be examined for concurrency detection but only the first 4 of them that are found independent may be issued for scheduling. The distinction between detection and dispatch is subtle and the distinction does not exist if all detected concurrent operations can always be sent for scheduling. *Scheduling* is the process of assigning specific operation functions and the corresponding source and result operands to designated resources at designated times under the constraints imposed by data dependency and limited number of resources.

An optimal schedule is defined as the one that minimizes the number of distinct execution time slots and, ignoring resource limitations, it corresponds to a schedule constrained solely by essential data dependency. This can be achieved by scheduling together all operations that are essentially independent of all incomplete operations. Thus, an optimal schedule minimizes the critical path length for the corresponding solution subgraph leading to a compact (less deep) graph representation. Viewing each operation as a *task* and each resource as a *processor*, the problem of *resource constrained processor scheduling*, can be mapped to the optimal scheduling problem. The resource constrainted processor scheduling problem is a known NP-complete problem [GaJ79]. Therefore, optimal scheduling is NP-complete also.

Although *instruction* is sometimes used to refer to multiple operations that have been scheduled together, these are used interchangeably in this section except when the distinction is critical.

Scope of concurrency detection. Concurrency detection begins by examining a consecutive set of operations from the serial execution sequence. The number of such operations simultaneously examined for detecting a concurrent subset defines the *scope* of concurrency detection. The larger the scope, the greater the probability of detecting a larger subset of concurrent operations. Identifying a stream of instructions that would necessarily be executed in sequence in a purely sequential execution is complicated due to the presence of conditional branches. Such branches direct the execution sequence along one of the multiple solution paths and the choice is known only at run time, whereas concurrency detection must precede execution. A *basic block* is defined as a maximal sequence of instructions containing a single conditional jump which is the last instruction in the sequence. There may be one or more unconditional jumps within a basic block.

Concurrency detection techniques can be classified into two categories depending on whether the scope is limited to within the basic blocks or stretched across basic blocks. The speedup achievable in the second category can be significantly more than

that obtainable in the first category, as evident from the comparison given in Table 1.1. Experiments done by Tjaden and Flynn [TjF70] and Riseman and Foster [RiF72] are some of the earliest results reported in the area of concurrency detection. In the first published report on concurrent execution, Tjaden and Flynn predicted an average speedup of about 1.86 through simulations in an IBM 7090 environment and limiting the scope to be within basic blocks. Riseman and Foster estimated an average speedup of 51 in an ideal environment with infinite resources. This showed that the speedup achievable with scope not limited to basic blocks can be an order of magnitude better than that possible with scope held within a basic block. A wide range of experiments reported so far (refer to Table 1.1), confirm the range of speedups reported by Tjaden/Flynn and Riseman/Foster.

Another important factor influencing the scope of concurrency detection is whether the detection is being done at run time (dynamically) or at compile time (statically). At run time only a fixed size instruction window can be used to examine a set of instructions, whereas at compile time potentially the entire program can be examined. While static detection techniques can afford a larger scope, they are limited to compile time information only. Dynamic techniques, with run time information available, are limited to a much smaller scope. The complete machine state is known only at run time, hence, an instruction sequence such as Multiply/Load/Multiply may create a resource dependence at compile time but could be non-existent if the Load caused a cache miss that sufficiently delayed the following Multiply. Wedig [Wed82] provides an analysis of the complimentary nature of static and dynamic concurrency detection. Static techniques restricted to basic blocks are also known as techniques for *local code compaction*, whereas those extended across basic blocks are considered aimed at *global code compaction*.

Level of Concurrency Detection. Four different levels (or stages) of program specification (transformation) were outlined at the beginning of Section 1.2. Attempts have been made to detect concurrency at all four levels. Systolic architectures [Kun82] exploit concurrency at the algorithm level. Data flow architectures (such as, [PHS85]) and recent VLIW architectures attempt concurrency detection at the level of microcode.

Processors capable of concurrent execution of multiple scalar operations at the assembly level have been referred to as *superscalars* (earliest reference to this term is found in [AgC87]). Imagine concurrency detection at assembly level with a scope limited to adjacent instructions only. Consider two adjacent divide operations:

```
DIV R1, R2, R3        ; R3 := R1 / R2
DIV R1, R4, R5        ; R5 := R1 / R4
```

If the machine organization is restricted to a single divide unit, then these two divides would be serialized. However, suppose each divide corresponds to several lines of microcode and the concurrency detection is attempted at the *microcode level*. Now it is certainly feasible to overlap certain micro operations corresponding to the two divides.

Table 1.1 Comparison of concurrency detection and scheduling strategies

Strategy	Scope within/beyond basic block	Level high-level assembly micro-code	Issue sequential out-of-order	Execution Completion sequential out-of-order	Scheduling strategy	Reported Speedup
Tomasulo [Tom67]	within	assembly	sequential *	out-of-order	data-flow	n/a
Thornton [Tho70]	within	assembly	sequential *	sequential	control-flow	n/a
Tjaden/Flynn [TjF70]	within	assembly	out-of-order	out-of-order	control-flow	1.86
Riseman/Foster [RiF72]	beyond	assembly	out-of-order	out-of-order	control-flow	51.2
Tjaden [Tja72]	within	assembly	out-of-order	out-of-order	control-flow	1.96
Kuck [KMC72]	beyond	high-level	out-of-order	out-of-order	static-stream static	8
Wedig [Wed82]	beyond	high-level	out-of-order	out-of-order	control-flow	3
Weiss/Smith [WeS84]	within	assembly	out-of-order *	out-of-order	static-stream data-flow	1.58
Nicolau/Fisher [NiF84]	beyond	micro-code	out-of-order	out-of-order	static Trace Scheduling	90
HPSm [PHS85]	beyond	micro-code	out-of-order	out-of-order	data-flow	n/a
Uht [Uht86]	beyond	high-level	out-of-order	out-of-order	control-flow	2
Hsu/Davidson [HsD86]	beyond	assembly	out-of-order	out-of-order	static-stream static Decision Tree Scheduling	1.3-3.9
Acosta et.al. [AKT86]	within	assembly	out-of-order	out-of-order	control-flow	2.79
Sohi/Vajapeyam [SoV87]	within	assembly	out-of-order *	out-of-order	data-flow	1.8
iWARP [Lam88]	beyond	micro-code	out-of-order	out-of-order	static Software Pipelining	3
CYDRA [RYY89]	beyond	micro-code	out-of-order	out-of-order	data-flow	n/a
Smith et.al. [SJH89]						
ideal fetch unit	beyond	assembly	out-of-order	out-of-order	control-flow	2.3-4.1
non-ideal fetch unit	beyond	assembly	out-of-order	out-of-order	control-flow	1.9-2.3
Johnson [Joh91]	beyond	assembly	out-of-order	out-of-order	control-flow	2

Note: Unless otherwise indicated, all control-flow strategies are based on dynamic instruction-stream.
Issue strategies marked with an asterisk (*) are limited to a maximum of one instruction per clock.
Other issue-strategies can issue more than one instructions per clock.
Speed-ups given without a range are best-case speedups.
Speed-ups reported should be taken with caution, as they are not relative to the same baseline processor.

For example, it may be possible to load the source operands to the divider inputs or do the divide-by-zero check on the second divide before the first one finishes. But in order to be able to detect such opportunities, the scope of detection must go beyond adjacent microcode lines. Thus, as the program transformation moves closer towards the machine level away from the user level, potential parallelism goes up along with the scope required for its detection (which also makes the detection a harder task).

In an ideal sense, concurrency detection done at the microcode level with an infinite scope has maximum potential. For example, in an extreme sense, this would even explore the possible microcode overlap of two different *sort* operations specified at the highest algorithmic level. But this also implies exposing lowest level machine resources at the highest level of specification, which may not be desirable. While most of the concurrency detection experiments have been attempted at the assembly level, certain techniques are aimed at concurrency detection at the lowest level. Dynamic techniques of this type fall into the classical data flow category [DeM74, ArG82], whereas some such recently emerging static techniques are referred to as the VLIW approach [NiF84].

Very Long Instruction Word (VLIW) machines are characterized by a central control unit issuing each cycle a single wide instruction word consisting of independent operations. Note that these instruction words are machine instructions (microcode lines) and, hence, there is no additional level of interpretation involved. Similar to earlier vector machines, VLIW machines carry out many fine grained and tightly coupled operations simultaneously, whereas, in contrast to the vector machines, these concurrent operations are dissimilar and logically unrelated. Typical instruction word length for some implementations ([Fis83], [CNO88]) range from 512 to 1K bits.

This horizontal format leads to poor code density in case the available parallelism is limited. In an attempt to improve code density, iWARP [CGL89] relies on two instruction formats. It uses a short instruction format in case of limited parallelism and a long format otherwise. The Multiflow TRACE [CNO88] uses a variable length memory representation that eliminates NOPs from the fixed length machine instruction format to improve the code storage efficiency. There are two other important implications of such a design strategy. Firstly, considering the fact that about 5 to 10 percent of operations at the lower levels (assembly or microcode) are conditional branches, a wide instruction word would contain multiple independent conditional branches. Therefore, such machines must be capable of performing tests for multi-way jumps to separate targets. Secondly, the memory system should be capable of supporting multiple memory references, which in turn should be scattered among different memory banks. Although such concerns are current implementation barriers for VLIW machines, the problems they represent are applicable to almost any approach to exploiting high fine grain concurrency. Responding to interrupts with restartable machine state poses another challenge for VLIWs and is discussed later.

Directly Executable Language machines. A machine architecture that retains all the information of the high level language allowing greater possibility of concurrency detection (done at the language level) has been proposed by Flynn and Hoevel [FlH79]. This representation, referred to as *Directly Executable Language*, provides a one-to-one correspondence between the states in the high level language and the machine states. Although at lower levels potentially more parallelism can be detected, this parallelism besides being fine grain (for example, overlapping micro-ops as opposed to overlapping say, FOR loop iterations) is also harder to detect. This is because at a machine level of representation that does not bear direct correspondence with the high level representation, most of the coarse grain concurrency information is not preserved in an easily recognizable form. For example, a FOR loop iteration count is more easily recognizable for concurrency at the language level than at the assembly level, and is still more difficult to recognize at the microcode level. Wedig [Wed82] provides details of concurrency detection at the language level.

Static Concurrency Detection and Scheduling. These techniques are based on information available prior to run time. One of the earliest and most extensive works in this area was done by Kuck and his colleagues [KMC72] for concurrency exploitation in a serial language such as FORTRAN. Their work explores height reduction techniques for program graphs, semantic analysis, and branch elimination to extract significant amount of hidden parallelism. While some of the static optimizations are strictly aimed at reducing non-essential data dependency and/or procedural dependencies and are thus machine independent; others also rely on explicit information about machine resources to resolve other dependencies.

Sometimes the range or domain of operations may be indirectly specified. For example, instead of a direct specification, like R2 or A[2], the domain of the jth operation may be $A[l]$, whereas the range of a preceding i^{th} operation may be $A[m]$. These two operations are data independent if the array indices m and l are not the same. Because, m and l may be arbitrary expressions, this *anti aliasing* or *disambiguation* may be difficult or even impossible to perform at compile time. In case of any doubt, the only safe option is to assume dependence in such cases. At times, a simple analysis may reveal the independence. For example, if $m=2x+1$ and $l=2y$, then there are no integer values of x and y such that $m=l$, since m is always odd and l is always even.

Banerjee [Ban79] has developed efficient algorithms for determining whether m and l, where each is a polynomial, may imply reference to the same variable, and hence a conflict. Nicolau [Nic89] proposes an alternative solution to the disambiguation problem. This technique known as run time disambiguation, relies on assumptions about the run time behavior of memory references to allow compile time code restructuring to extract available concurrency. For example, based on run time statistics, suppose m and l are most likely unequal. Given this information, the compiler is allowed to extract any potential for concurrency resulting from this disambiguation, conditioned on the fact that

$m \neq l$. This conditional (IF $m \neq l$) is evaluated at run time and if found to be untrue, sequential execution proceeds as if no optimization was done. On the other hand, in the more likely case of $m \neq l$, introduced optimization results in increased parallelism. The overhead of additional condition evaluation may be nil if it can be overlapped with some previous operation.

A similar approach can be taken for evaluating conditional branches. On the basis of run time statistics, the compiler can be made to pick the most likely branch path resulting in a larger basic block size which in turn implies larger potential for concurrent evaluation.

Trace Scheduling, developed by Fisher [Fis81], replaces block-by-block local code compaction with simultaneous code compaction of a trace across many basic blocks. A *trace* is defined as a loop free sequence of instructions which might be executed sequentially for some choice of data. Improved performance is obtained by optimizing along the trace most likely to be followed at run time. Heuristic or profile-based branch predictions are used for picking the trace along the solution path with highest probability. Such an approach is quite likely to result in schedules that will not correctly preserve the semantics in case the less likely off-trace path is taken at run time. A post processing phase inserts *compensation* code into the program graph on the off-trace branch edges to undo these inconsistencies, thereby restoring program correctness. Such concurrency detection has been typically attempted at the microcode level, and the large block size at this level implies wide instruction machine word formats. For data-dependent conditional branches, the fundamental assumption that there exists a most frequently executed solution path is questionable. In such cases the overhead of compensation code can offset any speedup in the off-trace paths.

After picking the most likely trace, trace scheduling generally does not distinguish the off-trace paths on the basis of their probabilities. As a result, the schedule generated is not very sensitive to the actual path probabilities. Hsu and Davidson [HsD86] propose a refined heuristic that addresses this issue. This technique, *decision tree scheduling*, while much more sensitive to actual path probabilities, is intended for code reordering to make efficient use of *guarded* store and jump instructions. These guarded instructions make efficient use of the delayed part of a conditional branch instruction (time slots taken for the branch resolution). Each guarded instruction is accompanied by a *guard expression*, which is just a boolean valued expression. Whenever, a guard expression evaluates to fault, it inhibits writing the final result, i.e., the update of the interface space. This effectively converts the guarded instruction into a NOP. The performance potential of this strategy is a function of how many time slots are available during branch resolution. The speedup reported for this technique in the Table 1.1 is based on a pipeline uniprocessor model for the scalar portion of the CRAY-1 computer with branches taking a constant 14 cycles.

Iterative constructs, or loops, are very common in numerical applications, and so deserve special attention. Two techniques have been most commonly used for loop optimization: *loop unrolling* and *software pipelining*. Loop unrolling consists of replicating the loop body n times, where n is the *degree* of unrolling. All conditional branches are removed from the replicated blocks except for the last one, and the index register increment is removed from all but the last replicated block. An advantage of this approach is elimination of some conditional branches, resulting in larger basic block size and hence the possibility for more speedup ([Nic89], [WeS87]). A disadvantage is that unrolling expands object code size.

Software Pipelining refers to successive initiation of iterations of a loop at constant intervals, even before the preceding iteration completes so that the loop throughput is improved. Using this approach, unlike with other techniques, pipeline stages of the functional units are not emptied at the iteration boundaries or some fixed multiple of iteration boundaries (as is the case with loop unrolling). The objective is to minimize the the interval at which the initiations take place. Such techniques are certainly not new and have been explored in a generalized sense (for example, initiations do not have to be at constant intervals and can instead follow a fixed pattern of intervals) for hardware pipeline scheduling [PaD76]. However, Lam [Lam88] proposes software pipelining as an effective and viable scheduling technique for VLIW processors.

Because the problem of finding an optimal schedule is NP-complete, static scheduling techniques rely on heuristics to restrict the search space. A hierarchical reduction scheme is proposed in [Lam88] to make software pipelining applicable to all innermost loops including those with conditional statements. Conditional OR nodes of the program graph are reduced to a single node with scheduling constraints representing the union of the scheduling constraints of its children. In addition to cyclical data dependency constraints, such a scheduling technique must also take into consideration resource constraints. Assume the initiation interval is m. If a resource is in use by an operation in some i^{th} iteration, in some cycle s, it will also be in use by successive iterations in cycles $s+m, s+2m, \ldots$ and so on. Therefore, another operation belonging to the ith iteration may not use the same resource in cycles s modulo m. This is known as the *modulo constraint* [RaG81]. Software pipelining has been used for compile time concurrency detection and scheduling on the iWARP machine [CGL89].

Another technique for detecting parallelizable loop iterations similar to run time disambiguation is *run time dependence checking* [Nic89]. Unlike the former, probabilistic estimates are not used, instead loops are prepared for run time dependence checking. This is achieved by inserting appropriate code that helps perform automatic dependence checking on different loop iterations and simultaneously schedules independent iterations.

The scheduling techniques mentioned above are mostly applied at lower levels of program transformation. *Percolation scheduling* can be used for program graph compaction for extracting both fine grain as well as coarse grained parallelism. The technique is based on certain core transformations which, when applied on adjacent nodes, help *percolate* them towards the top of the program graph. The goal of such transformations is to compact the program graph by moving operation nodes from the bottom of the graph for grouping with independent operation nodes towards the top of the graph. These transformations consist of various dependency checks and can be combined with a variety of guidance rules to direct the optimization process. Details of these transformations can be found in [Nic85].

At the micro operation level, as the nodes percolate up, nodes grouped together can be treated as a long instruction word with independent operation fields. Thus, percolation scheduling offers another alternative to code generation for the VLIW machines. While trace scheduling explores the program graph in a *top down* fashion along a trace, percolation scheduling searches the graph in a *bottom up* manner. If possible, operations belonging to different branch paths (traces) along with the condition for branch resolution are evaluated simultaneously and the undesired result discarded.

Dynamic Concurrency Detection and Scheduling. Dynamic concurrency detection techniques have the advantage of precise run time information for resolving dependencies related to conditional branches and indirect memory references. Compile time techniques, such as run time checking, although used during the compilation phase, provide run time support for concurrency detection and it is generally believed that a combination of static and dynamic support has performance potential exceeding either technique in isolation. A comparison of different dynamic techniques is given in [AKT86]. The order in which compiled instructions are executed during a purely sequential execution forms the *dynamic instruction stream.* The order in which instructions are generated by the compiler, which is same as the order in which they appear in system memory, forms the *static instruction stream.* Dynamic concurrency detection is either performed on the dynamic instruction stream or on the static instruction stream. The advantage of static stream analysis is reduced memory traffic, because instead of a memory load of each instruction to be executed, static stream detection works with a single load of the static sequence. Furthermore, static stream analysis can achieve the same amount of concurrency as that using dynamic stream analysis [Wed82].

Scheduling can either be done centrally at the time of decode, or in a distributed manner in the functional units themselves. The former approach is called *control flow scheduling*; the latter is called *data flow scheduling*.

There is a global station for *control flow scheduling* that receives information from the functional unit to detect and dispatch independent instructions. One of the earliest implementation of this idea is found in the CDC 6600, where the central station is called

a scoreboard [Tho70]. Under this scheme, an instruction to be executed on a functional unit can be issued even if its source operands are not available. The unavailability of the operand is indicated using a *ready bit*. As soon as the operand becomes available, the functional unit producing it notifies the central scoreboard, which updates the corresponding register and its associated ready bit. This updated information is also conveyed to the waiting functional units. Some important features of this algorithm are:

1) There is no direct communication path among the functional units, they communicate via the central station, the scoreboard,
2) Instruction dispatch (issue) logic blocks when it encounters an instruction that is resource dependent or output dependent on a pending instruction,
3) An instruction I_j that is order dependent on an instruction I_i is allowed concurrent execution with I_i, but the functional unit associated with I_j stays busy until the execution of I_i completes,
4) Dispatch logic is limited to one instruction per cycle.

Tjaden and Flynn [TjF70] suggest a lookahead scheme capable of more than one instruction issue per cycle using a *predecode stack* as the central station. This stack stores instructions in a modified format that explicitly encodes their dependency information. This approach further relies on a register renaming technique to reduce dependencies. An instruction that is independent of all instructions above it on the stack is dispatchable. Simultaneous bit-by-bit compare is used to detect and dispatch independent instructions on the stack. A stack size of around eight is found to be enough to extract all available concurrency. Acosta, et. al. [AKT86] present another variation of this idea using a *dispatch stack* which reduces the associative compare overhead associated with the former approach. In this case, the instruction format is further expanded to contain counters for its source and destination registers. There is a counter with each source register indicating how often it is designated as a destination registers in the preceding, incomplete instructions on the stack. Two separate counters are used to track how many times the destination register is designated as a source and destination register in previous incomplete instructions. These counters are added to compute an *issue index* for each instruction. This computation simplifies the issue logic. All instructions with null issue index are simultaneously dispatchable. As instructions complete, the stack is properly updated. Although the stack update requires content addressability, comparison hardware required overall is likely to be less than the previous approach.

The techniques described so far all work with the dynamic instruction stream. One of the first experiments using the static instruction stream for dynamic concurrency detection and scheduling was reported by Tjaden [Tja72] using ordering matrices for concurrency representation. This work was further extended by Wedig [Wed82]. An important problem with static stream analysis is the complexity of representing machine state at any time during execution. Instructions that complete execution have their

dependencies *deactivated*, so that they are excluded from further analysis. For instructions that are executed multiple times, Tjaden associates a flag which is set on instruction execution and allowed to be reset if the instruction is to be re-executed. This avoids multiple loading of such instructions for concurrency detection. An alternative representation is proposed by Wedig, that associates an *execution vector* with the task being analyzed. The elements of these vectors keep track of number of times different instructions have executed.

Unlike the control flow approach, dependency resolution for *data flow scheduling* is distributed across different functional units. The IBM 360/91 [AST67] was the first system to use data flow scheduling. Although the original algorithm was devised by Tomasulo [Tom67] for the floating point unit of this machine, it can be easily generalized to any system with multiple functional units. Under this scheme, each functional unit contains a set of *reservation stations*, where instructions are held pending execution. Each station contains a field for each of its source operands and the result. Each operand field either contains the operand value (if available) or it contains a *tag* indicating the functional unit that is supposed to produce that value. Each machine register is augmented by a *busy bit*. If this bit is clear, register contents are valid, else the register contains a pointer or a *tag* to the functional unit that is expected to produce the result as its next output. When a result is produced, a *common data bus* simultaneously relays it to all reservation stations as well the machine register files, which use associative comparison with their tags to read the result off the bus. As compared to Thornton's algorithm:

1) There is no central scoreboard and thus dependency resolution, where precedence is controlled by means of tags, takes place in a distributed manner.
2) A common data bus provides a direct communication path between functional units.
3) Automatic register renaming reduces order dependency. For example, in the sequence shown in Figure 1.2.a, register R2 in instructions X1 and X3 would be mapped to two reservation stations. As a result, not only execution of X1 and X3 can be overlapped, but unlike Thornton's approach, X3 can finish before X1 because the original contents of R2 have already been copied to the reservation station of the functional unit executing X1. This resolves order dependencies.
4) Output dependency does not block instruction dispatching either, since the register tag is always updated to point to the most recent functional unit from which the result is expected.
5) Instruction issuing is blocked when there are no available reservation stations for the desired functional unit. Issue rate is still limited to one instruction per cycle.

Weiss and Smith [WeS84] simulated performance of the CRAY-1 scalar architecture using a variation of Tomasulo's algorithm. The scheme proposed uses a *tag pool* consisting of a finite set of tags for assigning destination tags. Therefore, unlike

Tomasulo's algorithm, the tags are not in one-to-one correspondence with the reservation stations, and instruction issue can also be blocked if there are no tags available in the pool. This variation was based on the observation that only a subset of all possible reservation station fields may be active simultaneously, hence the common tag pool would reduce the associative search overhead. In another experiment, they propose a *tag search table* to eliminate the need for associative tag search. This restricts a particular tag to be used with only one reservation station. The search table is indexed by tag value and contains the address of the corresponding reservation station. A *used* bit associated with every tag in the pool blocks its multiple usage. They found that even this restricted version of Tomasulo's algorithm retains much of the performance gain using the associative version.

The work of Weiss and Smith was further extended by Sohi and Vajapeyam [SoV87]. In addition to a separate tag unit, responsible for managing the tag pool, this scheme contains another common pool for the reservation stations. As indicated earlier, under Tomasulo's algorithm instruction issue would block if there are no reservation station available for the desired functional unit, even though there may be unused stations with other units. Therefore, a common pool of reservation stations that are dynamically assigned, can be expected to provide improved performance.

At this point it can be seen that the machine organization starts to resemble that of data flow computers, which are characterized by token issuing and matching units similar to the tag pools described above. In the data flow model of computation [ArG82], execution of an operation is only contingent upon the availability of its input operands and a free functional unit. Thus, when implemented at a fine grain level, data flow tends to expose all available concurrency in the program graph. Complete concurrency exploitation at the machine level is facilitated by high level program specification using a functional language. This has traditionally been met with reluctance in the user community, since it implies setting aside a vast amount of software built over the years in traditional languages like FORTRAN, PASCAL, or C. Furthermore, programs written in functional languages tend to consume large amounts of memory space due to the single assignment rule and copying of data arrays. Some recent experiments have relied on some of the properties of the data flow model to utilize fine grain concurrency, while keeping the traditional program specification at the high level.

Patt, et. al. [PHS85] report a variation of data flow referred to as *restricted data flow* architecture. Unlike classical data flow machines, the data flow graph of only a small subset of the program is kept in the machine at one time. Thus, fine grain parallelism present in this active instruction window is utilized. A *merger* unit takes the data flow graph of each instruction from the dynamic instruction stream, resolves any existing data dependency using a variation of Tomasulo's algorithm, and merges it into the data flow graph resident in the active instruction window. An instruction that completes execution is *retired* from the active window when all the preceding instructions have also retired.

Directed data flow, coined in connection with the Cydra-5 architectural design [RYY89], refers to an architecture that supports the data flow model of computation in a compiler directed fashion. The compiler support is similar in many ways to the VLIW approach. In addition, it provides hardware support for overlapping execution of different loop iterations. It combines the register storage and functional unit inputs and outputs into a single entity referred to as the *context register matrix*. This provides a new context for each new loop iteration. These *iteration frames* are dynamically allocated at run time. Control dependencies are handled at the micro operation level by associating a *predicate* with each operation. This predicate determines whether the corresponding operation needs to be evaluated at all. As soon as all the control dependencies are resolved, the predicate is set, which makes this operation immediately schedulable.

In-sequence or Out-of-sequence detection and scheduling. In-sequence detection implies that concurrent instructions are restricted to be in monotonically increasing number sequence, or in the sequence of a purely serial execution. As a result, the instructions are examined in sequence, and detection and scheduling block every time a dependent pair is encountered. Out-of-sequence detection and scheduling means out of order concurrent instructions are allowed to be simultaneously executed. While, the former is simpler to implement, the latter may have significantly higher concurrency potential in a large scope. Foster and Riseman [FoR72] describe a preprocessing algorithm that generates a reordered code sequence which has the property that if the instruction at the top of the dispatch stack is found dependent and, hence, not immediately dispatchable, there will be no instruction below it that is ready for dispatching.

Independent of whether instructions are issued in-order or out-of-order, they may be allowed to complete in-order or out-of-order. The least restrictive option, that is allowing out-of-order instruction issue and out-of-order completion, exploits maximum concurrency. A major difficulty with out-of-order execution is restoration of machine state for restartability in case of interrupts.

1.2.4. Implementation Tradeoffs

Interrupt Handling. Interrupts pose a special challenge to architectures that overlap executions of elementary operations. Interrupts can be defined as normally unexpected events that are detected at run time and require modifications in the current execution sequence. The unexpected nature of interrupts means that the program corresponding to an interrupt service routine must be inserted arbitrarily into the executing program, during its execution. This does not deserve any special attention for machines with purely sequential execution, since it is accomplished simply by inserting a branch to the service routine immediately after the execution of current operation. However, for machines that overlap operation execution, an operation ϕ_j that is found independent of all incomplete

preceding operations and hence scheduled together with some operation ϕ_i, may be dependent on a newly inserted (and hence incomplete) operation ϕ_k that belongs to the interrupt service routine and precedes ϕ_j in purely sequential execution. Furthermore, on machines that permit out-of-order execution, ϕ_j may have completed execution long before ϕ_k is detected, which can happen only after ϕ_k is inserted. As a result, any schedule can potentially be rendered incorrect at run time if it does not include the possibility of branch to service routines at arbitrary points of sequential order of execution.

A possible solution to this problem can be to insert an OR node corresponding to a possible branch to different service routines after every operation in the purely sequential model of execution. While it would guarantee robust schedules under all combinations of interrupts, it reduces the basic block size to one instruction. Such a solution is unacceptable. An alternate solution is to start with a program graph that excludes the OR nodes corresponding to different interrupt possibilities. When the interrupt is detected, the schedules are modified to incorporate the newly added nodes. This approach is similar to trace scheduling, in that the emphasis is on concurrency exploitation along the most likely program trace, which is the one with no interrupts or exceptions. As a result, the additional overhead of providing compensation is incurred for any damage caused by the wrong guess when an interrupt is detected. This need to compensate further implies a delay between interrupt detection and recognition, where the recognition refers to start of execution of the corresponding service routine. This delay is sometimes known as interrupt latency.

At the time of interrupt recognition, if the state of the interface space is same as that during a purely sequential execution, the corresponding interrupt is called a *precise interrupt*. Precise interrupts have a long latency, since the recognition takes place only after the effects on the interface space of all the operations following the interrupt detection point have been undone. Not only this repair is expensive to implement, it has an unavoidable adverse side effect on performance because the operations undone need to be reexecuted. Not all interrupts require a complete restoration of machine state to that of a strictly sequential execution for program correctness. There are cases when a partial or even zero recovery would be acceptable, which leads to the concept of *imprecise interrupt*. Such interrupts allow the state of interface space at the time of recognition to be different from that during a strictly sequential execution. An imprecise interrupt can have its own degree of impreciseness depending on the difference between the two interface spaces.

Interrupts can also be classified into two categories depending on whether they are caused by an event internal or external to the program execution. Examples of internal interrupts (also known as *exceptions*) include events such as divide by zero, a page fault on an operand fetch, or an incorrect branch prediction. External interrupts are events such as, an I/O interrupt or a request to relinquish a shared bus.

Implementation of Precise interrupts. Smith and Pleszkun [SmP85] present implementation details for different strategies aimed at implementing precise interrupts in pipelined processors. The options suggested there can be broadly classified into two categories:

1) *Reorder buffer.* This strategy employs buffers to reorder the updates to the interface space such that an operation is allowed to update the space only if all the preceding operations have completed without any exceptions. The results waiting in the reorder buffer are unavailable for further computation, hence, although simpler to implement, this scheme forces in-order completion of instructions and the associated performance degradation. Variations on this scheme try to provide associative search ability in the reorder buffer to be able to bypass the interface space for items waiting in the buffer to be committed [SmP85]. The implementation cost of such variations is high.
2) *History buffer.* These buffers are used to keep a copy of the original contents of the registers and memory locations that get updated. When an instruction successfully completes execution, the corresponding history buffer entry is deleted. On detecting an interrupt, the offending instruction is used to index the buffer for restoring the machine state as if none of the following instructions had any effect on the interface space.

Sohi and Vajapeyam [SoV87] and Hwu and Patt [HwP87] have suggested further variations of above ideas.

Implementation of imprecise interrupts. Although these are much simpler to implement, there are tradeoffs. Once an interrupt is detected either all operations in progress can be aborted, or they can be allowed to run to completion before interrupt recognition, or the entire machine state can be saved so that the program execution can be restarted at the point of interrupt after servicing the interrupt. The first option has minimum latency, but suffers from the performance loss of reexecuting the partially executed operations. The third option suffers from the cost of saving the entire machine state. Thus, the second option offers a good compromise. In case of VLIW machines, where each instruction word issued consists of several independent operations, sometimes not all the information needed to carry an operation to completion is issued at the same time [RYY89]. In such cases, care has to be taken to selectively mask operations which would initiate new operations and allow only operations which are needed for completing the pending ones.

Concurrent and overlapped execution. The methods discussed so far have emphasized detection and concurrent execution of independent operations. Consider two operations ϕ_i and ϕ_j at a given level (say, assembly level) of program transformation, and the corresponding set of sub operations $\phi_{i1}, \cdots \phi_{ik}, \cdots$ and $\phi_{j1}, \cdots \phi_{jl}, \cdots$ after transformation to a lower level (say, the microcode level) of representation. It is quite

possible that ϕ_i and ϕ_j are detected as dependent but some sub-operations ϕ_{ik} and ϕ_{jl} are found independent, and hence executed concurrently. The simultaneous execution of ϕ_{ik} and ϕ_{jl}, while a concurrent execution at the lower level, contributes to overlapped execution of ϕ_i and ϕ_j at the higher level. If certain set of sub-operations is repeatedly used during the transformation in a regular sequence (for example, fetch the instruction, fetch the operand, execute the operation), it lends itself very naturally to a pipelined form of implementation.

Resource utilization has not been an explicit concern so far. It has been implicitly assumed that care has been taken to ensure proper utilization of resources at all levels. In fact, it is this very concern that manifests itself in the form of resource dependency. In order to achieve a cost-performance balance that is globally optimum, system resources at every level should be locally optimized, too. At times when concurrent execution is sacrificed in favor of better resource utilization, the performance penalty resulting from the resource dependency can be mitigated by using an optimized pipelined design for the existing resource. Thus, a combination of concurrent and overlapped (pipelined) execution holds the key to an optimal design. Machines with wider instruction words tend to have a slower clock period to be able to generate control signals for the additional hardware. A pipeline with twice the number of segments can potentially have about the same throughput as two pipelines of half the size. However, under less than ideal conditions, doubling the number of segments does not mean halving the clock period. Only if the optimal performance obtainable from one pipeline is significantly less than that from two would a resource duplication would be justified. Experiments done by Sohi and Vajapeyam [SoV89] report the performance potential of machines with restricted instruction word width and deeper pipelines.

Instruction fetch Limitations and Branches. The adverse impact of conditional branches in introducing control dependencies and thereby limiting the ability to lookahead has been discussed in detail in the previous sections. There are two other overheads that are associated with both conditional and unconditional branches: target address calculation and target fetch. In order to sustain a steady rate of multiple instruction issue per clock the system must also be capable of fetching multiple instructions at a time, to avoid *starvation.* This is easily done with a wide memory (cache) interface when the instructions to be fetched form a contiguous block of code with a known starting address. This is not the case when a branch is encountered.

For example, suppose a system is capable of simultaneously fetching six contiguous instructions and the third instruction in a group happens to be an unconditional branch. This means the last three instructions are incorrectly fetched and would need to be refetched after calculating the branch target. Compile time preprocessing can be used to alleviate the situation. A technique known as target copying is used to modify the instruction sequence by copying several lines of code from the branch target to the address locations immediately following the branch instruction. In the preceding

example, the compiler can copy three lines of code from the branch target to the locations right after the branch instruction. As a result, all the six instructions fetched simultaneously from contiguous locations are valid. This further allows the overlap of target address calculation with processing of the instructions copied from the target, so that contiguous fetches can continue without any interruption from the target address.

A similar approach can also be taken for conditional branches using static branch prediction techniques and target copying for branches predicted to be taken. A drawback of this technique is the resulting code expansion. Smith, et. al. [SJH89] and Johnson [Joh91] report a variety of experiments exploring the impact of such instruction fetch limitations on potential speedup using dynamic detection and scheduling techniques. Instruction fetch inefficiencies caused by branch delays and instruction misalignment are reported to be the primary performance impediments.

Operand fetch Limitations. A typical instruction requires more than one operand. A steady rate of, say, x instruction executions per clock can only be supported if the system is also capable of supplying operands in excess of x per clock (more likely $2x$ operands per clock). Unlike instruction fetches, operand fetches are from non-contiguous memory locations and a significant number of operand fetches refer to recently computed results still residing in a local register file. The approaches taken to address this problem either at the main memory interface or at the register file interface fall into two categories: multi-ported shared bank or single-ported multiple banks. A bank can be either a main memory (or cache) module or a register file. A multi-ported shared bank has better utilization than single-ported multiple banks, but is normally a costlier implementation. This tradeoff has been studied in detail for evaluating alternatives for a shared memory implementation for a multiprocessor system and a recent study of this tradeoff apllied to register file implementation appears in [SoV89].

Johnson [Joh91] provides a quantitative comparison for four major hardware features for exploiting instruction level parallelism at the assembly level: out-of-order execution, register renaming, branch prediction, and a four-instruction decoder. The conclusions are derived from trace driven simulation in a general purpose environment. The incremental speedup due to out-of-order execution, given the other three features, is found to be in the range of 1.5; that due to register renaming and branch prediction is reported in the range of 1.3. These features are interdependent. As a result, these incremental speedups can not be considered in isolation.

1.2.5 Summary

This section surveys the wide variety of options available for representing, detecting, and scheduling concurrent instructions. Data structures for concurrency representation including dependency graphs, ordering matrices, and petri nets were

described. Dependencies are grouped into data dependency, control dependency and resource dependency. The available design choices have been classified in categories related to the scope (within or beyond basic blocks), level (high level, assembly, or microcode level), time (compile time or run time) and order (in-order or out-of-order) of detection, scheduling, and completion (Figure 1.3). Run time scheduling techniques are further classified into centralized control flow and distributed data flow approaches. This set of available options is illustrated in Figure 1.4. Certain important implementation tradeoffs were analyzed, including those related to interrupt handling and to instruction and operand fetch limitations.

The data presented in Table 1.1 show a broad range of potential speedup; however, if the reports of Riseman and Foster [RiF72] and Nicolau and Fisher [NiF84] are excluded, the picture is more limited. These two studies can be interpreted as performance limits without implementation constraints. Exclusive of these two reports, the reported speedup speedup varies from about 1.8 to 8 times that of execution on conventional pipelined systems. The results of some of these studies can be summarized as follows:

In-order versus out-of-order execution. The advantage of out of order execution is strongly dependent on the depth of the execution pipeline. The longer the delay in executing various operations, the more significant the advantage of out of order execution. If all instructions execute in unit time and they are decoded every time unit in order, then there will be no advantage to their execution out of order. The longer the delay in execution, the more the advantage realized in the additional overlap provided by their out of order execution. In other words, deeper pipelines are more effective in utilizing the additional parallelism exposed by out-of-order execution. A similar observation is made by Sohi and Vajapeyam [SoV89], in a somewhat different context. They report the effectiveness of deeper pipelines in utilizing the parallelism exposed by loop unrolling or multiple operation issue. Based on reported results [AKT86, SoV87], a performance plot of the type shown in Figure 1.5 can be expected. Out-of-order execution when applied to a pipeline of depth about three achieves roughly 20 percent speedup relative to machines with same pipeline latency and in-order execution. A pipeline as deep as 8 or 10 stages may achieve a relative speedup as high as 50 percent. A deeper pipeline has disadvantages associated with its poorer utilization due to longer flush times during branches. As a result, the low latency processor will still have superior performance speedup.

Multiple instruction issue with out-of-order execution. In this situation there are two variables: the number of instructions that are inspected for independence during the decode stage, and the number of detected independent instructions that actually can be issued for execution. Again, the results of Acosta [AKT86] seem representative. Figure

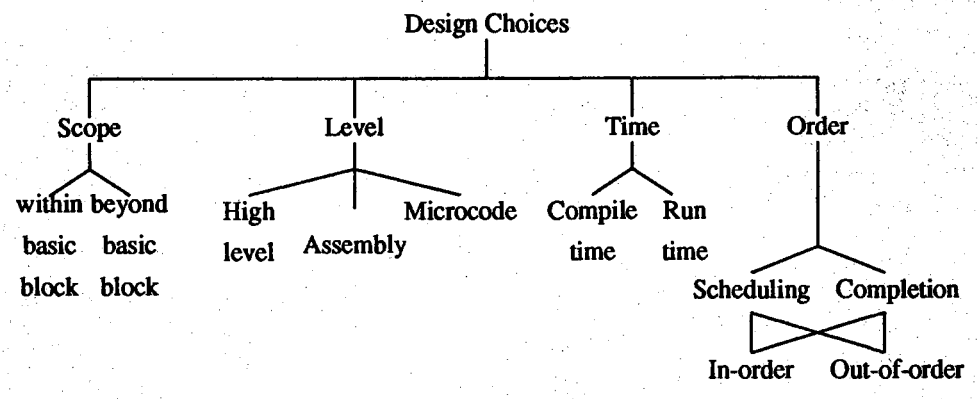


Figure 1.3 Available design choices for superscalar processors.

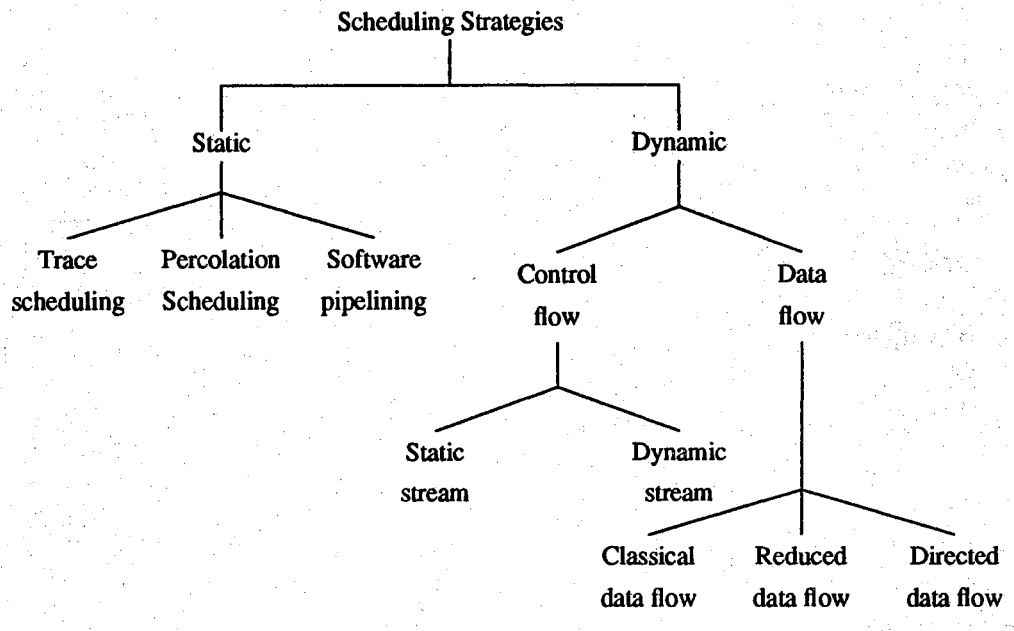


Figure 1.4 Classification of scheduling strategies.

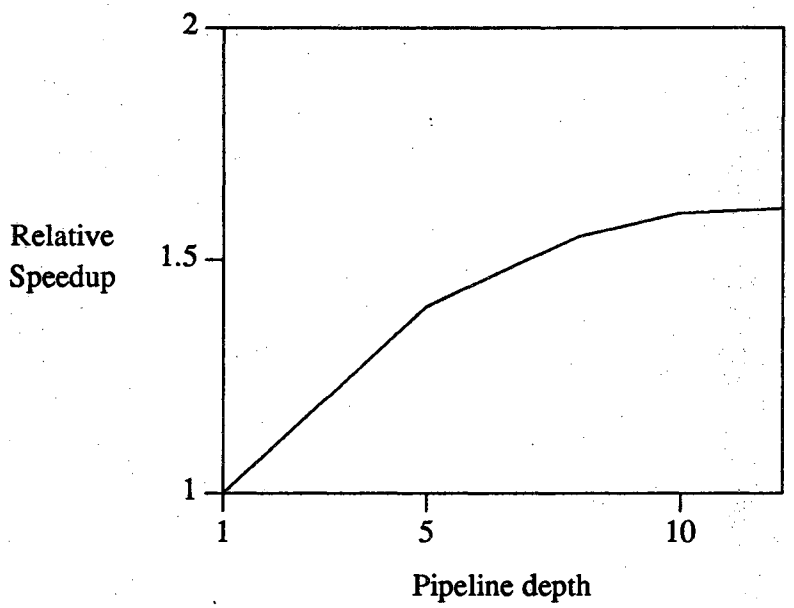
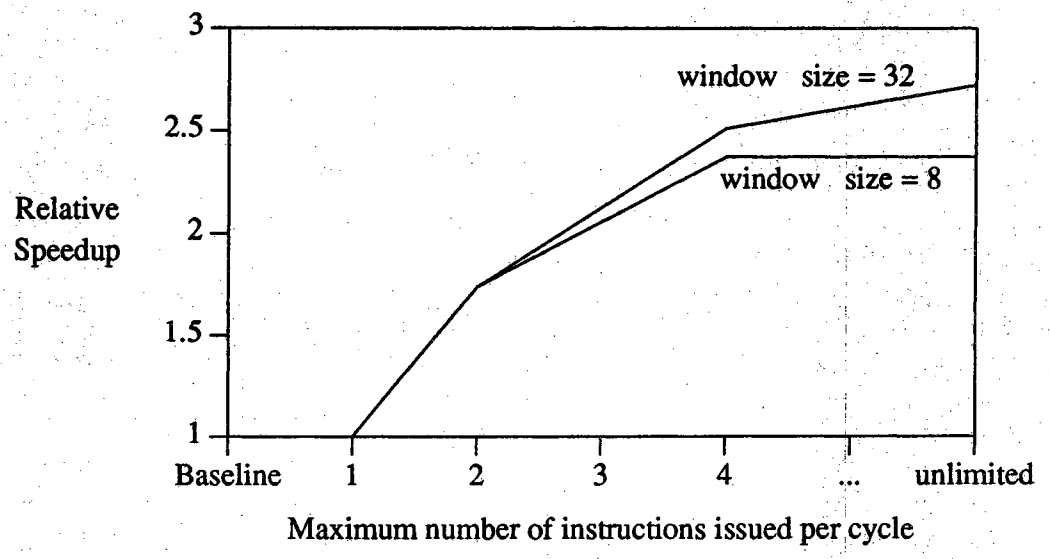


Figure 1.5 Speedup from out-of-order execution relative to in-order execution as a function of pipeline depth.

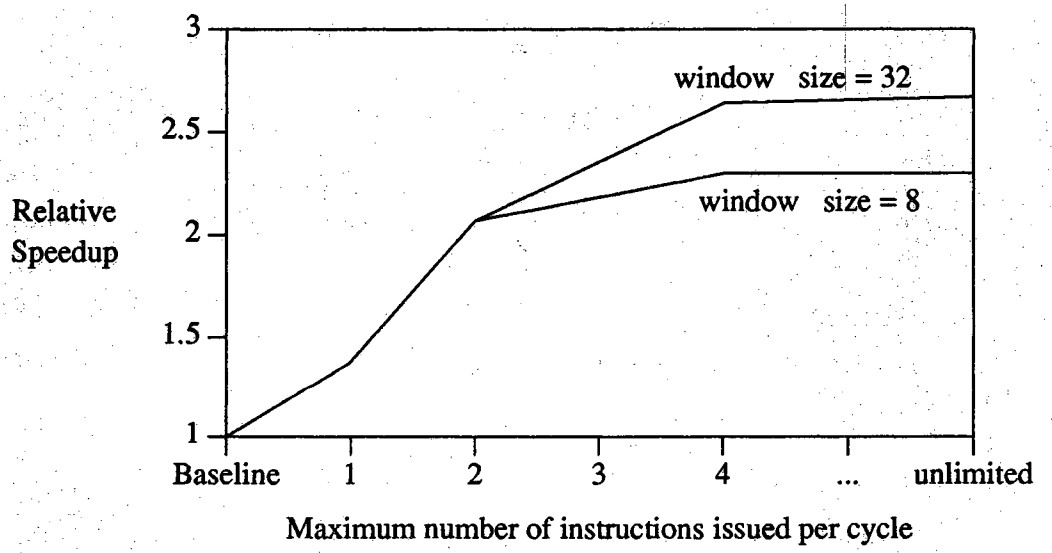
1.6.a shows the maximum available speedup given an ideal processor. For the ideal processor the baseline execution (speedup = 1) represents either in-order or out-of-order execution, since all the execution units execute with a unit delay. Issuing two instructions indicates a potential speedup of slightly less than 1.75 over the baseline. Issuing four instructions provides almost 2.5 speedup. The limit of the speedup is 2.8. Note that potentially almost all of the advantage is gained by inspecting eight instructions and issuing four. For Acosta [AKT86], the speedup is limited by basic block size, because scope of concurrency detection is limited to within the basic block. This is generally consistent with most earlier studies showing maximum speedup potential somewhere in the range of 1.5 to 3. When more reasonable hardware constraints are placed on the processor model [AKT86], as shown in Figure 1.6.b, the relative speedups remain much the same as ideal. However, some of the relative advantages shift. Now, whenever limited to single instruction issue per clock, out-of-order issue and execution achieves a 40 percent performance improvement over the baseline case of in-order issue and execution. An intermediate performance point (not shown in the figure) would be sequential issue with out-of-order completion, as in the CRAY-1. Speedup in this case is typically around 20 percent [AKT86]. Still, the overall speedup potential is limited to about 2.5 and most of the gain is again achievable with an instruction window of about eight instructions.

Software Assistance. Achieving significant speedup requires techniques that allow concurrency detection beyond the basic block. This can be done in hardware, in software, or both. Hardware alone, because of the complexity involved, seems to have limited potential. Using combinations of hardware and software techniques, it may be possible to achieve speedups of four to eight times [Wed82]. Parallelism and speedup uncovered by software duplicates in part the parallelism uncovered by the hardware. In one experiment in this area, Wedig [Wed82] reports an overall speedup in hardware plus software detection of concurrency of three, but the hardware or software alone would have accomplished a speedup of two. Thus, software detection of concurrency is potentially complementary to hardware, but overlap is present. The system designer should carefully partition the problem of concurrency detection lest duplicate effort detect the same events.

Finally, branches pose a significant bottleneck to concurrent execution. Future research needs to be directed to compile time and run time effort to reduce the branch overhead and to implementations that can simultaneously resolve multiple branches to independent targets. Memory system design would also be considerably affected by concurrent execution. The discussion in this chapter has ignored the details of the memory system enhancements essential to sustaining instruction and operand throughput. To map the simultaneous memory requests to independent memory banks would be vital to achieving overall performance improvement. On the other hand, loops (especially for



(a)



(b)

Figure 1.6 Multiple instruction issue with out-of-order execution and with scope limited to within the basic block, relative to a processor with in-order execution and single instruction issue per cycle; assuming single-cycle functional unit processor (a) and multiple-cycle functional unit processor (b). These graphs are derived from results reported in [AKT86].

scientific applications) hold significant speedup potential due to their regular dependency and control structure. A combination of complementary compile time and run time support may be the key to concurrency extraction and for resolving most of the performance barriers.

1.3 Dissertation Overview

The research described in subsequent chapters assumes a common architectural framework shown in Figure 1.7. Under ideal conditions, the organization is capable of providing a throughput of k results per cycle. For $k=1$, the system reduces to the classical single pipeline architecture. Chapter 2 concentrates on optimizing the performance of a single pipeline. Optimization of a pipeline here refers to partitioning the pipeline into the number of segments such that overall throughput is maximized. Architectures that opt for deeper single pipeline as opposed to multiple pipelines have been referred to as *superpipeline* architectures. Branches pose a serious performance bottleneck for such pipelined machines, as they interrupt the sequential flow of the instruction stream. Chapter 3 builds a common analytical platform for comparing different branch strategies in use for single-pipeline processors. Commonly-used branch strategies reduce the branch delay by predicting a certain execution path and continue to fetch along the predicted path. In case of incorrect prediction, the instructions fetched along the predicted path are wasted. Chapter 3 also examines this associated cost of wasted instruction fetches. Chapter 4 considers the cost-performance tradeoffs between the superscalar and superpipeline architectures. Performance of superscalars is critically dependent on the utilization of the multiple resources. The essential and control dependencies in the instruction stream are the primary limiting factor against the perfect utilization of the k pipelines. Chapter 5 proposes an analytical model for these program dependencies. The inputs to the proposed model also provide characterization of the inherent parallelism in program traces. A number of benchmarks are characterized using the Multiflow TRACE compiler. This characterization is used for predicting the attainable speedup under resource and scope constraints. The predicted speedup is close to the actual measured throughput of the compiler generated traces. Chapter 6 discusses a simplified extension of the model to include multiprocessors. The extended model is used to analyze combined systems, such as a superpipelined multiprocessor and a superscalar multiprocessor. Chapter 7 summarizes the work and the contributions of dissertation. Finally, Chapter 8 outlines some ideas for future work in this area.

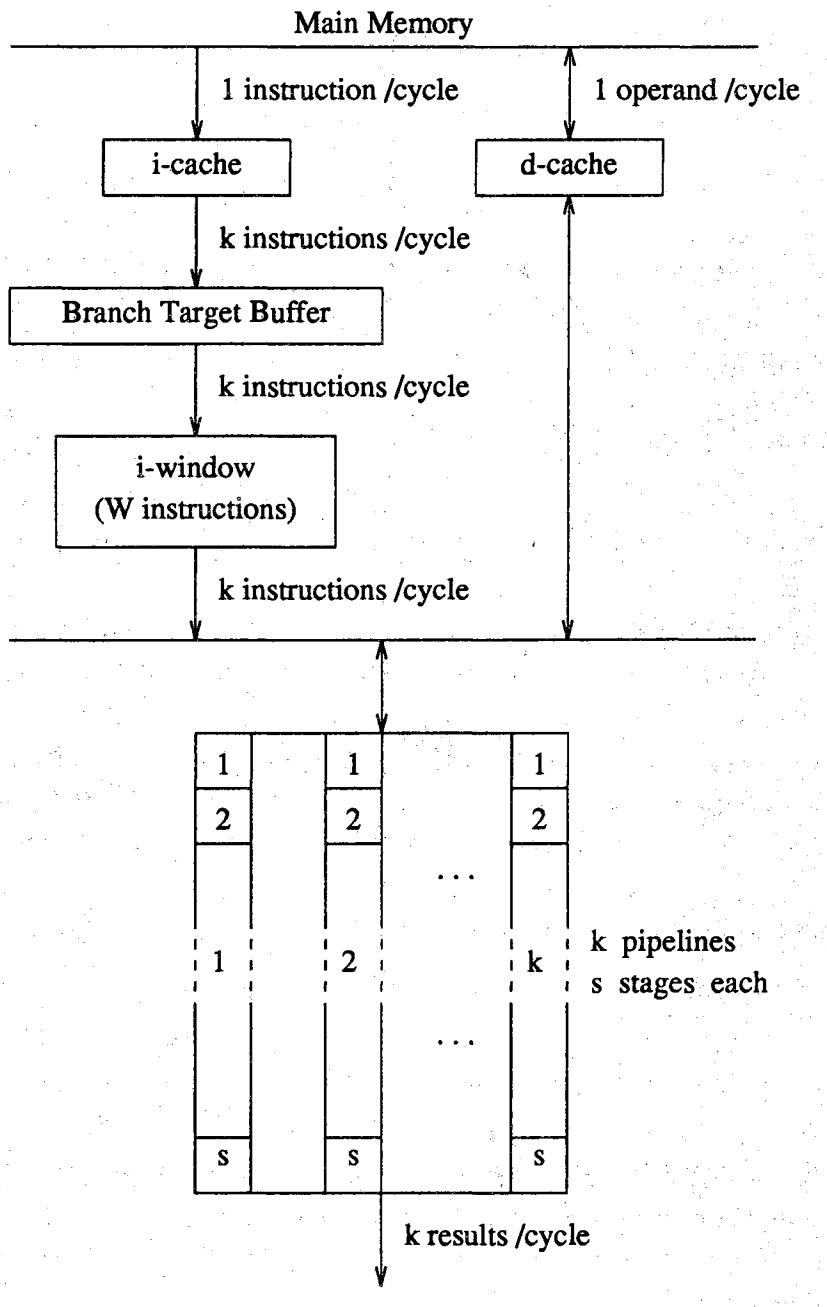


Figure 1.7 Architectural framework used for this research

CHAPTER 2
OPTIMAL PIPELINING

2.1 Introduction

Pipelining is one of the most attractive and widely used design alternatives in high-speed computer systems as it offers a potential speedup of s when s pipeline stages are used. This chapter is an attempt to understand the tradeoffs and overhead that limit this theoretical speedup. A mathematical model is developed to provide insight into the effective roles played by different parameters involved.

The following are the main practical constraints that limit the performance of pipelined processors:

i) *Instruction dependencies.* An instruction may be dependent on previous instructions for either data or control. This may cause less than full utilization of the pipeline.

ii) *Resource conflicts.* An instruction may require the use of a certain pipeline resource during the same period as an earlier instruction; thus necessitating a delay in its start. This can also limit utilization of the pipeline [KuS86].

iii) *Latch overhead.* This places some constraints on the maximum clock frequency that can be used. There are three main components of this overhead [KuS86]:

   a) propagation delay through the latch,

   b) data skew resulting from the difference between the minimum and maximum signal propagation times through various logic paths, and

   c) clock skew between the different stages of the pipeline.

iv) *Partitioning overhead.* A pipeline stage must consist of an integer number of gate levels, hence the propagation delay of a pipeline stage is quantized, which may reduce the maximum clock frequency used for the entire pipeline.

v) *Setup, or flush time, overhead.* The larger the pipeline the more the time required to fill it and flush it. This time can have a significant effect on the overall throughput. Note that apart from the initial setup time, additional flushes result from instruction dependency.

vi) *Control path limitations.* The time required to generate control signals for the pipeline stages also determines a minimum data path delay within any pipeline segment [KuS86].

The above constraints may be typed as those that limit the full utilization of the pipeline and those that limit the maximum clock frequency. Besides the constraints mentioned above, insufficient utilization of a pipeline can also result from not having enough data to keep the pipeline full. Such a restriction arises frequently in systems where full utilization of a computational resource is limited by, for example, insufficient I/O bandwidth.

2.1.1 Previous Research

A significant body of work has been reported on detecting pipeline hazards, resulting from instruction dependency or resource conflict, and optimal scheduling [Sha77, TjF70]. However, most of these studies assumed no restriction on clock period. In the area of latch timing, Cotten [Cot65] and Hallin and Flynn [HaF72] developed some basic latch timing constraints. Hallin and Flynn's work was extended by Fawcett [Faw75]. Kunkel and Smith [KuS86] further analyzed Fawcett's constraints and also provided some CRAY-1S simulation results to illustrate the effect of different overheads. They simulated the specific case of the polarity-hold latch with a single-phase clock.

The following latch timing constraints [Faw75, HaF72] form the basis for analyzing and modelling the latch overhead:

i) Minimum clock high time: The clock pulse must be wide enough to ensure that valid data is latched,

ii) Maximum clock high time: The clock pulse must be shorter than the minimum propagation delay from the input of one latch to the input of the next, and

iii) Minimum clock period: The minimum clock period must be longer than the maximum propagation delay from the input of one latch to the input of the next, to ensure valid data is latched.

Kunkel and Smith [KuS86] begin with performance measurements assuming no latch overhead. Next, they include the data skew component of latch overhead. Finally, clock skew is incorporated, first assuming two-level fanout and then assuming four-level fanout circuitry. In each case scalar, vector, and combined loops are used as the three kinds of inputs. Based on these results they conclude that 8 to 10 levels of gate delay per segment yields optimum, combined (scalar and vector) performance.

Kunkel and Smith do not provide much insight into the factors governing the nature of the performance curves, i.e., the reasons behind certain characteristics displayed by the

performance measurements. This omission provides the motivation for this chapter:

"Can a theoretical model be developed that will include different overheads associated with a generic pipeline and provide insights which will help predict the nature of modulations in the performance curve and the optimal performance?"

The next section presents a theoretical model aimed at better understanding of the behavior of a single-pipeline architecture.

2.2 A Generic Model

Let T be the latency of a logic tree without any pipelining. If the tree is divided into s segments, without considering any kind of overhead, the clock period with pipelining is

$$\Delta t = T/s \ .$$

Considering full utilization, throughput G is

$$G = 1/\Delta t \ .$$

Pipeline utilization can be quantified in terms of a utilization parameter, u defined as

$$u = s_{av}/s \ ,$$

where s_{av} is the average number of segments active at a time.

Thus, $u = 0$ for unutilized pipelines and $u = 1$ for fully utilized pipelines. Therefore, actual throughput can be written

$$G = u/\Delta t \ . \qquad \{2.1\}$$

Equation (2.1) represents the effect of pipeline limitations that result in inefficient utilization of the pipeline.

The actual clock period would not simply be inversely proportional to the number of segments, but rather involve certain overhead components. Pipeline overheads can be grouped into the following two categories:

a) *Static overhead* (c): This overhead is associated with each pipeline stage and is independent of the number of partitions of the pipeline. Propagation delay overhead (through the stage latch) and the clock-skew overhead fall in this category.

b) *Dynamic overhead* (κ): This overhead is a function of the number of partitions of the pipeline, i.e., it is a function of s. Data skew overhead, setup and flush time overhead, and partitioning overhead belong to this category.

With static and dynamic overheads included clock period, Δt becomes

$$\Delta t = \kappa (T/s) + c \ . \qquad \{2.2\}$$

Under ideal conditions, i.e., without any overhead, $\kappa = 1$ and $c = 0$. For example, consider some of the timing constraints developed by Kunkel and Smith [KuS86] on the basis of earlier work in this area by Fawcett [HaF72]. Assuming polarity-hold latches and a single-phase clock, after satisfying the constraints mentioned in Section 2.1.1, the minimum clock period with pipelining, as derived in [KuS86], is

$$\Delta t = (n+2)\, t_{\max} \ , \qquad \{2.3\}$$

where n is the number of gate levels between latches (excluding two levels of gate delay in the latch itself) and $t_{\max}$ is the maximum gate delay.

Using the terminology developed in this section, Equation (2.3) can be written as

$$\Delta t = T/s + 2\, t_{\max}$$

after separating the constant overhead term. In order to satisfy the lower bound on clock period (third constraint in Section 2.1.1), there must be a minimum number of gate levels between latches for proper operation. Thus, if s is large, delay padding may be required. Again, repeating the result derived by Kunkel and Smith [KuS86], assuming wire-delay-padding, the clock period in this range is

$$\Delta t = (1-\mu)\, T/s + (6-4\mu)\, t_{\max} \ , \qquad \{2.4\}$$

where μ is the ratio of minimum gate delay to maximum gate delay. This indicates a dynamic overhead, $\kappa = 1-\mu$, and a static overhead, $c = 6-4\mu$. Interestingly, κ is less than 1 in this example. Recall that Equation (2.4) is valid only when using delay padding to satisfy the constraint on the minimum number of gate levels between latches. Since s is typically large in this circumstance, any apparent reduction in Δt due to reduced κ is more than offset by a larger constant overhead term c, as compared to Equation (2.2) under ideal conditions.

Combining Equations (2.1) and (2.2),

$$G = \frac{u}{\kappa (T/s) + c} \ . \qquad \{2.5\}$$

The number of segments which maximizes the throughput can be obtained by solving,

$$\frac{\partial G}{\partial s} = 0 = c\, \dot{u}\, s^2 + \kappa T\, \dot{u}\, s + u\, \kappa T \ , \qquad \{2.6\}$$

where $\dot{u}$ is the first order derivative of u with respect to s. The above equation does not presume any specific utilization pattern. Hence, it can be used for any known utilization pattern.

Clearly pipeline utilization is a function of the number of pipeline segments. In only the simplest problem, a linear function $u = b - as$ (where a and b are arbitrary

constants) can be expected. Normally, shorter pipelines are easily filled and hence result in higher utilization. As the number of segments starts to go up, utilization starts to drop in a nonlinear manner. There is an upper limit to pipeline utilization independent of the number of segments which can be set, for example, by the maximum memory bandwidth. This maximum utilization, $u_{\max}$, is independent of s. As loading in a program environment is likely to cause at least a second order term, in this chapter a second order utilization pattern is assumed for the purpose of simulation, thus,

$$u = u_{\max} - r s^2 - vs \ . \qquad \{2.7\}$$

Therefore,

$$\dot{u} = -2rs - v \ , \qquad \{2.8\}$$

where the coefficients r and v are constants for any given program environment. These can be empirically determined and depend upon the amount of vectorization and instruction dependency in a given program, in addition to other factors. In Equation (2.7), r represents the effect of increasing dependency and issuing delay between instructions. For example, a two segment pipeline can only have a single stage dependency but with increasing number of segments, utilization would tend to drop due to increased dependency. Variable v represents the first order coefficient in the utilization model.

This is one of the simpler possible models for program utilization. The second order equation has been chosen only so that, as the number of segments changes, the utilization changes at a varying rate. Any equation of order two or more can capture this effect. Alternative and more complicated approaches to modelling pipeline utilization are discussed in the following chapters.

Using Equations (2.7) and (2.8), Equation (2.6) can be simplified to,

$$e s^3 + f s^2 + g s + h = 0 \ , \qquad \{2.9\}$$

where

$$e = 2 r c$$

$$f = c v + 3 \kappa T r$$

$$g = 2 \kappa T v$$

$$h = -(u_{max}) \kappa T \ .$$

Equation (2.9) can be solved to obtain the optimal number of partitions, s_{opt} under different conditions of utilization and overhead parameters. This equation is used to generate performance tables (Tables 2.2 - 2.6) and corresponding graphs (Figures 2.1 - 2.5) to illustrate sensitivity with respect to each parameter. Utilization coefficients have been varied over a range such that the utilization given by Equation (2.7) is between 0

Table 2.1 Nomenclature and nominal values of model parameters.

Explanation	Symbol	Nominal value
Latency of the logic tree	T	100 ns
Nominal number of segments	s_{nom}	5
Throughput of the pipeline	G	
Optimum number of segments	s_{opt}	
Constant term in the utilization equation	$u_{\max}$	0.6
First-order coefficient in the utilization equation	v	0.0.01
Second-order coefficient in the utilization equation	r	0.004
Constant overhead term	c	10 ns
Dynamic overhead term	κ	1.3
Branch frequency	b	0.1
Two-clock instruction frequency	x	0.1
Throughput at nominal values of all parameters	$G_{nominal}$	
Throughput, normalized relative to $G_{nominal}$	G_{norm}	
Throughput at $s = s_{opt}$, normalized relative to $G_{nominal}$	G_{opt}	
Throughput at $s = s_{subop} = 1$, normalized relative to $G_{nominal}$	G_{subop}	
Throughput at $s = s_{ovrop} = 10$, normalized relative to $G_{nominal}$	G_{ovrop}	
Throughput at $s = s_{nom}$	G_{nom}	
Throughput gain at $s = s_{opt}$, relative to G_{nom}	ΔG_{opt}	
Throughput gain at $s = s_{subop} = 1$, relative to G_{nom}	ΔG_{subop}	
Throughput gain at $s = s_{ovrop} = 10$, relative to G_{nom}	ΔG_{ovrop}	

and 1.

A given partitioning of a pipeline can be considered sub-optimal or over-optimal depending on whether the number of segments in the pipeline is less than or more than the optimal number of segments, respectively. Performance measurements have been taken at sub-optimal, optimal, and over-optimal points. All the throughput measurements are normalized with respect to $G_{nominal}$, which represents the throughput at certain nominal values of *all* parameters, as listed in Table 2.1.

There are clearly other options for normalization. The chosen option is preferred assuming an interest in estimating throughput with respect to an existing (nominal) computer design. Under the nominal conditions here, static overhead is assumed to be about one tenth of the period without pipelining. Dynamic overhead is assumed to be 1.3, as compared to 1 in the ideal case. The assumed nominal values of the utilization coefficients result in about half utilization of the pipeline.

Suppose there is an existing pipeline design with a certain number of segments, s_{nom}, and corresponding throughput, G_{nom}. Now, if the number of partitions is changed to s with corresponding throughput G, then the throughput gain in moving from s_{nom} to s pipeline segments can be defined as the ratio

$$\Delta G = G/G_{nom} \quad . \qquad \{2.10\}$$

Tables 2.7 through 2.11 and the corresponding graphs, Figures 2.6 through 2.10 show the sensitivity of this gain as a function of overhead and utilization coefficients. All the gain calculations are with respect to a reference pipeline having number of segments $s_{nom} = 5$.

2.3 Inferences

The effects of the overhead and the utilization coefficients on the actual (normalized) throughput can be summarized as:

i) As the static overhead (c) increases, the optimum throughput (G_{opt}) and the optimum number of partitions (s_{opt}) decreases. From Figure 2.1, it can be seen that for small values of c, changes in c have a predominant effect. This indicates the possibility of dramatic change in the optimal throughput (G_{opt}), as well as optimal partitioning (s_{opt}), if a balanced clock (negligible unintended skew) is disturbed.

ii) As the dynamic overhead (κ) increases, G_{opt} decreases whereas, unlike the previous case, s_{opt} increases. From Figure 2.2, it can be seen that similar to the earlier case, a given $\Delta\kappa$ has more effect when κ is small than when it is large. In a typical system where κ is close to 1, and c is close to 0, on comparing Figures 2.1 and 2.2, it can be

Table 2.2 Normalized throughput (G_{norm}) versus static overhead (c).

Static overhead, c (ns)	Optimal number of segments, s_{opt}	Normalized throughput		
		G_{subop}	G_{opt}	G_{ovrop}
0	6.29	0.36	1.47	0.62
10	5.51	0.33	1.01	0.35
20	5.03	0.31	0.78	0.24
30	4.69	0.29	0.64	0.19
40	4.44	0.28	0.55	0.15
50	4.23	0.26	0.48	0.13
60	4.05	0.25	0.43	0.11
70	3.91	0.23	0.39	0.10
80	3.78	0.22	0.35	0.09
90	3.66	0.21	0.32	0.08
100	3.56	0.20	0.30	0.07

Table 2.3 Normalized throughput (G_{norm}) versus dynamic overhead (κ).

Dynamic overhead, κ	Optimal number of segments, s_{opt}	Normalized throughput		
		G_{subop}	G_{opt}	G_{ovrop}
1.00	5.34	0.43	1.20	0.40
1.10	5.40	0.39	1.13	0.38
1.20	5.46	0.36	1.07	0.36
1.30	5.51	0.33	1.01	0.35
1.40	5.55	0.31	0.96	0.33
1.50	5.59	0.29	0.91	0.32
1.60	5.62	0.28	0.87	0.31
1.70	5.65	0.26	0.83	0.30
1.80	5.68	0.25	0.79	0.29
1.90	5.71	0.23	0.76	0.28
2.00	5.73	0.22	0.73	0.27

Table 2.4 Normalized throughput (G_{norm}) versus constant term of the utilization model (u_{max}).

Constant term of the util. model, u_{max}	Optimal number of segments, s_{opt}	Normalized throughput		
		G_{subop}	G_{opt}	G_{ovrop}
1.00	7.08	0.56	2.06	1.74
0.95	6.91	0.53	1.92	1.57
0.90	6.73	0.51	1.78	1.39
0.85	6.54	0.48	1.64	1.22
0.80	6.35	0.45	1.51	1.04
0.75	6.15	0.42	1.38	0.87
0.70	5.95	0.39	1.25	0.70
0.65	5.73	0.36	1.13	0.52
0.60	5.51	0.33	1.01	0.35
0.55	5.27	0.31	0.89	0.17
0.50	5.02	0.28	0.78	0.00

Table 2.5 Normalized throughput (G_{norm}) versus first-order coefficient of the utilization model (v).

First-order coeff. of the util. model, v	Optimal number of segments, s_{opt}	Normalized throughput		
		G_{subop}	G_{opt}	G_{ovrop}
0.0010	6.09	0.34	1.14	0.66
0.0030	5.96	0.34	1.11	0.59
0.0050	5.82	0.34	1.08	0.52
0.0070	5.69	0.34	1.05	0.45
0.0090	5.57	0.34	1.02	0.38
0.0110	5.45	0.33	1.00	0.31
0.0130	5.33	0.33	0.97	0.24
0.0150	5.21	0.33	0.95	0.17
0.0170	5.09	0.33	0.92	0.10
0.0190	4.98	0.33	0.90	0.03

Table 2.6 Normalized throughput (G_{norm}) versus second-order coefficient of the utilization model (r).

Second-order coeff. of the util. model, r	Optimal number of segments, s_{opt}	Normalized throughput		
		G_{subop}	G_{opt}	G_{ovrop}
0.0005	11.03	0.34	1.57	1.57
0.0010	9.02	0.34	1.40	1.39
0.0015	7.90	0.34	1.29	1.22
0.0020	7.14	0.34	1.21	1.04
0.0025	6.58	0.34	1.15	0.87
0.0030	6.15	0.34	1.09	0.70
0.0035	5.80	0.34	1.05	0.52
0.0040	5.51	0.33	1.01	0.35
0.0045	5.26	0.33	0.97	0.17
0.0050	5.04	0.33	0.94	0.00

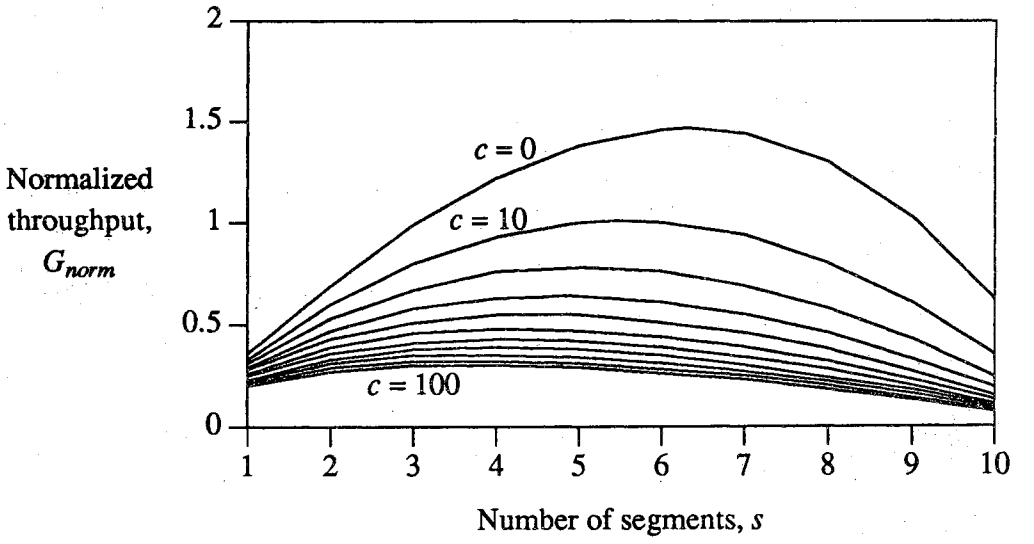


Figure 2.1 Normalized throughput (G_{norm}) versus static overhead (c).

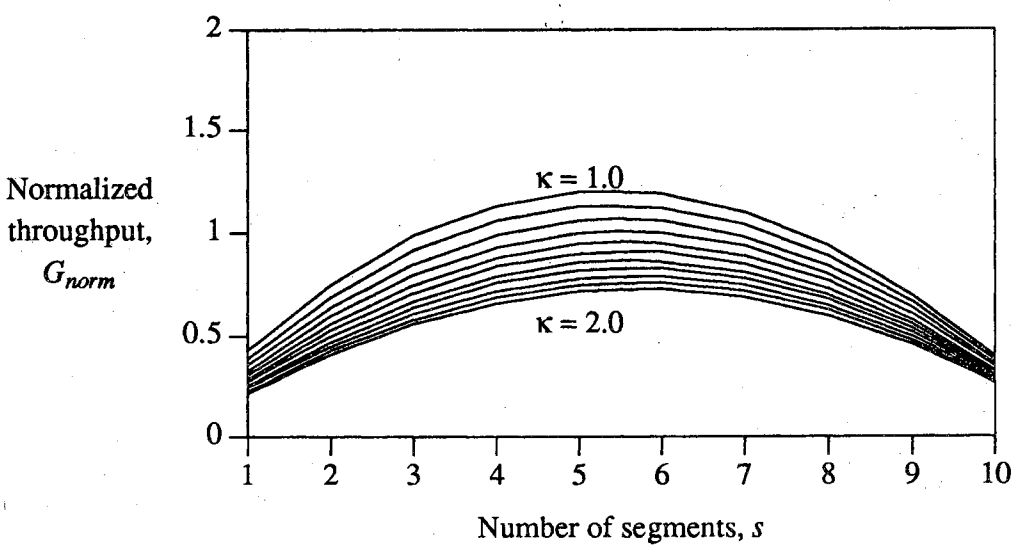


Figure 2.2 Normalized throughput (G_{norm}) versus dynamic overhead (κ).

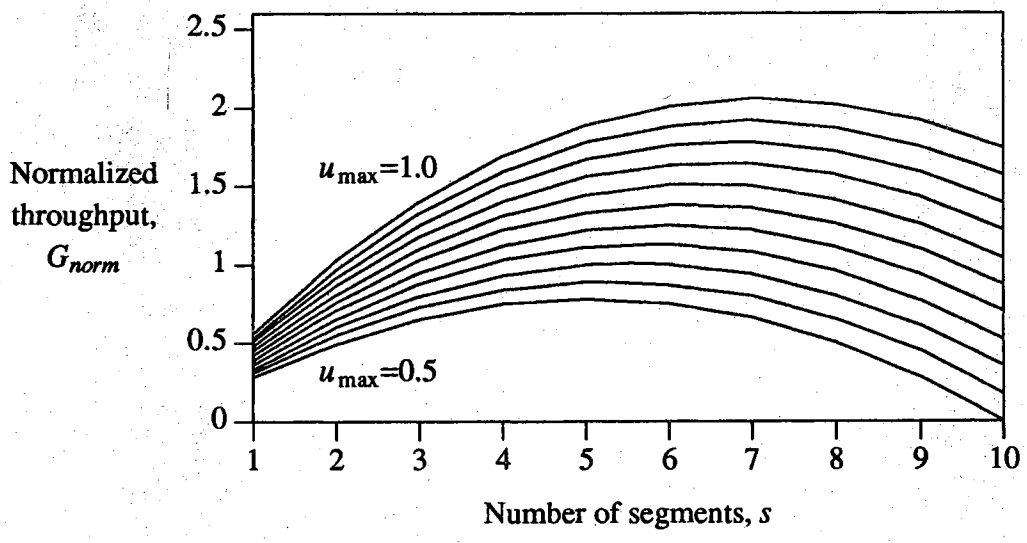


Figure 2.3 Normalized throughput (G_{norm}) versus constant term of the utilization model (u_{max}).

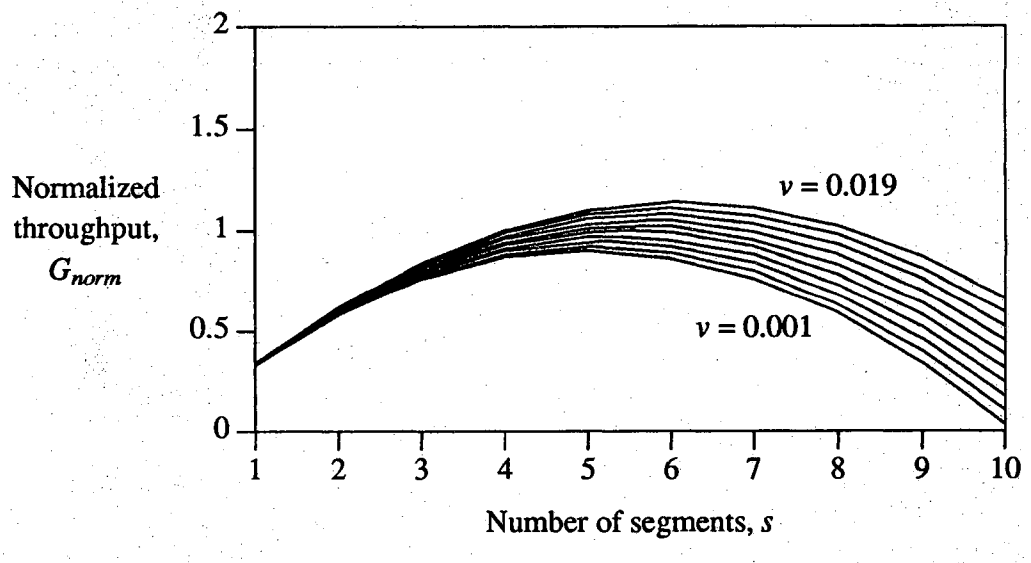


Figure 2.4 Normalized throughput (G_{norm}) versus first-order coefficient of the utilization model (v).

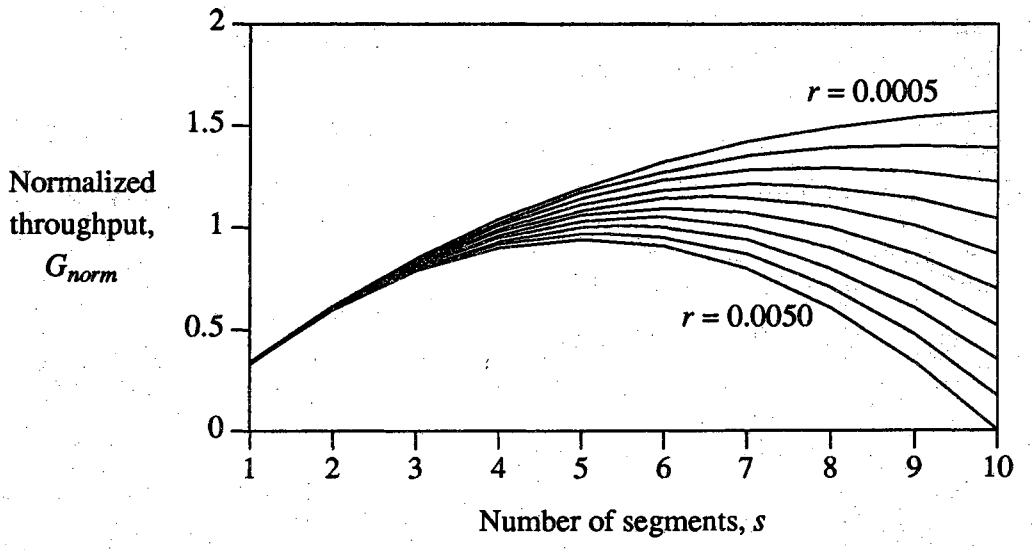


Figure 2.5 Normalized throughput (G_{norm}) versus second-order coefficient of the utilization model (r).

Table 2.7 Throughput gain (ΔG) versus static overhead (c).

Static overhead, c (ns)	Optimal number of segments, s_{opt}	Throughput gain		
		ΔG_{subop}	ΔG_{opt}	ΔG_{ovrop}
0	6.29	0.26	1.06	0.44
10	5.51	0.33	1.01	0.35
20	5.03	0.40	1.00	0.31
30	4.69	0.46	1.00	0.29
40	4.44	0.51	1.01	0.28
50	4.23	0.55	1.02	0.27
60	4.05	0.59	1.02	0.26
70	3.91	0.63	1.03	0.26
80	3.78	0.66	1.04	0.25
90	3.66	0.69	1.05	0.25
100	3.56	0.71	1.05	0.25

Table 2.8 Throughput gain (ΔG) versus dynamic overhead (κ).

Dynamic overhead, κ	Optimal number of segments, s_{opt}	Throughput gain		
		ΔG_{subop}	ΔG_{opt}	ΔG_{ovrop}
1.00	5.34	0.36	1.00	0.33
1.10	5.40	0.35	1.01	0.34
1.20	5.46	0.34	1.01	0.34
1.30	5.51	0.33	1.01	0.35
1.40	5.55	0.33	1.01	0.35
1.50	5.59	0.33	1.01	0.36
1.60	5.62	0.32	1.01	0.36
1.70	5.65	0.32	1.01	0.36
1.80	5.68	0.32	1.02	0.37
1.90	5.71	0.31	1.02	0.37
2.00	5.73	0.31	1.02	0.37

Table 2.9 Throughput gain (ΔG) versus constant term of the utilization model (u_{max}).

Constant term of the util. model, u_{max}	Optimal number of segments, s_{opt}	Throughput gain		
		ΔG_{subop}	ΔG_{opt}	ΔG_{ovrop}
1.00	7.08	0.30	1.09	0.92
0.95	6.91	0.30	1.08	0.88
0.90	6.73	0.30	1.07	0.83
0.85	6.54	0.31	1.06	0.78
0.80	6.35	0.31	1.05	0.72
0.75	6.15	0.32	1.04	0.65
0.70	5.95	0.32	1.03	0.57
0.65	5.73	0.33	1.02	0.47
0.60	5.51	0.33	1.01	0.35
0.55	5.27	0.34	1.00	0.20
0.50	5.02	0.36	1.00	0.00

Table 2.10 Throughput gain (ΔG) versus first-order coefficient of the utilization model (v).

First-order coeff. of the util. model, v	Optimal number of segments, s_{opt}	Throughput gain		
		ΔG_{subop}	ΔG_{opt}	ΔG_{ovrop}
0.0010	6.09	0.31	1.03	0.60
0.0030	5.96	0.31	1.03	0.55
0.0050	5.82	0.32	1.02	0.49
0.0070	5.69	0.33	1.02	0.44
0.0090	5.57	0.33	1.01	0.38
0.0110	5.45	0.34	1.01	0.32
0.0130	5.33	0.34	1.00	0.25
0.0150	5.21	0.35	1.00	0.18
0.0170	5.09	0.36	1.00	0.11
0.0190	4.98	0.37	1.00	0.04

Table 2.11 Throughput gain (ΔG) versus second-order coefficient of the utilization model (r).

Second-order coeff. of the util. model, r	Optimal number of segments, s_{opt}	Throughput gain		
		ΔG_{subop}	ΔG_{opt}	ΔG_{ovrop}
0.0005	11.03	0.28	1.32	1.31
0.0010	9.02	0.29	1.20	1.19
0.0015	7.90	0.30	1.13	1.07
0.0020	7.14	0.30	1.09	0.94
0.0025	6.58	0.31	1.06	0.80
0.0030	6.15	0.32	1.03	0.66
0.0035	5.80	0.33	1.02	0.51
0.0040	5.51	0.33	1.01	0.35
0.0045	5.26	0.34	1.00	0.18
0.0050	5.04	0.35	1.00	0.00

concluded that a given Δc would normally have stronger impact on system performance than an equivalent $\Delta\kappa$.

iii) As utilization (u) increases, G_{opt} as well as s_{opt} increase. A study of Figures 2.3 - 2.5 leads to the conclusion that: higher-order coefficients have a stronger effect on optimal partitioning (s_{opt}) and on the optimal throughput (G_{opt}) as compared to the lower-order coefficients. In other words, higher-order coefficients would require tighter control in order to maintain a certain level of performance. Also, for large s the slope of the performance curves (i.e., $\partial G/\partial s$) in Figure 2.3 becomes highly insensitive to the variations in $u_{\max}$, the constant term in the utilization model. A mathematical explanation for this, although not presented here, can be derived from the expression for $\partial G/\partial s$ given in a later section.

The following conclusions can be made from the gain plots:

i) As the static overhead (c) increases, gain increases for less than the existing nominal number of segments, (s_{nom}, see Table 2.1). For more than s_{nom} segments, gain decreases with increasing c (Figure 2.6). A gain can be considered an incentive if it is greater than 1. Thus, with increasing static overhead, there is higher incentive to modify an existing (reference) pipeline to have a lesser number of segments. This behavior is seen as a result of s_{opt} going down with increasing c.

ii) As the dynamic overhead (κ) increases, gain decreases for fewer than the reference number, s_{nom} of segments. For more than the existing number of segments, gain increases with increasing κ (Figure 2.7). Therefore, with increasing κ, there is less incentive to change an existing pipeline to a smaller number of segments and vice versa.

iii) As the utilization (u) increases, gain decreases for less than the existing number of segments. For more than the reference number of segments, gain increases with increasing utilization (Figures 2.8 - 2.10). Again, because s_{opt} increases with higher utilization, there is increasing incentive to redesign an existing pipeline to have a larger number of segments. Note that a variation in any of the utilization coefficients has a stronger performance impact on the system with large number of segments than the system with fewer segments.

An interesting but not obvious property of the throughput gain plots is the non-monotonicity of the optimal gain. Look at the plot for gain versus static overhead (Figure 2.6). It can be seen that the optimal (maximal) gain is at its minimum for a fixed static overhead, when c is at 20ns. Figure 2.11 shows the effect of utilization change on the optimal gain minima. If the utilization is increased, a shift of minima is noticed to when c is at 50ns.

Therefore, for a certain change in the static overhead c, say from c = 25ns to 45ns, scalar code (smaller utilization) can show an increase in optimal gain, whereas vector

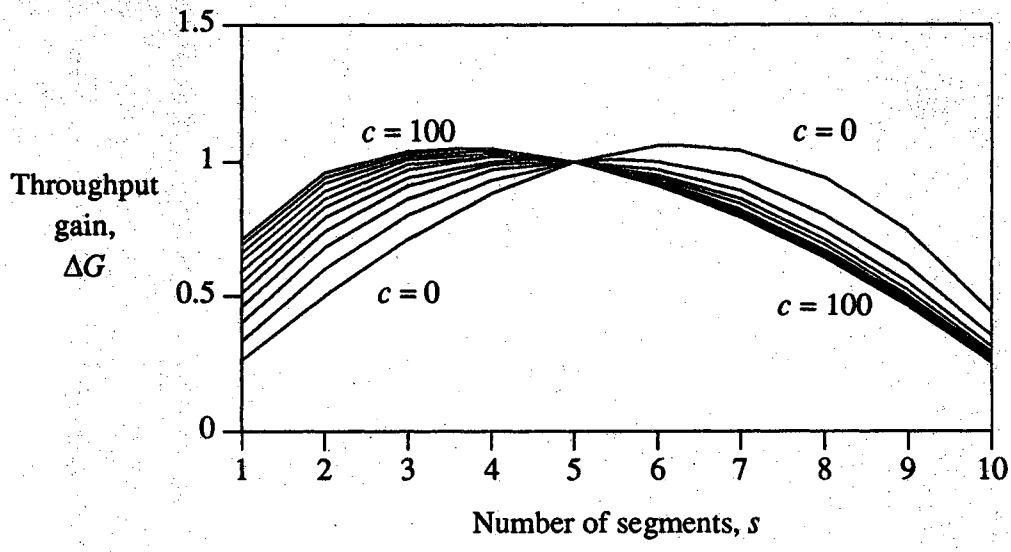


Figure 2.6 Throughput gain (ΔG) versus static overhead (c).

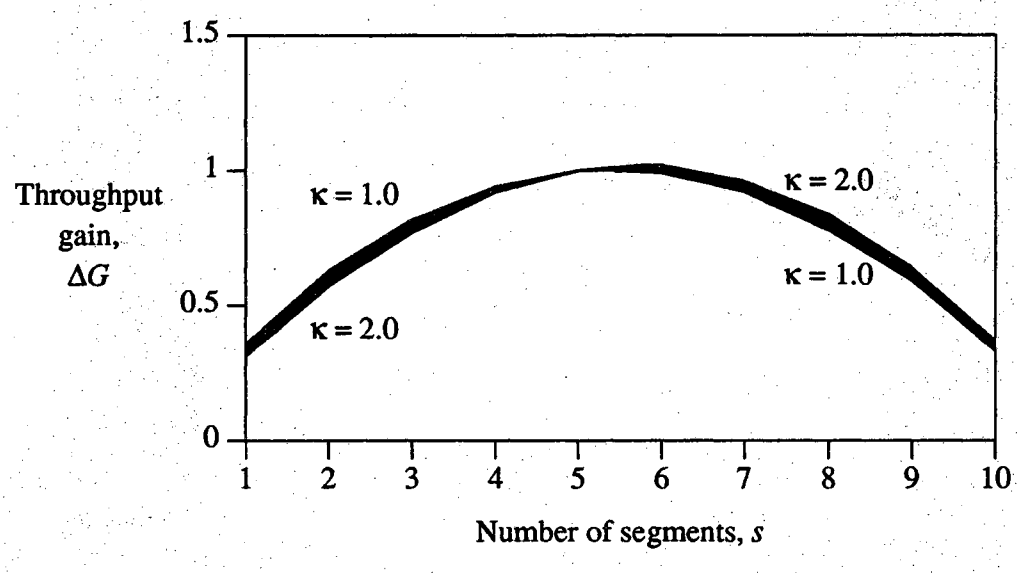


Figure 2.7 Throughput gain (ΔG) versus dynamic overhead (κ).

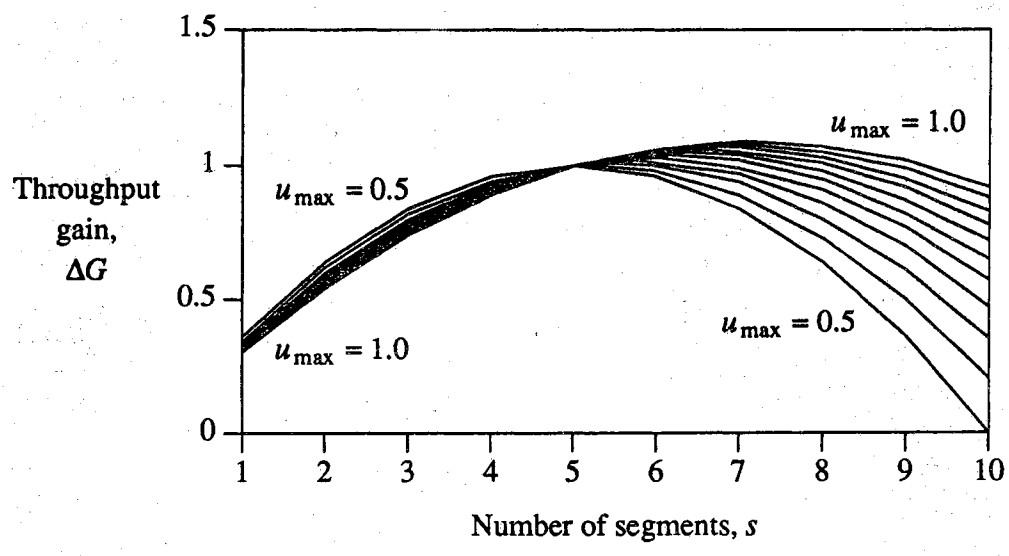


Figure 2.8 Throughput gain (ΔG) versus constant term of the utilization model (u_{max}).

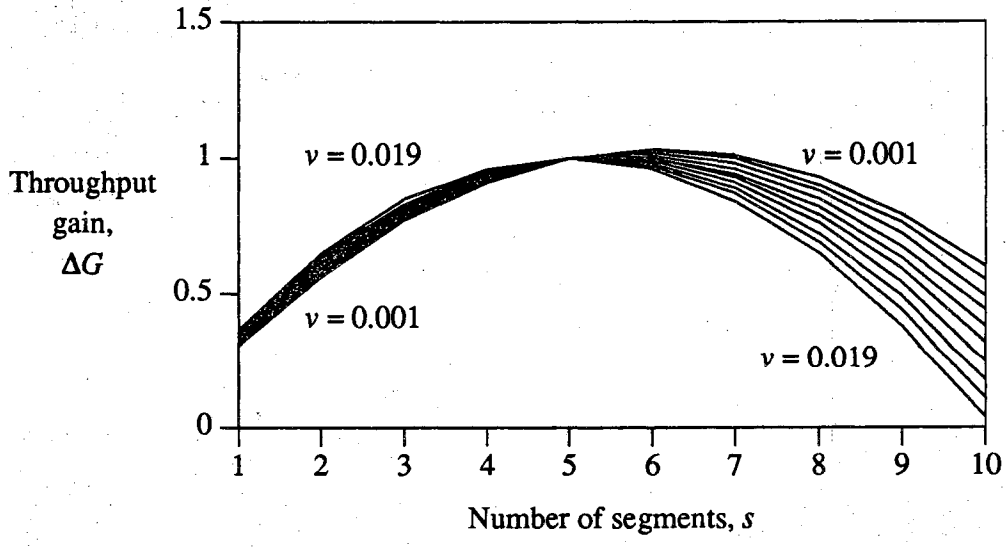


Figure 2.9 Throughput gain (ΔG) versus first-order coefficient of the utilization model (v).

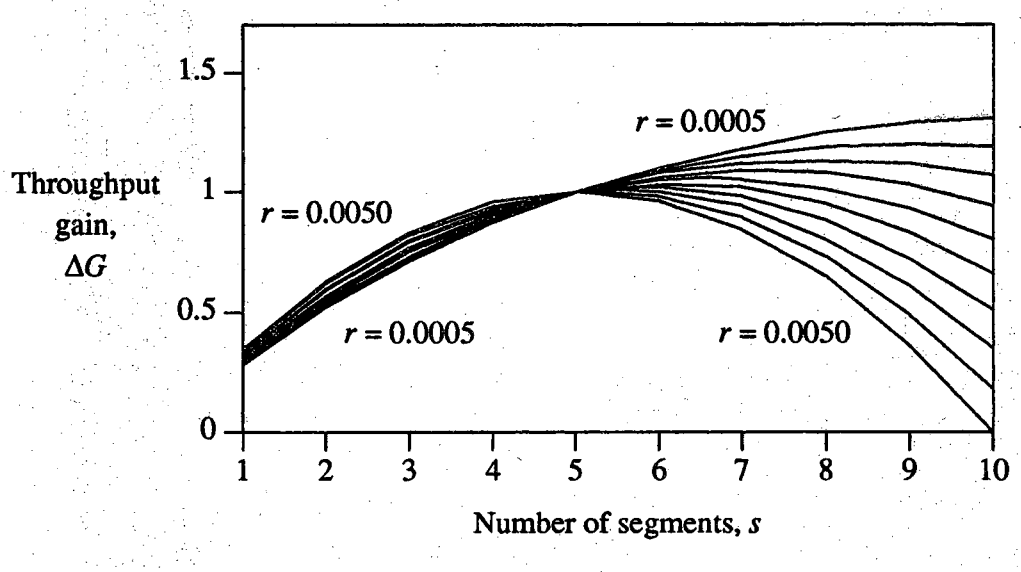


Figure 2.10 Throughput gain (ΔG) versus second-order coefficient of the utilization model (r).

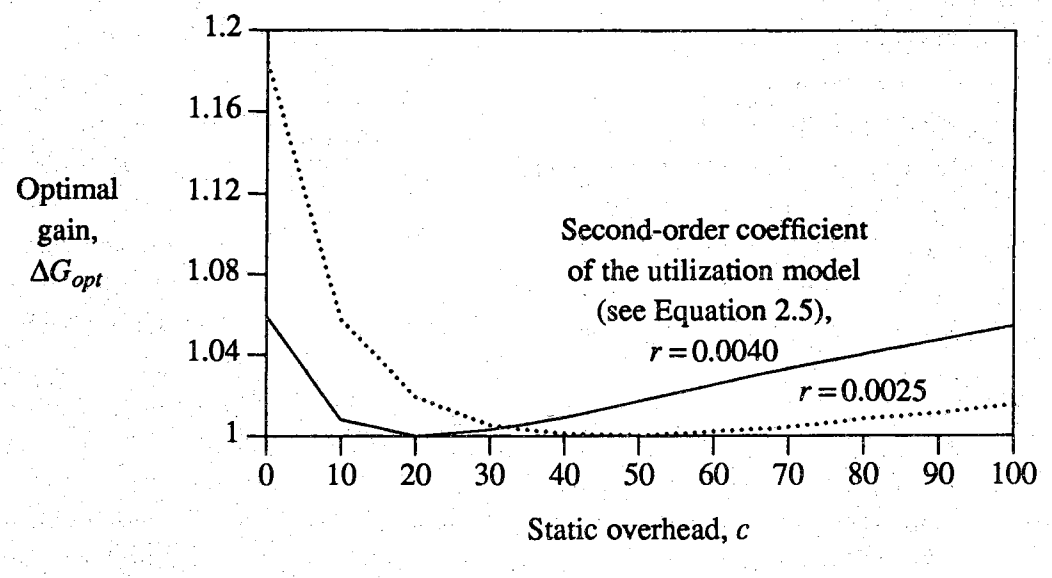


Figure 2.11 Optimal throughput gain (ΔG_{opt}) versus static overhead (c)

code (higher utilization) continues to show a decrease in the optimal gain. A similar effect is observed in Kunkel and Smith's simulation results, when as a result of moving from 2-level fanout clock skew overhead (i.e., a clock distribution logic with 2 levels of gate delay) to 4-level fanout clock skew overhead (in other words, moving to higher constant overhead), scalar optimal gain increases, whereas the vector optimal gain decreases. The optimal gain minima occurs when s_{opt} drops to s_{nom}. Since s_{opt} is greater for a vector environment, with an increase in c it drops to s_{nom} later than in the case of a scalar environment.

2.3.1 Correspondence with Previously Published Experimental Results

As illustrated in Section 2.2, the minimum clock period expressions in the Kunkel and Smith's paper [KuS86] can be rearranged to highlight static and dynamic overhead terms. In the range where s is large, inclusion of data skew overhead decreases the dynamic overhead (κ), while the static overhead (c) increases. The decrease in κ and the increase in c, both result in a reduced throughput gain reported by Kunkel and Smith. Also, there is dramatic reduction in actual throughput because of a change in c from zero to a positive non-zero value. The increase in static overhead (c) also reduces the optimum number of segments, s_{opt}. This reduction in s_{opt} becomes noticeable for the scalar code in their study. But for the vector code, because of higher utilization than the scalar code, s_{opt} still stays high enough to be unnoticed. Inclusion of clock skew overhead further increases the constant overhead (c) and hence, s_{opt} continues to move towards a smaller number of segments. With increasing static overhead, throughput gain continuously increases in the suboptimal region, whereas it decreases in the over-optimal region of all the performance tables obtained in [KuS86].

2.4 Potential Improvements to the Model

The definition of the utilization parameter, u, as given in relation to its role in Equation (2.1) best fits the case of pipelines where each segment always takes only one clock period to perform its operation. Such pipelines are typically at the subsystem level, e.g., a floating point multiplier pipeline. If pipelines have variable delay, where a segment may take more than one clock to complete, are included then u as defined earlier does not fit the need. For example, if every stage takes 2 clocks then though the utilization as defined may be 1 (or 100 percent), the throughput would only be 1 result every 2 clocks and not 1 result every clock, as given by Equation (2.1). Such pipelines are typically at the system level, e.g., an instruction fetch-decode-execute pipeline.

In Equation (2.1), in a more generic sense, pipeline utilization (u) should be thought of as the factor by which the maximum possible throughput ($1/\Delta t$) is modified to yield the actual throughput (G). Actual throughput is strictly determined by the rate at which the outputs are available from the last segment. If any data item takes more than one clock in the last segment, a decrease in throughput results. Similarly, if any data item takes more than one clock in any segment, say segment i, the effect of this slowdown of segment i will ripple through the pipeline and result in the same slowdown at the last segment, segment s, after ($s-i$) clocks. Assume that while the effect of slowdown of one stage is rippling to the final stage, there is no other stage that slows down. Under these conditions, the following equation provides a model for u,

$$u = \frac{1}{1+(s-1)b+\sum_{i=1}^{s}\sum_{j=2}^{J}(j-1)x_{i,j}} \qquad \{2.11\}$$

where b is the average number of setup and flush sequences per data item, J is the maximum number of cycles any data item spends in any segment, and $x_{i,j}$ is the probability that a data item takes j cycles in the ith segment.

For example, consider a 5-stage instruction pipeline in an environment such that an average of 1 out of every 10 instructions takes 2 clocks in the execution stage (last segment) and 1 out of every 10 instructions is a branch instruction. Then, $s=5$, $b=0.1$, $J=2$, $x_{5,2}=0.1$, and, $x_{1,2}=x_{2,2}=x_{3,2}=x_{4,2}=0$. This gives $u=10/15$. Any random sequence of 10 instructions would be expected to lose 4 clocks during a branch and 1 clock due to a 2-clock instruction and, hence, take 15 clocks.

For further analysis, assume a simplified view of an instruction pipeline such that $J=2$ and for any segment i, $x_{i,2}=x$. As an example, if the ith segment refers to the operand fetch stage, x refers to the fraction of instructions spending an additional clock during operand fetch. Typically, each stage would have its own independent fraction of instructions which occupy the stage for more than one clock; because this is not critical to the current discussion, additional variables are not introduced, From Equation (2.11)

$$u = \frac{1}{1+(s-1)b+sx} \ . \qquad \{2.12\}$$

Therefore, throughput can be written

$$G = \frac{1}{1+(s-1)b+sx} * \frac{1}{\kappa(T/s)+c} \ . \qquad \{2.13\}$$

For maximum throughput,

$$s_{opt} = \sqrt{\frac{(1-b)\kappa T}{(x+b)c}} \ . \qquad \{2.14\}$$

As static overhead (c) approaches zero, the optimum number of segments (s_{opt}) approaches infinity. As observed before, for small values, c has a dominant effect. Tables 2.12 and 2.13 and corresponding graphs, Figures 2.12 and 2.13, show the variation in normalized throughput as a function of the number of pipeline segments. As branch frequency (b) decreases, s_{opt} and corresponding optimal throughput both increase. In other words, for a fixed partitioning, the pipeline becomes suboptimal as b decreases. The same observation holds for segment slowdown frequency (x).

Effect of Buffering: So far, additional buffering at a segment output has been ignored. In the presence of such buffers, slowdown of an intermediate segment does not necessarily slow down the final segment. Although it is quite often used in system-level pipelines (e.g., instruction FIFO), its inclusion would considerably complicate the model. The solution to a general model of this type can be derived using queueing theory techniques.

Second-order Effects: The utilization model assumed in Section 2.2 also hides certain second-order details. For example consider the rate of change of throughput with respect to s. From Equations (2.1) and (2.2),

$$\frac{\partial G}{\partial s} = \frac{\dot{u}}{T} + \frac{u\kappa T}{s^2 \Delta t^2}. \qquad \{2.15\}$$

Considering the given utilization model, with increasing s, the first term in the Equation (2.15) becomes more and more negative, whereas the second term becomes less and less positive. In other words, dG/ds monotonically decreases with increasing s leading to diminishing return (reduced throughput improvement) with increasing s. In an actual environment this may not be the case. Utilization of a pipeline does not necessarily go down with an increasing number of segments. For example, if a larger number of partitions leads to better mapping of the reservation tables (i.e., better resource allocation), utilization might even go up. Also, for the same number of segments, utilization may change in a statistical and/or periodic fashion. If these possibilities are incorporated into the utilization model, throughput curves could potentially have multiple maxima and minima and there would be points of maximum and minimum return from incremental change in partitioning.

2.5 Summary

This chapter provides an approximate model of the behavior of a pipeline and the understanding of the factors involved in determining the optimal performance. In spite of its simplicity, the model can be considered a useful first-order tool for comparative study

Table 2.12 Normalized throughput (G_{norm}) versus branch frequency (b).

Branch frequency, b	Optimal number of segments, s_{opt}	Normalized throughput		
		G_{subop}	G_{opt}	G_{ovrop}
0.02	10.30	0.44	1.36	1.36
0.04	9.44	0.44	1.26	1.26
0.06	8.74	0.44	1.18	1.17
0.08	8.15	0.44	1.10	0.09
0.10	7.65	0.44	1.04	1.03
0.12	7.21	0.44	0.99	0.97
0.14	6.83	0.44	0.94	0.91
0.16	6.48	0.44	0.90	0.86
0.18	6.17	0.44	0.86	0.82
0.20	5.89	0.44	0.83	0.78

Table 2.13 Normalized throughput (G_{norm}) versus segment slowdown frequency (x).

Segment slowdown frequency, x	Optimal number of segments, s_{opt}	Normalized throughput		
		G_{subop}	G_{opt}	G_{ovrop}
0.02	9.87	0.48	1.42	1.42
0.04	9.14	0.47	1.30	1.29
0.06	8.55	0.46	1.20	1.19
0.08	8.06	0.45	1.11	1.10
0.10	7.65	0.44	1.04	1.03
0.12	7.29	0.44	0.98	0.96
0.14	6.98	0.43	0.93	0.90
0.16	6.71	0.42	0.88	0.85
0.18	6.46	0.41	0.84	0.80
0.20	6.24	0.41	0.80	0.76

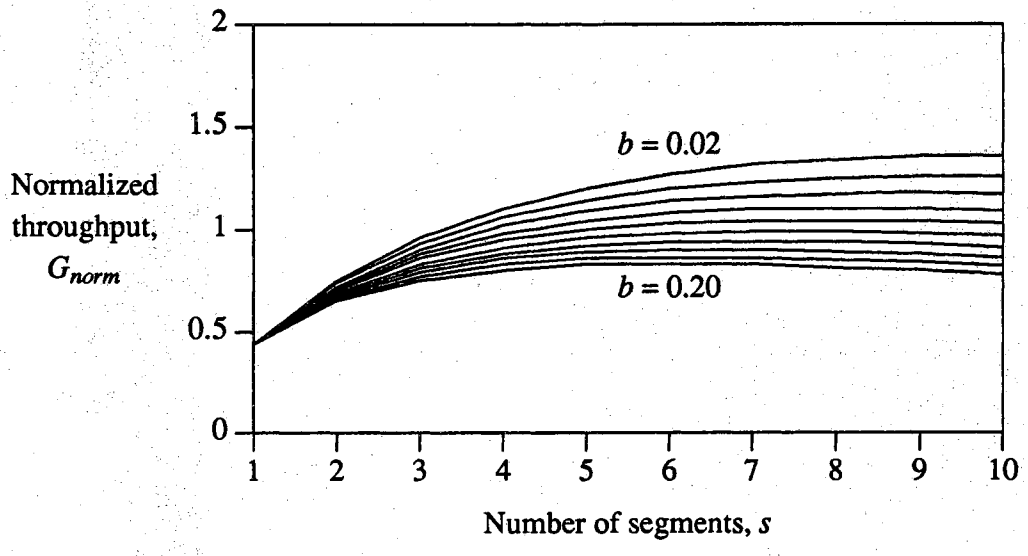


Figure 2.12 Normalized throughput (G_{norm}) versus branch frequency (b).

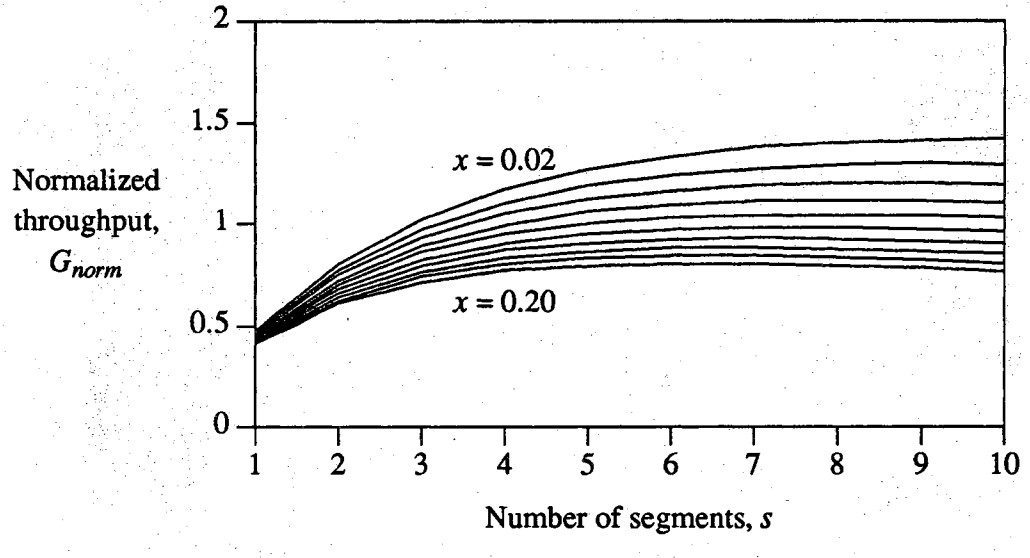


Figure 2.13 Normalized throughput (G_{norm}) versus segment slowdown frequency (x).

or sensitivity analysis of the performance of a pipeline in different environments with different overheads. Pipeline utilization models were presented for both sub-system as well as system level pipelines. Effects of branching and segment slowdown were also considered in the case of simple system-level pipelines.

Small changes in the constant overhead term were shown to have a large impact on optimal pipeline behavior. Increasing dynamic overhead increases the optimal number of segments, whereas increasing static overhead requires fewer segments for optimal performance. The results obtained are found to be in very close agreement with CRAY-1 simulation results obtained by Kunkel and Smith [KuS86], providing an analytical basis for their results as well as additional insight in the pipeline optimization problem.

It is fair to conclude at this point that there are constraints that limit the speedup attainable through a single pipeline. One way to move beyond the optimum throughput of a single pipeline may be by adding several such pipelines. Architectures adopting such an approach are referred to as superscalar architectures and they form the basis of discussion in the Chapter 4 that models multiple pipelines.

As alluded to in Section 2.4, branches pose a significant threat to high pipeline utilization. The drop in utilization due to the inability to fetch the instructions arising from the uncertainty due to conditional branches gets further magnified on systems with multiple pipelines. A wide variety of branch strategies have been proposed to reduce the branch delay. Next chapter analyzes these strategies through a probability based model in the context of single-pipeline systems. Chapters 4 and 5 extends this analysis to superscalars with speculative execution.

CHAPTER 3
BRANCH STRATEGIES: MODELLING AND OPTIMIZATION

3.1 Introduction

Instruction dependency introduced by conditional branch instructions, which are resolved only at run-time, can have a severe performance impact on pipelined machines. A variety of strategies are in wide use to minimize this impact. Additional instruction traffic generated by these branch strategies can also have an adverse effect on the system performance. Therefore, in addition to the likely reduction a branch prediction strategy offers in average branch delay, resulting excess instruction traffic can be an important parameter in evaluating overall strategy effectiveness. The objective of this chapter is two-fold: to develop a model for different approaches to the branch problem and to help select an optimal strategy after taking into account the additional instruction traffic generated by branch strategies. The first section presents the details of the model which also forms the basis of a new classification of the different branch strategies commonly employed. The following sections derive certain inferences from the results obtained and lead us to some hybrid strategies.

3.1.1 Previous Research

Throughput in a pipeline environment is obtained by overlapping different instructions in different stages of execution. This implies an ability to predict and issue successive instructions before the complete execution of a given instruction. Dependence of an instruction on the result of a predecessor instruction limits this ability. Tjaden and Flynn [TjF70] provide an early framework in the area of formalizing the concept of instruction dependency. The effect of conditional branches on system performance was further substantiated by Riseman and Foster [RiF72]. Interest in different branch strategies for minimizing performance impact has been renewed with the advent of new RISC machines. Most of the recent work in this area has concentrated on specific branch strategies and on improving prediction accuracy. Smith [Smi81] discusses in detail different strategies for improving prediction accuracy. Lee and Smith [LeS84] and McFarling and Hennessey [McH86] examine a range of schemes for

reducing branch penalty. DeRosa and Levy [DeL87] provide a quantitative comparison for different design alternatives for the branch instruction. Hsu and Davidson [HsD86] suggest a scheme whereby a large number of branch delay slots may be filled with guarded instructions, on machines such as the CRAY-1, where conditional branch resolution may take 14 clocks. These instructions are considered "guarded" because if branch resolution is not as expected, they are effectively treated as NOPs. Ditzel and McLellan [DiM87] and Grohoski et. al. [GKT90] discuss branch strategies as implemented on the Clipper and RS6000 processors respectively.

3.2 The Model

Consider a pipeline with s segments (Figure 3.1) executing an instruction J, which enters the pipeline the very next clock after instruction I. Assume a pipeline segment delay as equivalent to the system clock period. Suppose the instruction J at the start of its p_j^{th} stage of execution requires the result available at the completion of the q_i^{th} stage of execution of instruction I. The *degree of dependency* in such a case is defined as $d_{ij}=q_i-p_j$, where $q_i > p_j$. Suppose that instead of entering the pipeline the very next clock after I, J followed after an additional delay of x_{ij} clocks. Thus, if instruction I entered the pipeline at clock i and J entered at clock j, then $x_{ij}=j-i-1$. The degree of dependency is now reduced to

$$d_{i,j}=(q_i-p_j-x_{ij}) \ , \qquad \{3.1\}$$

where a segment freeze possibility, i.e., the possibility that a data item may spend more than one clock in a certain pipeline segment, is ignored. If $d_{ij} \leq 0$, I and J have null pipeline dependency, which means this dependency has no impact on pipeline throughput. On the other hand If $d_{ij} > 0$, I and J have positive pipeline dependency, which suggests this dependency has impact of d_{ij} clocks on the pipeline throughput. In other words, there is no pipeline output for d_{ij} clocks. The degree of dependency is maximum when $p_j = p_{j(min)} = 1$, $q_i = q_{i(max)} = s$ and $x_{ij} = x_{ij(min)} = 0$, i.e. $d_{ij(max)} = s-1$.

Next consider instruction dependency due to *branch* instructions. Let I represent a conditional branch instruction. In that case, the following instruction J, cannot be fetched until the execution of I is complete. Assuming that the instruction fetch (IF) stage is the first stage of the pipeline ($p_j = 1$) and the execution (E) stage, which tests the condition code, as the last pipeline stage ($q_i=s$), this leads to maximum pipeline dependency of $s-1$. Although condition code testing by the branch instruction I can be typically done in a stage prior to the execution stage, normally it can only be done after the previous instruction $I-1$ clears the execution stage and sets the condition code. So, branch instructions can potentially result in the maximum possible slowdown of $s-1$

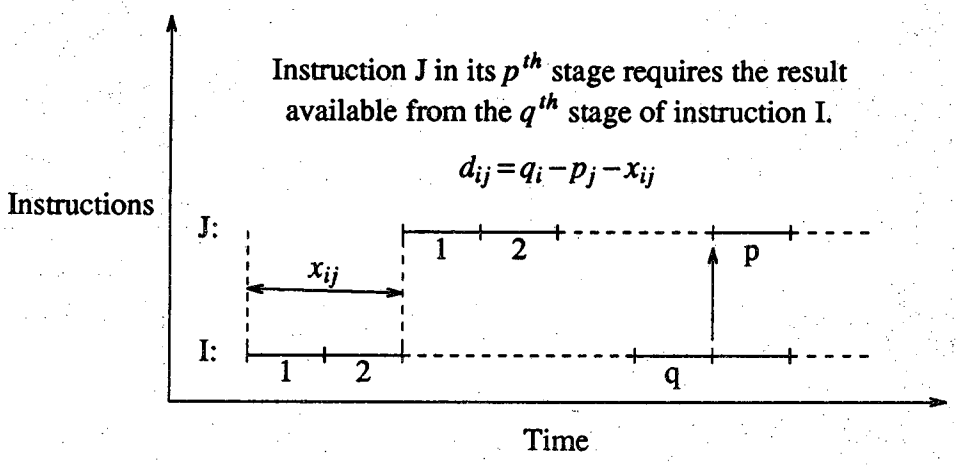


Figure 3.1 Instruction dependency in a pipeline.

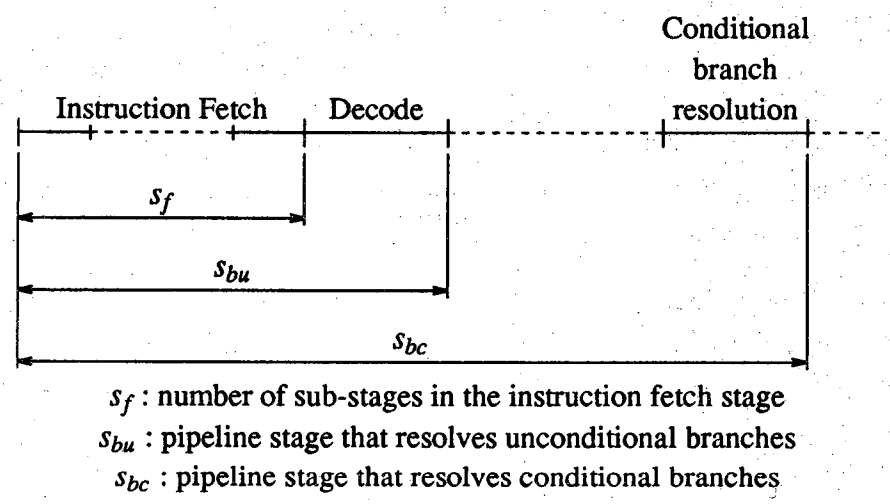


Figure 3.2 An instruction pipeline.

clocks. In general, branch instructions need not wait until the last pipeline stage for their resolution, especially unconditional branches.

A pipeline stage is considered frozen if it cannot accept a new data item at the end of the current clock period. Such a situation arises when some unexpected condition is encountered, such as a cache miss or a branch. A freeze implies delay at the subsequent pipeline stages as they wait for the frozen stage output. A successful branch instruction involves the fetch and execution of an *out of sequence* instruction. Fetching the branch target instruction consists of i) a target address calculation and ii) a target fetch. Each of these steps can cause a freeze. In this chapter other possible freeze conditions are deliberately ignored.

3.3 Classification of Branch Strategies

Branch strategies can be classified based on how they attempt to reduce the branch penalties, as shown in Table 3.1. The names of most of the strategies are self-descriptive. The unobvious ones are briefly described below.

The Loop buffer strategy is based on a high-speed memory in the instruction fetch stage of the processor. Some CDC machines (6600, 7600, and Star 100) as well as the CRAY-1 have used this idea. These buffers (Figure 3.3) can detect if the branch target (forward or backward) lies within the environment captured by the buffer and if so, the instruction fetch delay and the possible freeze delay are eliminated. Since a hit in the loop buffers avoids any external memory access, it also reduces extra instruction traffic in case of incorrect prediction. Although, loop buffers may appear to be similar to instruction caches, they are much smaller in size and, hence, lower in implementation cost. This strategy further assumes that branches are not likely to be taken.

Usually branch instruction execution does not require any operand fetch. Some IBM machines (370 series) use the operand fetch (OF) slot of the pipeline for fetching from the branch target path. The branch is still assumed not likely to be taken. The *Fetch Target in OF-slot* strategy is based on this technique.

The *Fetch Both Paths* strategy, also used on some IBM machines (370/168, 3033) uses the brute-force approach of fetching (not decoding) both the sequential and non-sequential instruction streams in case a branch is decoded.

The *Delayed Branch* [McH86] and *Predict Branch Always Taken with Target Copy* strategies modify the instruction sequence at compile time. The former delays the entry of the dependent branch instruction by inserting instructions that are common to both the sequential and non-sequential paths. In the latter strategy, a portion of target code, as dictated by the effective pipeline length for branch resolution, is copied (Figure 3.4)

Table 3.1 Classification of branch strategies.

Strategy	Label	Reduce dependency by			Reduce	Reduce
		increasing p_j	decreasing q_i	increasing x_{ij}	target-fetch freeze	address-calc freeze
Predict Never Taken	PBNT	X				
Loop Buffer	LB	X			X	
Pre-calculate Target Address	PTA	X				X
Fetch Target in OF-slot	FTOF	X			X	X
Predict Always Taken	PBAT		X			
Predict Always Taken with Target Copy	PTTC	X	X		X	X
Fetch Both Paths	FBP		X		X	X
Delayed Branch	DB	X	X	X	X	X
Taken/Not-taken Switch in the Decode Stage	TNTD		X			
Branch Target Buffer	BTB	X	X			X

Note: X indicates how the strategy attempts to reduce branch cost.

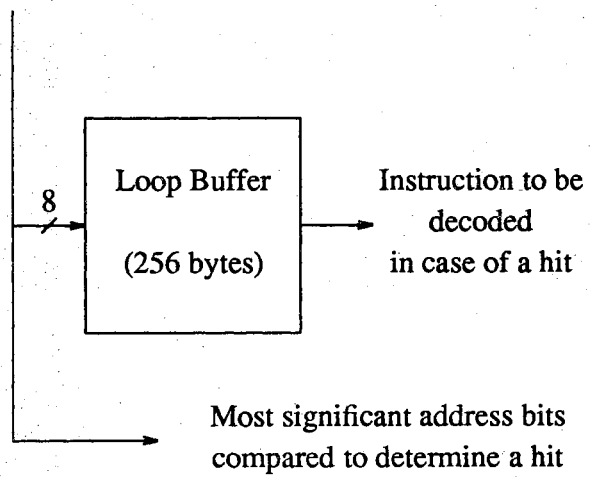


Figure 3.3 A loop buffer.

following the branch instruction. This strategy is also assumed to predict branches as always taken. Note that the *Delayed Branch* and *Target Copying* strategies also indirectly reduce the address calculation and target fetch freezes by delaying reliance on the target code and thereby offering time to calculate the address and fetch the target.

The last two strategies in Table 3.1 are based on active branch prediction [LeS84]. This prediction information can be obtained and improved for accuracy in many different ways [Smi81]. *Branch Target Buffer (BTB)* refers to a small associative memory in the instruction fetch stage of the processor. Instruction fetch addresses are associatively matched with the buffer contents and in case of a hit it predicts the most likely branch outcome as well as the most recent target address (Figure 3.5). As a result, target fetch does not need to wait for the branch decode and target address calculation. In case of a miss in BTB, branch instructions are handled in a manner similar to the *Predict Branch Never Taken* strategy.

3.4 Branch Prediction

Branch strategies do not eliminate branch delay, they reduce it with a certain probability. An implicit assumption about the most likely branch outcome and commitment to the sequential or to the branch path is made to varying degrees. This commitment normally reduces the penalty associated with the chosen path but may increase the penalty of taking the discarded path in case of incorrect prediction. As a result, overall performance improvement becomes critically dependent on the probability of correct prediction.

Table 3.2 defines and explains the terms associated with the model. Note that for $K=0$ or $b=0$, performance throughput, G, is assumed to be at its peak rate of one instructions per cycle. Thus, all other pipeline overheads (discussed in Chapter 1 and also in [DuF90]) are ignored.

Cost of Branch Prediction. The discussion above has centered around assessing the performance of different branch strategies. Consider the two primary costs involved: i) implementation cost and ii) operational cost. Implementation cost refers to the hardware/software costs involved in implementing the branch strategy. Since such costs are variable with technology, this cost is ignored. On the other hand, *operational cost* refers to the added run time cost, for example, the additional instruction traffic that results on the system bus with every incorrect branch prediction. Although incorrect predictions are the primary source of extra instruction traffic, even delayed correct prediction can cause wasted instruction fetch. For architectures that allow machine state update by instructions in the predicted path, there is an additional run time overhead of

```
          CMP     R1, R2
          JZ      xx
        * ADD     R3, R4
        * SUB     R3, R5
        * INC     R4
        * ADD     R3, R4
          MOV     R6, R7
          ADD     R6, R2
          MOV     R1, mem
        ...       ..
        ...       ...
xx:       ADD     R3, R4
          SUB     R3, R5
          INC     R4
          ADD     R3, R4
xx+4:     MOV     R6, R3
        ...       ...
        ...       ...
```

Instructions marked with an asterisk (*) are the instructions copied from the target (xx) at compile time. In the case of a successful branch, control transfers to the label xx+4, after executing the marked (*) target instructions via sequential fetch. In case the branch is not taken, marked instructions are discarded without execution after fetch and decode.

Figure 3.4 Predict branch always taken with target copy (*PTTC*).

Branch instruction address	Branch prediction	Predicted target address
. . .	. . .	. . .

Figure 3.5 A branch target buffer.

Table 3.2 Table of definitions.

Predicted	Actual	Probability	Branch Penalty	Instruction Traffic Penalty
no branch	no branch	$p_{n,n}$	$K_{n,n}$	$I^+_{n,n}$
no branch	branch	$p_{n,b}$	$K_{n,b}$	$I^+_{n,b}$
branch	no branch	$p_{b,n}$	$K_{b,n}$	$I^+_{b,n}$
branch	branch	$p_{b,b}$	$K_{b,b}$	$I^+_{b,b}$
Av. Branch Penalty, $K = p_{n,n} * K_{n,n} + p_{n,b} * K_{n,b} + p_{b,n} * K_{b,n} + p_{b,b} * K_{b,b}$				
Average Throughput, $G = \frac{1}{1 + K * b}$				
Av. Wasted Instruction Traffic, $I^+ = p_{n,n} * I^+_{n,n} + p_{n,b} * I^+_{n,b} + p_{b,n} * I^+_{b,n} + p_{b,b} * I^+_{b,b}$				
Merit Ratio, $MR = \frac{1}{(1 + K * b) * (1 + I^+ * b)}$				

Notes:

All four probabilities, $p_{n,n}, p_{n,b}, p_{b,n}$, and $p_{b,b}$ can be expressed in terms of the probability of *branch–to–be–taken* prediction and the probability of *correct* prediction (refer to Appendix-A).

Variable b denotes branch frequency.

shadowing the original machine state to be able to recover from an incorrect prediction. For the sake of simplicity, this cost is not included in the calculations, and it is not expected it to alter the conclusions. The only operational cost studied is that of the additional instruction traffic. Refer to Table 3.2 for the terms associated with this cost of wasted instruction fetches.

An ideal machine which can always correctly predict the branch outcome and if needed, can start fetching the target path right after the branch instruction fetch, would have, $K = I^{+} = 0$ and, hence, $G = 1$, resulting in unit merit ratio, MR, irrespective of the branch frequency, b. Interestingly, freeze conditions, which tend to increase the branch delay, reduce the average additional instruction traffic. When a certain path is predicted, freeze situations reduce the number of instructions that can be fetched, which reduces the number of wasted instruction fetches in case of incorrect prediction. This reduction has been taken into account in the calculation details provided in Appendix A (also in [DuF89]).

The following simplifying assumptions have been made (Figure 3.2):

a) The Instruction fetch stage is assumed to consist of s_f slots (each containing a prefetched instruction) followed by the decode stage.

b) Let s_b refer to the pipeline length up to the stage that resolves a pending branch instruction. For unconditional branches, branches are assumed to be resolved as soon as they are decoded, therefore, $s_b = s_{bu} = s_f + 1$. For conditional branches, $s_b = s_{bc}$, and is dependent on the pipeline stage that sets the condition code.

c) Each instruction is assumed to make a common trip through the pipeline stages. For pipelines with functional-level stages, such as fetch and execute stages, this should be a reasonable assumption.

d) Additional instruction traffic during freeze handling, e.g., in software page fault handling is ignored.

e) For the sake of simplicity, handling of multiple pending branches in the pipeline is restricted. If a branch is predicted as likely to be taken, it is assumed that additional branches are not encountered. before resolving the first branch. This assumption can be a source of some significant inaccuracy only for very long pipelines with prediction schemes which allow this possibility.

f) Finally, any on-chip instruction cache has been ignored in the discussion as it has no impact on the relative nature of the branch delay and additional instruction traffic performance curves.

3.5 Results

The model described above can be used to obtain the average branch delay (K), average number of wasted instruction fetches per branch (I^+) and the overall merit ratio (MR) once the variables defining the system environment are defined. Certain nominal values are assumed for some of these variables (Table 3.3); e.g., branch frequency, b = 0.25, where 80 percent of the branches are conditional. The probability of a freeze during target address calculation is assumed to be 0.5 with a freeze duration of 2 cycles. The probability of freeze during target fetch is ignored. For the delayed branch approach, an average of one useful common instruction (i.e. $u = 1$) is assumed. One such machine employing the delayed branch approach, MIPS [McH86, GrH86], reported use of a single delay slot about 70 percent of the time. There may be special cases, such as when using guarded instructions [HsD86], where a significant number of delayed branch slots may be utilized. Based on Smith [Smi81], a correct prediction probability of 0.85 is assumed for conditional branches. For Branch Target Buffer, the probability of correct target address prediction is optimistically set at 0.9, assuming stable branch targets [Smi81]. The probability of a BTB-hit for non-branch instructions for writable code segments is assumed very low at 0.05. Assume nominal loop buffer hit ratio, p_{lh} = 0.6 and nominal BTB-hit ratio, p_{th} = 0.8. Peuto and Shustek [PeS77] report a hit ratio of 0.6 for a loop buffer of ±256 entries, whereas Lee and Smith [LeS84] report a hit ratio of around 0.8 for a target buffer with 256 entries and a set size of 4 or 8. Set size refers to the degree of associativity in contrast to the fully associative BTB search.

The initial focus is on the input parameter, p_{sb}, successful branch probability (conditional and unconditional combined). Results are obtained for the three performance parameters: average branch delay, K; average number of wasted instruction fetches, I^+; and the cost-performance merit ratio, MR, as shown in Figures 3.6 through 3.11. Appendix A provides details of these calculations. While p_{sb} is varied, other parameters are kept at their nominal values.

3.5.1 Inferences

The following inferences can be made regarding the three performance parameters as a function of the successful branch probability (p_{sb}):

a) The BTB outperforms the others over the entire typical operating range ($0.55 < p_{sb} < 0.7$).

b) The predict-branch-always-taken scheme with target copy ($PTTC$) emerges as a good second choice around p_{sb} of 0.65 or more. Interestingly, even without any

Table 3.3 Nominal values of model parameters.

Average branch frequency, b	0.25
Average fraction of conditional branches	0.8
Overall fraction of successful branches, p_{sb} (conditional/unconditional combined)	0.6
Number of pipeline stages until unconditional branch resolution, s_{bu}	2
Number of buffer sub-stages in the instruction fetch stage, s_f	1
Number of pipeline stages until conditional branch resolution, s_{bc}	5
Probability of freeze during target address formation	0.5
Duration of target-address-calculation freeze	2 cycles
Probability of freeze during target-fetch, p_f	0
Duration of target-fetch freeze	10 cycles
Probability of loop-buffer hit, p_{lh}	0.6
Probability of BTB hit, p_{th}	0.8
Probability of correct address prediction from BTB	0.9
Probability of BTB-hit for non-branch instruction	0.05
Average number of delay-slots filled in delayed branch approach	1
For cases with active prediction schemes (*TNTD, BTB, TNTLB, TNBTB*)	
Correct prediction probability for unconditional branches	1.0
Correct prediction probability for conditional branches	0.85

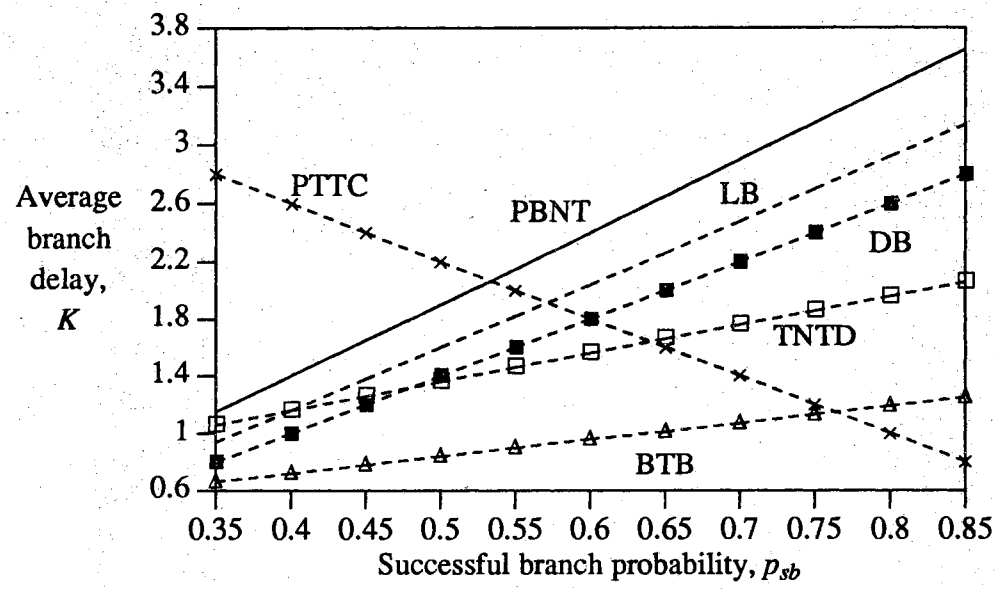


Figure 3.6 Average branch delay versus successful branch probability for *PBNT*, *LB*, *PTTC*, *DB*, *TNTD*, and *BTB* strategies.

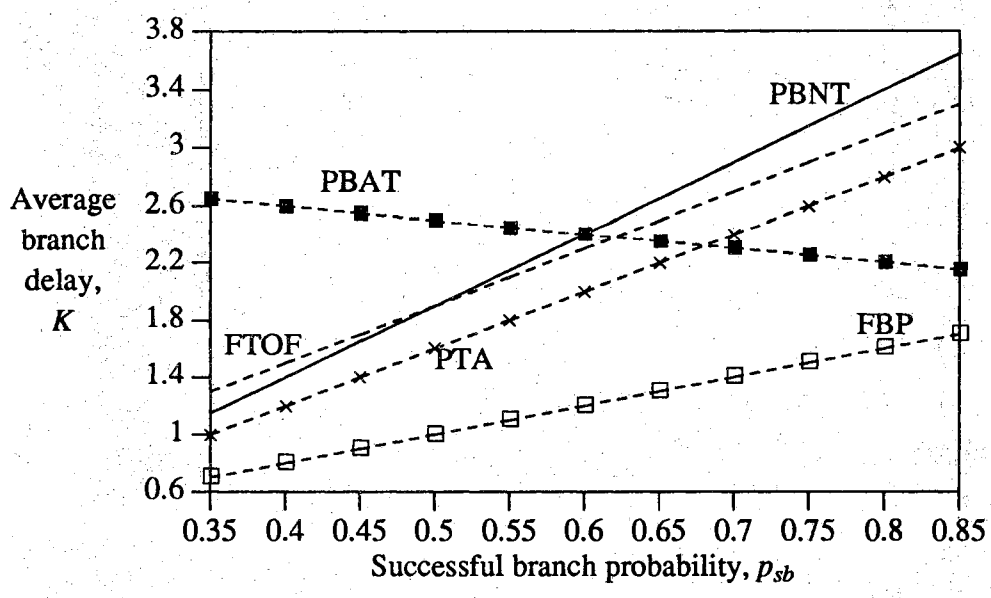


Figure 3.7 Average branch delay versus successful branch probability for *PBNT*, *PTA*, *FTOF*, *PBAT*, and *FBP* strategies.

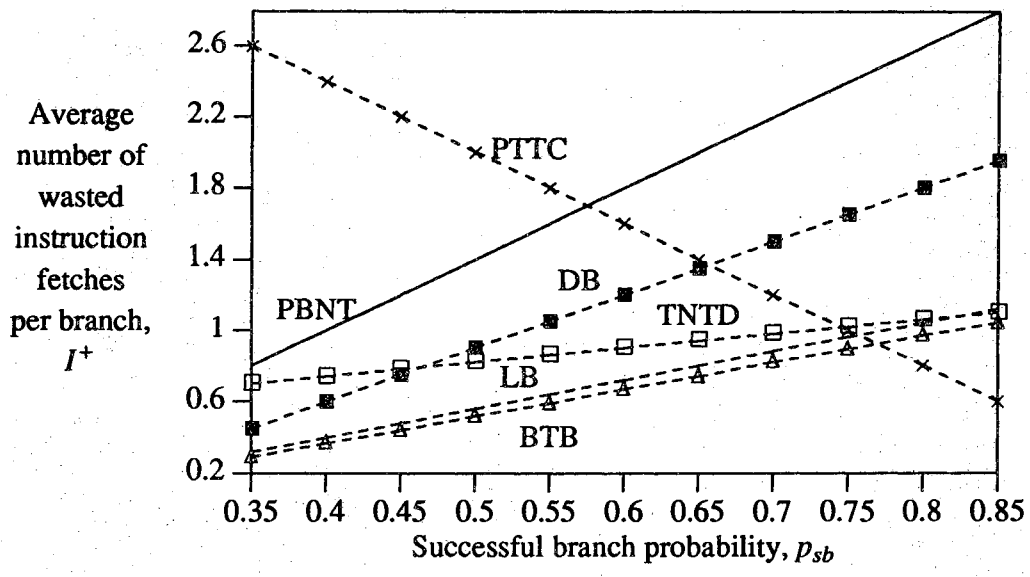


Figure 3.8 Average number of wasted instruction fetches per branch versus successful branch probability for *PBNT*, *LB*, *PTTC*, *DB*, *TNTD*, and *BTB* strategies.

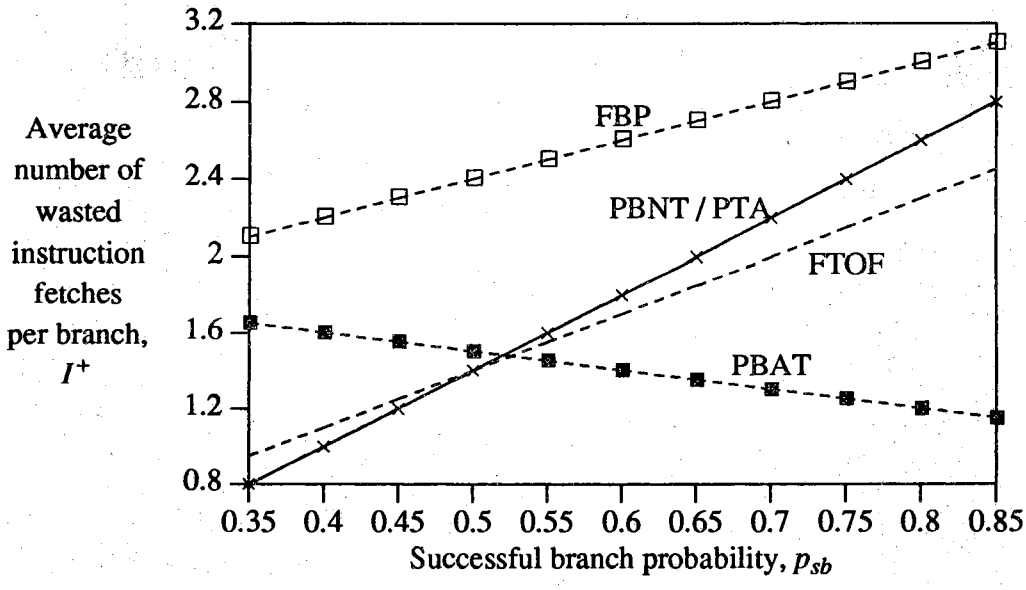


Figure 3.9 Average number of wasted instruction fetches per branch versus successful branch probability for *PBNT*, *PTA*, *FTOF*, *PBAT*, and *FBP* strategies.

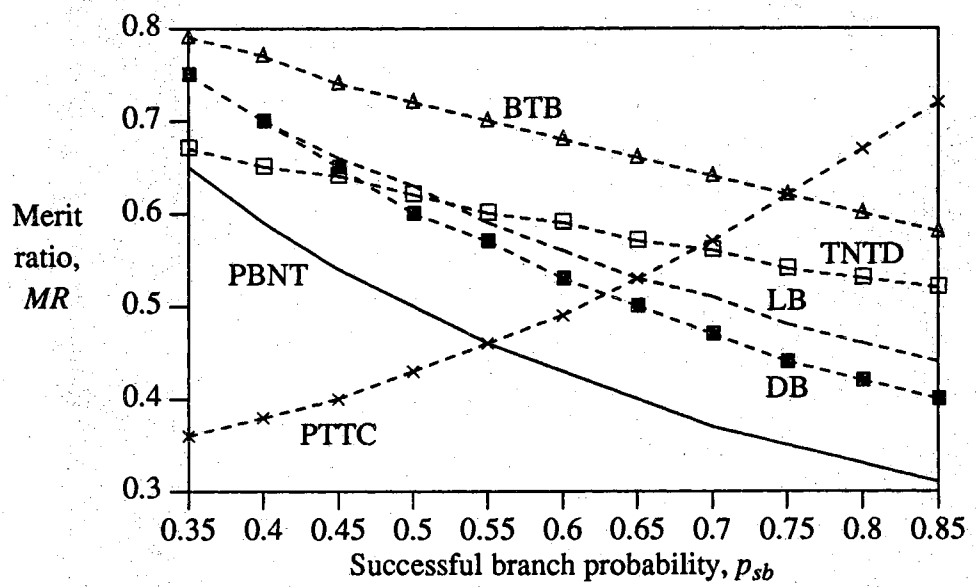


Figure 3.10 Merit ratio versus successful branch probability for *PBNT*, *LB*, *PTTC*, *DB*, *TNTD*, and *BTB* strategies.

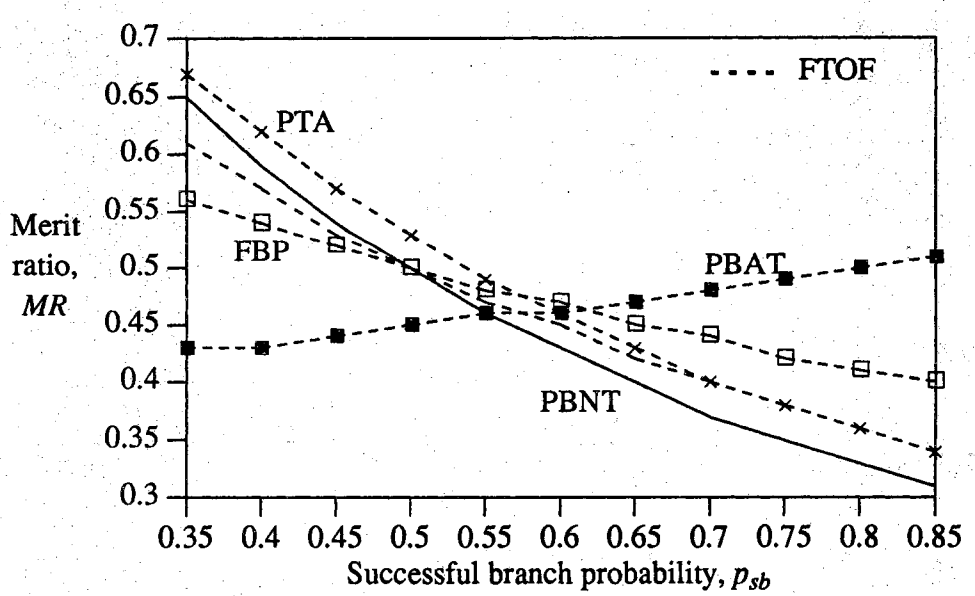


Figure 3.11 Merit ratio versus successful branch probability for *PBNT*, *PTA*, *FTOF*, *PBAT*, and *FBP* strategies.

active branch prediction support, it exhibits better performance potential than BTB around $p_{sb} \geq 0.75$. This advantage stems primarily from the fact that this scheme does not have to pay the delay penalty of incorrect target address prediction. BTB has a cost for incorrect target address prediction even with correct branch prediction. As a cautionary note, *PTTC* also exhibits the steepest slope in terms of all the three performance parameters as opposed to the relatively stable performance curves of the active prediction schemes like, *Branch Taken/Not-taken Switch in the Decode Stage* and *BTB*.

c) In terms of excess instruction fetches, loop buffer scheme performs almost as well as the BTB. Loop buffers can significantly reduce the cost of excess instruction traffic resulting from incorrect predictions.

d) At nominal p_{sb} (0.6) both *Predict Branch Never Taken* and *Predict Branch Always Taken* have the same branch delay. Which of the two should be the preferred scheme? A look at the additional instruction traffic cost can help resolve the issue.

Predict Branch Always Taken has lower cost of wasted instruction fetches and hence has better merit ratio (*MR*). In the absence of any address calculation freeze (or target fetch freeze), *Predict Branch Never Taken* on average wastes more instructions during misprediction than *Predict Branch Always Taken*. A similar dilemma between *Predict Always Taken with Target Copy* strategy and *Delayed Branch* can be resolved in favor of *Delayed Branch*, due to its lower added instruction traffic cost. For both the schemes implementation costs are almost identical, hence for the two strategies in question, hence the excess instruction traffic is the important decisive factor. Interestingly, at $p_{sb} = 0.5$, three different strategies: predict branch never-taken, target fetch in the OF-slot, as well as the scheme to fetch both the paths, show almost identical merit ratios. Here implementation cost can probably be the only decisive factor.

e) Not only does excess instruction traffic cost help choose between two almost equally performing strategies, it can also caution us about otherwise very well performing strategies. *FBP* (fetch both paths) provides an interesting example in this regard. In terms of average branch delay (*K*) it performs almost as well as the *BTB* strategy. But, after considering the cost of wasted instruction fetches, in terms of the overall merit ratio (*MR*), *FBP* is not much better than the worst performing *Predict Branch Never Taken* strategy. Thus, the average branch delay alone does not determine overall performance. conclusion based solely on average branch delay, *K* may be elusive one as far as the overall system-performance is considered. Garcia and Huynh [GaH80] discuss the efforts made to reduce the resulting high contention on the system bus in an early IBM 370 implementation using *FBP*.

The variation in system performance as a function of the number of buffer stages in the instruction fetch stage has also been computed. Again *BTB* outperforms every other

strategy, followed by *Predict Always Taken with Target Copy*, in terms of average branch delay (K) for any amount of buffering in the fetch stage. All the strategies are seen to have almost identical performance slopes on the merit ratio curve and show identical sensitivity with respect to s_f. Figures A.1 to A.3 in Appendix A contain these plots.

Finally, performance curves were generated as a function of s_{bc}, i.e. the total number of pipeline stages required for conditional branch resolution. BTB continued to be the first choice for any number of segments in terms of overall merit ratio. But for long pipelines ($s_{bc} > 6$) it slipped, instead fetching both paths (*FBP*) finally won with its constant branch delay with respect to s_{bc}. Note that just a branch taken/not-taken switch in the decode stage (*TNTD* scheme) significantly reduces the branch delay. The additional reduction in branch delay obtainable through BTB rapidly decreases with larger s_{bc}. Figures A.4 to A.6 in Appendix A contain these plots.

Therefore, in the typical operating range ($0.6 < p_{sb} < 0.75$) there are three competing strategies: Loop Buffer (*LB*), Predict Branch Always Taken with Target Copy (*PTTC*) and Branch Target Buffer (*BTB*). The branch delay numbers for *Predict Branch Never Taken*, *Delayed Branch*, and *BTB* under nominal conditions come quite close (within 30 percent) to those reported by McFarling and Hennessey [McH86], even though the nominal conditions while close, are not exactly the same as theirs. Assuming branch frequency, $b = 0.2$, the results indicate a throughput (G) of around 10 percent in the above mentioned operating range of p_{sb}. This is also in close agreement with MIPS simulation results [Gro83] of around 9 percent and the analysis of DeRosa and Levy [DeL87], suggesting an improvement of around 8 percent.

In the following section some hybrid strategies are discussed that are based primarily on these three strategies. Delayed branch and *Taken/Not-taken Switch in the Decode Stage* also show good performance potential in possible combinations with above strategies.

3.6 Hybrid Strategies

The following hybrid strategies are considered:

a) Predict Branch Always taken with target-copy and delayed branch. (*TTCDB*): This is the only hybrid strategy considered with almost no additional implementation cost and only some software (compiler) cost.

b) Predict Branch Always taken with target-copy, delayed branch and Loop buffer (*TTDLB*).

c) Taken/Not-taken Switch in the Decode Stage with Loop buffer (TNTLB).

d) Taken/Not-taken Switch in the Decode Stage with Branch target buffer (TNBTB). Finally, consider a combination of *TNTD* and *BTB*. For a miss in the BTB, instead of falling back on the default *Predict Branch Never Taken* case, this strategy assumes a branch taken/not-taken switch in the decode stage similar to *TNTD*.

3.6.1 Inferences

Sensitivity plots of the performance parameters, K, I^+, and MR are obtained with respect to p_{sb} and s_{bc} (Figures 3.12 - 3.15).

a) Around the nominal values of system parameters and of the hybrid strategies, the minimum implementation cost strategy, *TTCDB*, performs better than every non-hybrid strategy except *BTB*. For p_{sb} around 0.7, it even outperforms *BTB* in terms of average branch delay (K) as well as merit ratio (MR).

b) Around the nominal conditions, the last three hybrid strategies: *TTDLB*, *TNTLB*, and *TNBTB* are almost equally competitive. For shorter pipelines ($s_{bc} < 5$) *TTDLB* has a slight edge over the other two. For longer pipelines active branch prediction becomes more important and *TNTLB* and *TNBTB* perform better than the rest and continue to follow each other closely. Therefore, on a system with a branch-taken/not-taken prediction switch in the instruction decode stage, if one were to choose between the addition of either the loop buffer or the branch target buffer, careful consideration should be given to implementation cost issues which may tilt the balance slightly in favor of the loop buffer based *TNTLB* scheme.

c) For $p_{sb} = 0.7$ or more, at nominal s_{bc}, *TTDLB* strategy outperforms the others and emerges as the first choice, in terms of all the three performance parameters. Around $p_{sb} = 0.7$, *TTDLB* reduces the branch delay to less than one third as compared to the *Predict Branch Never Taken* strategy.

Effect of loop (target) buffer hit probability. Target buffer based strategies show more sensitivity to the hit ratio, p_{th}, than the loop buffer based strategies in terms of average branch delay (Figure 3.16). Loop buffer based strategies are more sensitive than the target buffer based strategies in terms of the excess instruction traffic cost with respect to the corresponding hit-ratio p_{lh} (see Figure A.12). As a result, both classes of strategies exhibit almost identical slope on the merit ratio performance curve (see Figure A.13).

Effect of Target-fetch freeze probability. The discussion so far has ignored any potential for a freeze (due to say, cache miss or page fault) while attempting to fetch the branch target. Assuming a fetch freeze duration of 10 clocks, the performance sensitivity with

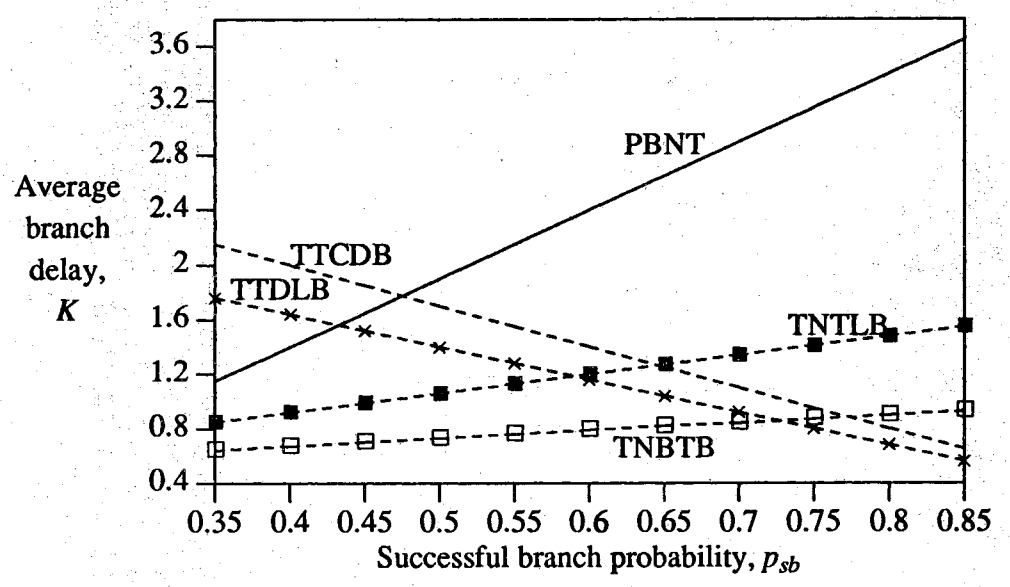


Figure 3.12 Average branch delay versus successful branch probability for *PBNT*, *TTCDB*, *TTDLB*, *TNTLB*, and *TNBTB* strategies.

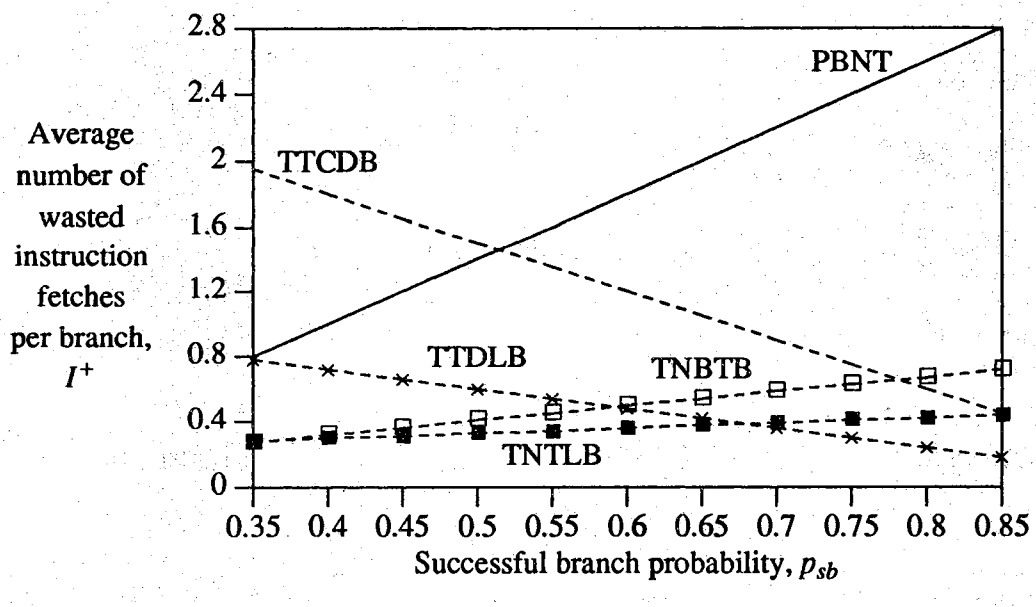


Figure 3.13 Average number of wasted instruction fetches per branch versus successful branch probability for *PBNT*, *TTCDB*, *TTDLB*, *TNTLB*, and *TNBTB* strategies.

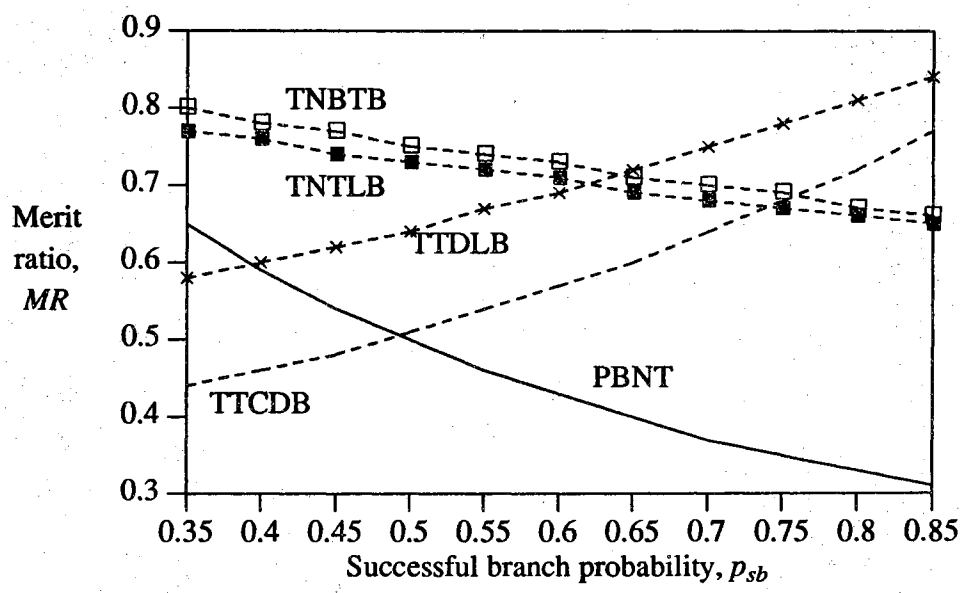


Figure 3.14 Merit Ratio versus successful branch probability for *PBNT*, *TTCDB*, *TTDLB*, *TNTLB*, and *TNBTB* strategies.

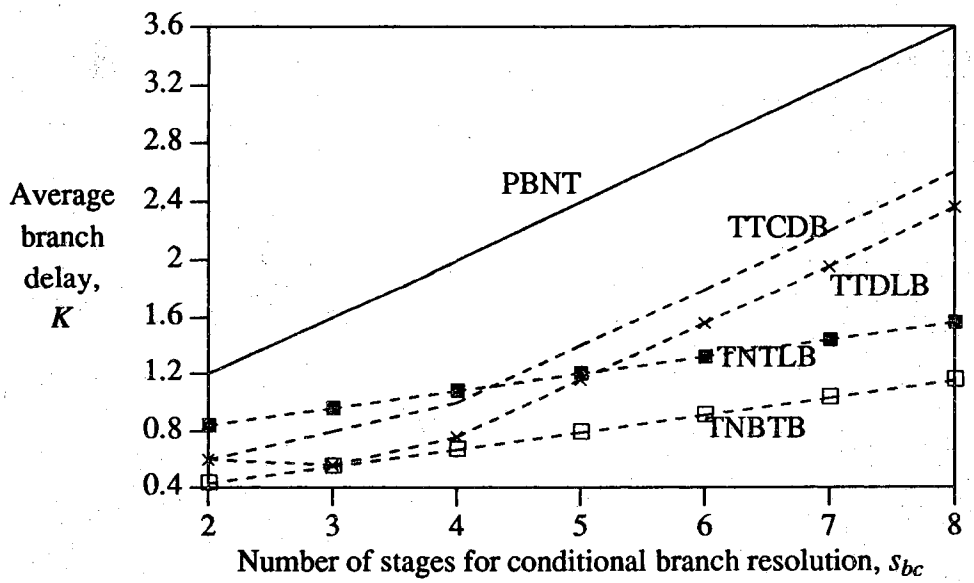


Figure 3.15 Average branch delay versus number of stages for conditional branch resolution for *PBNT*, *TTCDB*, *TTDLB*, *TNTLB*, and *TNBTB* strategies.

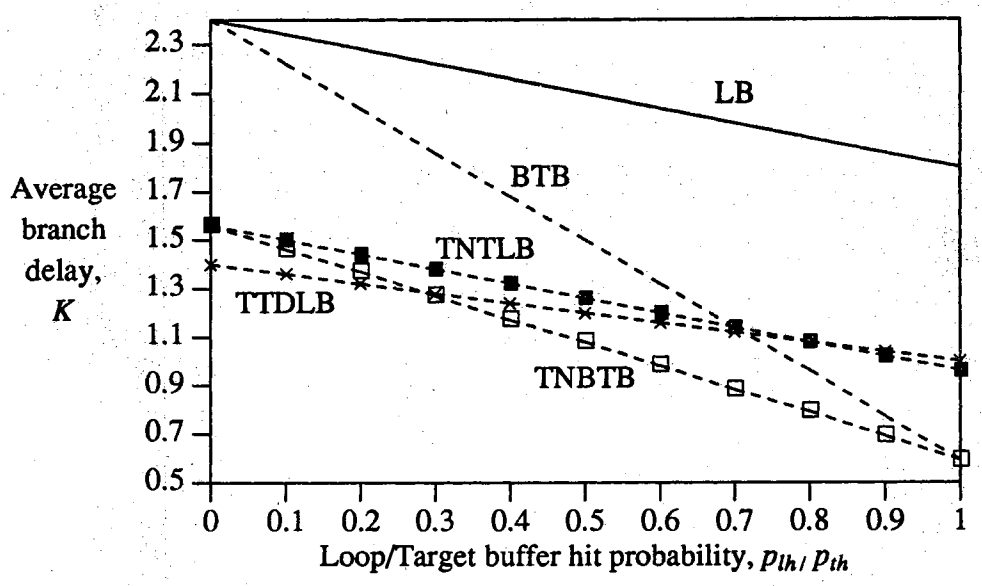


Figure 3.16 Average branch delay versus Loop/Target buffer hit probability for *LB*, *BTB*, *TTDLB*, *TNTLB*, and *TNBTB* strategies.

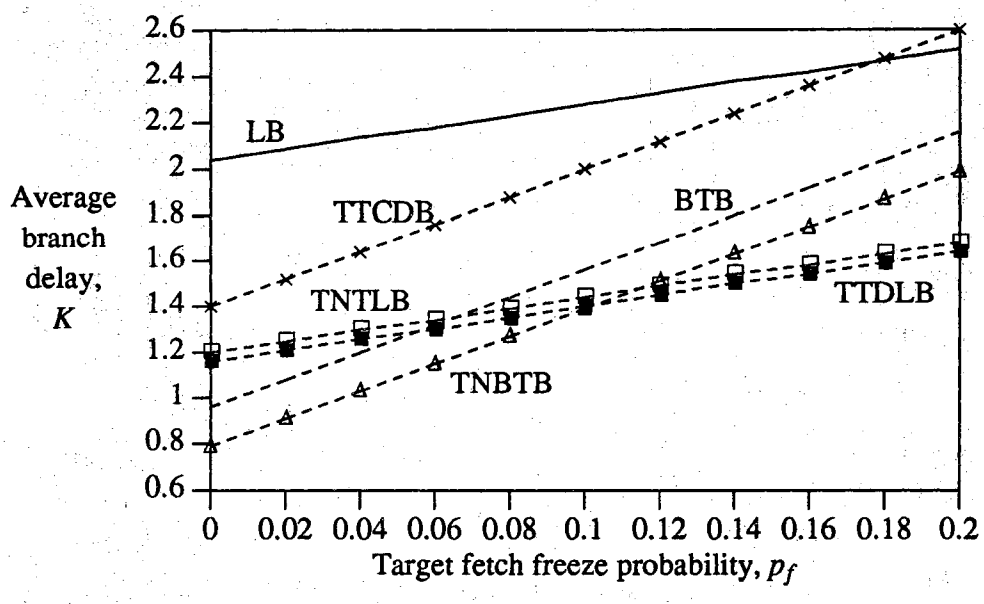


Figure 3.17 Average branch delay versus target fetch freeze probability for *LB*, *BTB*, *TTDLB*, *TNTLB*, and *TNBTB* strategies.

respect to the fetch-freeze probability (p_f) is analyzed next. Loop buffer based strategies are at an advantage in such a case because a hit in the loop buffer also eliminates any page fault potential associated with external memory access. As a result, loop buffer based strategies show more performance stability with respect to p_f than the BTB-based strategies. For example, if p_f increases from 0 to 0.1 average branch delay for the loop buffer based *TNTLB* strategy increases by 20 percent, whereas that in the case of the BTB-based *TNBTB* strategy increases by more than 75 percent (Figure 3.17).

3.7 Summary

A common analytical platform, based on certain system and program parameters, can be developed for classifying and comparing different branch strategies. Such an approach has the advantage of being far less time consuming and more flexible compared to simulation-based approaches. Excess instruction traffic caused by different branch strategies has been overlooked in the past. Additional instruction traffic helped to distinguish an overall performance difference between some apparently equally well performing strategies. A branch strategy using a branch-taken/not-taken switch in the decode stage is found to be almost as effective in combination with the loop buffer as with the branch target buffer. In a typical microprocessor environment with less than five segments with a successful branch probability around 0.6, a branch strategy based on default prediction of branch always taken, along with compiler support for target copy and delayed branch is shown to provide performance potential comparable to a branch strategy based on Branch Target Buffer.

Finally, certain components of branch delay have been ignored in this chapter. For example, some machines [SJH89] have an added delay during branches if the target is misaligned. Also some compilers, such as trace scheduling [Fis81] compiler have an additional overhead of patch-up code if their compile time prediction of a branch is found incorrect at run time. These delay components are analyzed in detail in Chapter 5.

CHAPTER 4
SUPERPIPELINED VERSUS SUPERSCALAR

4.1 Introduction

Recent advances in technology have made it now feasible to put multiple execution pipelines on the same chip. Previous chapters have explored some of the issues issues associated with optimal design of single pipelines systems. This chapter extends the analysis into the realm of superscalar processors. An analytical model is proposed as an alternative tool for analyzing the tradeoff between superpipelined processors. The factors that contribute to performance limits are analyzed. The duality of superpipelines and superscalars is examined in detail and certain imperfections of this duality are described. Jouppi and Wall [JoW89] studied tradeoffs of superpipelined and superscalar machines via simulations. Smith, et. al. [SJH89] investigated the performance limits of such machines as a result of instruction fetch inefficiencies.

4.2 Superpipeline/Superscalar Tradeoff Model

Consider performing an operation using a circuit having g gate-levels of propagation delay. The quantity g is the *operation gate delay*. Suppose there are n such operations and a set of k pipelines, each s stages deep, to support execution (see Fig. 1.7). Each pipeline latency is assumed to be s clocks. If inter-stage buffers are assumed to have one gate-level of delay then, $g_{pipelined} = g + s - 1 + \varepsilon$, where ε is the smallest integer such that $g_{pipelined}$ evenly divides $g + s - 1 + \varepsilon$. The quantity $s - 1 + \varepsilon$ is the overhead due to pipelining. If the circuit is not pipelined, i.e., $s = 1$, then $\varepsilon = 0$ and pipelining overhead is null. At most, $\varepsilon = s - 1$, so the worst case pipeline overhead is $2(s - 1)$ gate delays.

For n a multiple of $k * s$, the first set of k results from the k pipelines is produced after s clocks, and the remaining $n - k$ results take $(n - k)/k$ clocks. Let T_n be the time to execute n operations. Then,

$$T_n = \left\lceil s + \frac{n-k}{k} \right\rceil \left\lceil \frac{g_{pipelined}}{s} \right\rceil \text{ gate delays}$$

$$\leq \left\lceil s + \frac{n-k}{k} \right\rceil \left\lceil \frac{g+2(s-1)}{s} \right\rceil \text{ gate delays (worst case).} \qquad \{4.1\}$$

The ideal throughput represented by Equation (4.1) is difficult to achieve in practice due to additional delays that can be grouped into the following categories:

a) *Scheduling Delays:*
   i) *Instruction Fetch Delay:* delay due to restricted main memory bandwidth on an instruction cache miss,
   ii) *Branch Delay:* instruction fetch delay due to uncertainties in the execution path,
   iii) *Dependency Check Delay:* delay due to run time dependency check in an instruction window, and
   iv) *Dependency Delay:* delay scheduled to satisfy dependency constraints.

b) *Execution Delays:*
   i) *Operand Fetch Delay:* delay in fetching the operand(s) from memory, and
   ii) *Multiple Cycle Operations:* delay due to operations that take more than one clock in the execution stage.

Although the delays listed above are fairly independent of each other, there is some overlap. For example, consider an instruction I that takes multiple cycles to execute. Also assume that the following instruction J is data-dependent on I. In such a case, the execution delay of instruction I can also be viewed as the scheduling delay for instruction J. Alternatively stated, the dependency of an instruction J on an instruction I *lingers* for multiple clocks, if the execution of I takes more than one clock. For single pipelines, lingering dependency can be modelled the same way as the effect of multiple cycle operations in Section 2.4. However, in this chapter the delays due to multiple cycle operations and restricted memory bandwidth for instruction fetch are ignored on the premise that the hardware is designed to deliver k operations per cycle throughput on a sustained basis. Delays due to run time dependency checking are ignored on the assumption either compilation was done conservatively to eliminate the possibility of run time dependencies or that sufficient hardware is provided to do dependency checking without incurring any delay.

Let the branch probability be b with each branch taking $g*d_b$ gate delays for resolution. For example, d_b=0.5 implies that the branch delay is half the operation gate delay, or roughly half the pipeline length because of pipeline overhead.

Operand fetch delay is modelled assuming that on-chip cache has no access delay and internal bandwidth (cache to pipelines) is k operands at a time. Further assume that external bandwidth (main memory to cache) is limited to one operand at a time, hence, miss processing is sequential. The operand-miss probability is $\varepsilon = w * (1-h)$, where w is the probability of operand reference for an operation and h is the cache hit probability. Assume cache miss processing takes $g*d_m$ gate delays.

These further assumptions are made:

1) Instructions are issued simultaneously to the k pipelines, and there is no inter-stage buffering of intermediate results. This means that if there is a pending branch in any one of the pipelines that delays its following instruction fetch, then all the k pipelines freeze. Similarly, any cache miss on an operand fetch for one pipeline delays all pipelines. Thus, pipelines are synchronized. To do otherwise is a complicated hardware task of dubious cost effectiveness.
2) In the absence of any branch delay, assume that k operations are always issued to the k pipelines. Thus, issuing constraints imposed due to data dependency are ignored. This is the weakest of all assumptions and it is further addressed in following chapters where utilization constraints due to such dependencies are discussed in detail.
3) Delays as a result of instruction cache misses are ignored. Note that instruction cache misses on branches can be assumed included in the branch delay.

Assuming only one cache miss at a time, the additional delay term to be added to Equation (4.1) is:

$$n[b\, g\, d_b + \varepsilon\, g\, d_m - b\, \varepsilon\, min(d_b, d_m)\, g] \ ,$$

where the last term accounts for the overlap of the instruction fetch and operand fetch delays due to branches and cache miss, respectively.

The instructions undergoing simultaneous execution must be independent in order to have been scheduled together. Simultaneous cache requests are then independent random variables, and are assumed to be identically distributed. Thus, multiple cache misses follow a binomial distribution. Allowing one operand fetch per pipeline per clock, there can be up to k simultaneous cache misses. Allowing for multiple simultaneous cache misses Equation (4.1) becomes

$$n\,[b\, g\, d_b + \varepsilon\, g\, d_m] - \frac{n}{k}\left[\sum_{i=1}^{k} g\, b\, \varepsilon_i\, min(d_b, i\, d_m)\right] \qquad \{4.2\}$$

where ε_i is the probability that i cache misses occur simultaneously, and

$$\varepsilon_i = {}^kC_i\, \varepsilon^i\, (1-\varepsilon)^{k-i}$$

The mean value of the distribution is $k\,\varepsilon$ and ${}^kC_i = \frac{k!}{i!\,(k-i)!}$ Assuming further that $d_b < d_m$, Equation (4.2) becomes

$$n\,g\,(b\,d_b + \varepsilon\,d_m) - \frac{n\,g\,b\,d_b\,(1-\varepsilon_0)}{k} \qquad \{4.3\}$$

Combining Eqs. (1) and (3), the total time (in terms of gate delay) for n operations is

$$T_n \leq \left[s + \frac{n-k}{k}\right]\left[\frac{g+2(s-1)}{s}\right] + n\,g\,(b\,d_b + \varepsilon\,d_m) - \frac{n\,g\,b\,d_b\,(1-\varepsilon_0)}{k} \qquad \{4.4\}$$

Equation (4.4) is useful in deciding whether or not an additional pipeline will yield a significant throughput improvement justifying its additional cost. Figure 4.1 plots throughput as a function of the number of pipelines (k).

Resource utilization, u, is

$$u = 1 - \frac{\textit{wasted time slots in units of gate delay}}{\textit{total available time slots in units of gate delay}}$$

where the numerator is the delay from Equation (4.2) and the denominator is the sum of the delays from Equations (4.1) and (4.2). Figure 4.2 shows utilization versus number of pipelines. The duality of superscalar and superpipeline systems, i.e., any throughput achieved using a pipeline of certain depth can also be achieved using a corresponding number of pipelines of depth one, is evident. The throughput benefit for a given increase in pipeline number or depth decreases for greater initial pipeline number or depth. Because pipeline replication is more area-intensive than additional segmentation, and segmentation is more and more difficult to obtain, the guiding rule of design should be: segment pipelines to the extent feasible, then replicate pipelines.

4.3 Performance Limits

Looking at the throughput curves in Figure 4.1, it would be reasonable to ask: *What are the performance limits as another segment or another pipeline is added?*

First consider the throughput limit when using additional pipelines. Rearrange Equation (4.4) by grouping terms that are functions of k and those that remain. The rearranged form can be written

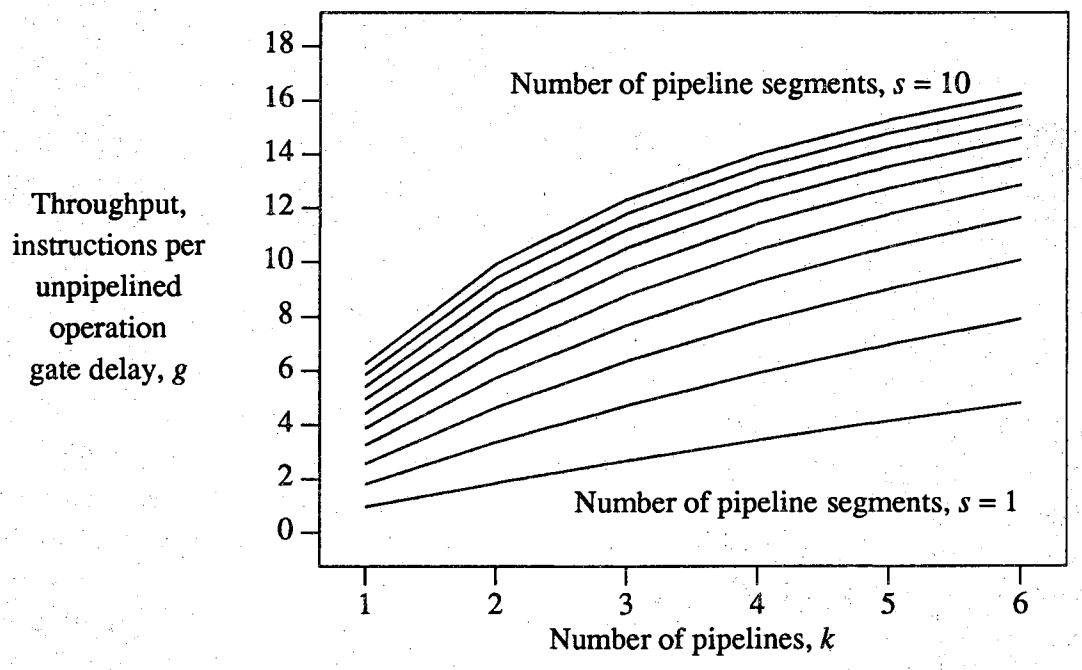


Figure 4.1 Normalized throughput versus number of pipelines, with the following nominal assumptions: data cache reference probability = 0.5, data cache miss probability = 0.05, data cache miss duration = 0.5 * operation delay, branch probability = 0.2, and branch delay = 0.15 * operation delay.

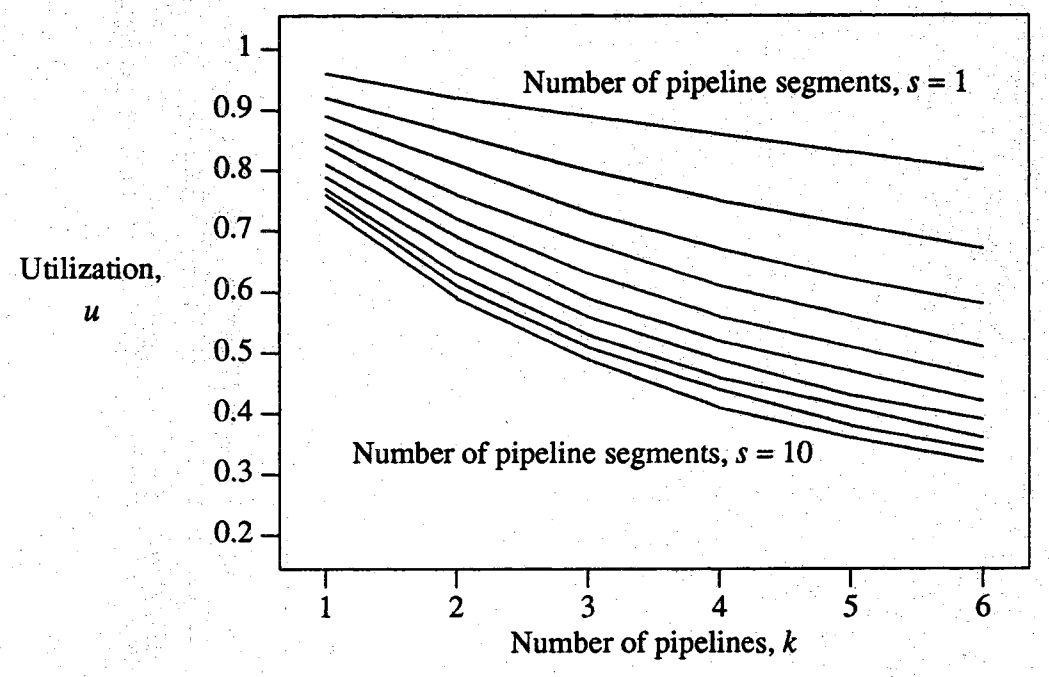


Figure 4.2 Utilization versus number of pipelines (parameter values the same as in Figure 4.1).

$$\frac{ng}{T_n} = G = \left[\frac{A}{k} + B \right]^{-1} \tag{4.5}$$

where

$$A = \frac{1}{s} + \frac{2}{g} - \frac{2}{g\,s} - b\,d_b\,(1-\varepsilon_0)$$

and

$$B = \frac{1}{n} + b\,d_b + \varepsilon\,d_m - \frac{4}{g\,n} + \frac{2}{g\,n\,s} - \frac{1}{n\,s} + \frac{2\,s}{g\,n} \ .$$

For a continuous instruction stream, n can be assumed to be large. Therefore,

$$B \approx b\,d_b + \varepsilon\,d_m \ .$$

As $k \to \infty$,

$$G \to B^{-1} \approx (b\,d_b + \varepsilon\,d_m)^{-1} \ . \tag{4.6}$$

This limit is independent of s and is simply a function of the branch penalty, which limits instruction fetch, and the cache miss penalty, which limits the execution time. For the set of parameter values used in Figure 4.1, the above limit evaluates to $G = 23.53$. Considering the fact that normally $b \gg \varepsilon$, the above limit has been referred to as the *fetch bottleneck*, also sometimes known as *Flynn's bottleneck* [Fly72].

Now consider the performance limit when deepening the pipelines. Equation (4.4) can again be rearranged to yield

$$\frac{ng}{T_n} = G = \left[\frac{C}{s} + Ds + E \right]^{-1} \tag{4.7}$$

where

$$C = \frac{1}{k} - \frac{2}{g\,k} + \frac{2}{g\,n} - \frac{1}{n} \ , \quad D = \frac{2}{g\,n}$$

and

$$E = \frac{1}{n} + b\,d_b + \varepsilon\,d_m - \frac{4}{g\,n} + \frac{2}{g\,k} - \frac{b\,d_b\,(1-\varepsilon_0)}{k} \ .$$

Unlike the previous case, as $s \to \infty$, $G \to 0$. This is because the overhead of additional buffers grows with additional segments. Therefore, beyond a point, this overhead overtakes the gain due to segmentation. The issue of optimal pipelining was studied in detail in Chapter 1 and in [DuF90]. Ignoring synchronization overhead, (which is a reasonable assumption for superscalar-type processors, unlike multiprocessors), there is

no reason why throughput should decrease due to the addition of a pipeline. Thus, the duality of superpipelines and superscalars is not perfect. As s grows, g must remain at least of the same order as s, so for large n

$$C \approx \frac{1}{k}\ ,\ Ds \approx 0\ ,\ \text{and}\ E \approx \frac{-b\,d_b\,(1-\varepsilon_0)}{k} + b\,d_b + \varepsilon\,d_m\ .$$

Therefore, as s increases,

$$G \rightarrow \left[b\,d_b + \varepsilon\,d_m - \frac{b\,d_b\,(1-\varepsilon_0)}{k} \right]^{-1} . \qquad \{4.8\}$$

This limit is same as that given by Equation (4.6) except the last term, which vanishes for large k. Recall that this last term represents the saving due to hiding some branch delay when overlapped with data-cache miss processing for one of the pipelines. This saving is apportioned over the k pipelines and hence becomes negligible for large k. For the set of parameter values used in Figure 4.1, the limit on throughput given by Equation (4.8) evaluates to 23.92.

4.4 Modelling Resource Utilization

The scheduling and execution delays listed in Section 4.2, although different in their original causes, have a common impact. They introduce unwanted bubbles (pipeline stalls) in the system, which finally ripple through different stages to cause loss of net system throughput. As one stage is delayed in delivering the intermediate result to its successor, the successor stage waits idly, and hence the system utilization drops. It is relatively much easier to predict a system performance in an ideal setting, assuming no such loss. In other words, modelling this drop in system utilization in a non-ideal, real-time environment is the key to an accurate performance prediction.

Consider the generic drop in utilization as system resources of a certain kind are added, such as increasing the number of pipeline stages, increasing the number of pipelines, or adding more processors. If these added resources are fully utilized and if any overhead is ignored, system throughput should increase in an easily predicted manner. For example, if two pipelines are always busy, the throughput should be twice that of the single-pipeline system. But the added resources are often accompanied by a reduction in overall utilization. There may be different approaches to utilization modelling:

a) A purely empirical approach would be to experimentally collect the utilization data as a function of the number of resources for the chosen set of benchmarks being

used for performance measurement. This data from a certain machine can be used in future as a guide in performance prediction for a similar machine. This approach has been used in Chapter 6 for generating some utilization curves.

b) One problem with the previous approach is its inability to predict the utilization beyond the range of experimentation. A formal approach to alleviate this drawback would be to characterize the nature of the empirically collected utilization curves with the aim of extracting some key components that might aid in predicting beyond the range of experimentation. For example, the rate of decrease in utilization might exhibit a simple relationship with the number of resources. In a strict mathematical sense, the collected utilization curve can be approximated by a polynomial (using standard approximation procedures). This was the adopted approach in Chapter 2, which assumes a generic polynomial utilization model that is empirically derived.

c) One major drawback with both previous approaches is that they do not offer any useful insight to the system designer. Often a system designer is faced with the question of whether it would be more profitable (in terms of improved throughput) to reduce the dependency delays in the instruction stream and thereby increasing the system utilization, or to simply replicate system resources with reduced utilization. Neither of the previous approaches can resolve such tradeoffs. An alternative approach would be to model such specific utilization related tradeoffs based on some characteristic empirically collected distributions. This approach is illustrated in the following chapter.

4.5 Summary

The analytical model developed in this chapter allows easy, comparative evaluation of superpipelines and superscalars. It is extended in Chapter 6 to include multiprocessors. The parameters contributing to throughput numbers are given in units of gate delays, facilitating model use by IC designers. When validated by measurements on actual systems, the model allows evaluation of possible benefits to be obtained by modest modifications of the basic parameters of circuit organization. With respect to superpipelines and superscalars tradeoff, the model supports the following design rule of thumb: segment a pipeline to the extent possible to improve throughput, then replicate the pipeline for further throughput increases. The performance limit for these systems has been derived and it supports the fetch bottleneck observation of previous researchers. The next chapter provides an analytical model for dependency delays that were ignored in this chapter.

CHAPTER 5
INSTRUCTION-WINDOW SIZE TRADEOFFS AND CHARACTERIZATION OF PROGRAM PARALLELISM

5.1 Introduction

Identifying independent operations that can be scheduled for execution in parallel has always been a key to execution speed enhancement. At the instruction level, detection of concurrent operations begins by examining a consecutive set of instructions from a serial execution sequence, or *instruction stream*. The instruction stream can be analyzed either at compile time or at execution time. The number of instructions simultaneously examined for detecting a concurrent subset is the *scope* of concurrency detection. On computers that do run time concurrency detection, the *instruction window* comprises the set of instructions examined for possible scheduling for simultaneous execution. The larger the scope, the greater the probability of detecting a subset of instructions of a given size that can be scheduled for concurrent execution.

The conditional branch instructions of a program partition it into a collection of *basic blocks*, or instruction stream segments each ending with a conditional branch. A conditional branch directs the execution sequence along one of two or more possible paths and the direction taken is known only at run time. Yet, a consecutive set of instructions of size equal to the desired scope must be available and concurrency detection must precede execution. This dilemma can be overcome by using branch prediction to identify the most likely execution sequence. Concurrency detection can then proceed using the instruction stream as assumed by the branch prediction method.

Conditional branch predictions will err occasionally. Therefore, any concurrency detection scheme that groups operations across conditional branches (i.e., beyond basic blocks) must also have a mechanism to undo the effect of executed operations, if any, that do not lie on the actual execution path.

A variety of studies have been done to assess the performance potential of concurrency detection techniques. While some studies, based on idealistic hardware resource assumptions, report a speedup potential in the range of 50 to 100 [RiF72, NiF84], others report a speedup potential of only 1.5 to 10 for specific architectures and specific sets of applications. In the latter category, studies considering only

within-basic-block concurrency detection, such as those by Tjaden and Flynn [TjF70], Weiss and Smith [WeS84], Acosta, et. al. [AKT86], and Sohi and Vajapeyam [SoV87], report speedup of about 1.5 to 2.5. Wedig [Wed82] and Smith, et. al. [SJH89] assume beyond-basic-block concurrency detection and find potential speedup of about 2 to 4. Acosta, et. al. [AKT86] and Smith, et. al. [SJH89] have also reported the performance impact of instruction window size on dynamic concurrency detection through simulation based techniques. Finally, the studies [KMC72, Lam88, HsD86, CGL89] rely on compile time support to enhance speedup potential and have reported speedups in the range of 4 to 8.

The following section describes the analytic model used to study the performance tradeoffs associated with instruction window size, and introduces a measure of the available amount of parallelism in an instruction stream. In Section 5.3, different costs associated with conditional branches are introduced and a measure of the cost of extracting the available parallelism is defined. Experimental results are presented in Section 5.4. Finally, Section 5.5 describes the issues to improve performance prediction accuracy.

5.2 The Analytic Performance Model

Consider an instruction window of size $W+1$ consisting of a stream of instructions labeled $I_0, I_1, I_2, \ldots, I_W$, where I_0 is the first instruction in the window. A necessary condition for two instructions I_i and I_k to be schedulable for simultaneous execution is that they have no dependencies. An instruction is dependent on another if it uses the result of the other, or if it overwrites a value to be read by the other, or if it overwrites the result of the other. An instruction dependent on another cannot be executed prior to or simultaneously with the instruction it depends upon without changing the meaning of the program. The sufficient condition for scheduling I_i and I_k together is that there must also be no instruction I_j in the instruction stream between I_i and I_k and on which I_k depends. If such an I_j exists, then I_k must execute after that I_j, and by implication, after I_i.

Let $I_{i,j}$ represent the event that instructions I_i and I_j are mutually independent, and let $I_{i,\ldots,k}$ represent the event that instructions I_i through I_k in the instruction stream are pairwise independent. $P(I_{i,\ldots,k})$ denotes the probability of the event $I_{i,\ldots,k}$.

Because instruction I_0 is dispatched unconditionally, consider the remaining instructions in terms of whether they are scheduled together with I_0 or not. Let $I_i{:}y$ represent the event that instruction I_i is scheduled with I_0, and let $I_i{:}n$ represent the event that instruction I_i is not scheduled with I_0. Because I_j can be scheduled with I_0 only if I_j is independent of all the preceding instructions between I_0 and I_j,

$$P(I_j{:}y)=P(I_{j-1,j}, I_{j-2,j}, I_{j-3,j}, \ldots, I_{0,j})$$

$$=P(I_{j-1,j})P(I_{j-2,j} \mid I_{j-1,j})P(I_{j-3,j} \mid I_{j-1,j}, I_{j-2,j}) \cdots P(I_{0,j} \mid I_{j-1,j}, I_{j-2,j}, \ldots, I_{1,j}). \quad \{5.1\}$$

Let $p_{i,k}=P(I_{i,k} \mid I_{j,k}$, for all j such that $i<j<k)$ (see Figure 5.1). This $p_{i,k}$ is the *conditional independence probability* of instructions I_i and I_k. Thus

$$P(I_j{:}y)=\prod_{l=1}^{j} p_{(j-l),j} \ .$$

Assume the instruction stream is a stationary random process, that is, the probability of instruction independence is independent of the instruction window position with respect to the instruction stream, then $p_{i,k}$ is a function only of the distance between I_i and I_k. Hence, $p_{i,k}$ may be written p_δ, where $\delta=k-i$. Then,

$$P(I_j{:}y)=\prod_{\delta=1}^{j} p_\delta \ . \quad \{5.2\}$$

If p_δ is constant, then

$$P(I_j{:}y)=\prod_{\delta=1}^{j} p_\delta=p \cdot p \cdots p=p^j \quad \{5.3\}$$

Note that a constant p_δ does not mean that any two instructions, say I_1 and I_{10}, are equally likely to be independent of a third instruction, say I_0. Rather it does mean that I_1 and I_{10} are equally likely to be independent of I_0, provided that I_{10} is not already dependent on an intermediate instruction, I_m for $0<m<10$. (There is no instruction between I_0 and I_1.) $P(I_j{:}y)$ reflects the influence of both compiler design and hardware resources on inherent program character. However, p_δ is more purely representative a given, fixed program sequence. Here, the program level is assembly language.

The probability that exactly $k-1$ instructions in the window are dispatchable along with I_0 is

$$P_{k-1}(I_1, I_2, \ldots, I_W)=P_{k-2}(I_1, I_2, \ldots, I_{W-1})*P(I_W{:}y)+P_{k-1}(I_1, I_2, \ldots, I_{W-1})*P(I_W{:}n) \quad \{5.4\}$$

At run time, the probability of being able to dispatch at least k instructions is of more interest than the probability of having exactly k dispatchable instructions. The probability of having at least $k-1$ dispatchable instructions in addition to I_0 is

$$P_{\geq k-1}(I_1, I_2, \ldots, I_W)=\sum_{j=k-1}^{W} P_j(I_1, I_2, \ldots, I_W) \ .$$

The above equation can also be written in the following form, which may be more computationally efficient

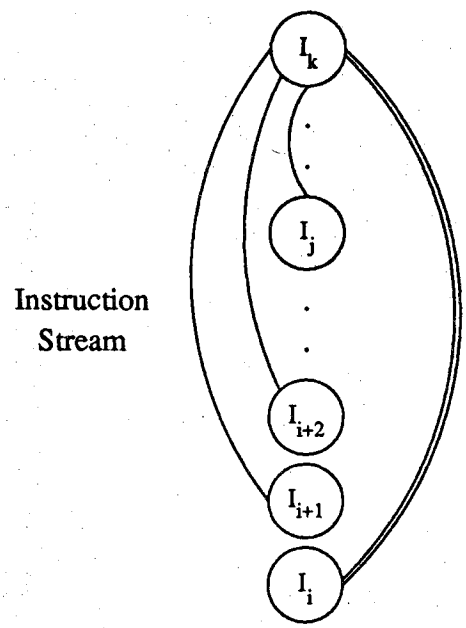


Figure 5.1 Illustration of dependencies determining conditional independence probability, $p_{i,k}$. Each single arc indicates a pair of instructions that are given to be independent. The double arc denotes the dependence in question for $p_{i,k}$.

$$P_{\geq 0}(I_1, I_2, ..., I_W) = 1$$

$$P_{\geq k-1}(I_1, I_2, ..., I_W) = 1 - \sum_{j=0}^{k-2} P_j(I_1, I_2, ..., I_W) \text{ , for } k > 1 \text{ .}$$

Figures 5.2 through 5.4 depict $P_{\geq k-1}$ as a function of p_δ and W. The following observations can be made:

1) Figure 5.2 shows that a given variation in p_δ, for higher (lower) values of p_δ becomes increasingly more (less) crucial as k grows. For example, a compile time effort to increase p_δ (say, by register renaming) from 0.55 to 0.65 while quite noticeable when there are three dispatchable instructions (k=3), is almost unnoticeable for k=5. An increase in p_δ from 0.75 to 0.85 although unnoticed for k=2, is significant when k=4.

2) Larger window size can only be justified with an accompanying compile time effort to reduce dependence by increasing the conditional independence probability, as is evident from Figures 5.3 and 5.4.

Plots such as in Figures 5.2 through 5.4 can be useful in isolating execution performance bottlenecks. Based on the operating point, the plots reveal whether the bottleneck is insufficient inherent parallelism in the stream (low p_δ, suggesting more compiler effort for reducing instruction dependencies), or not examining enough instructions (suggesting increased window size which might further indicate the need to do beyond-basic-block scheduling), or insufficient resources for utilizing available parallelism (suggesting good payoff for additional hardware). The operating point for a new processor design can be determined at an early stage, so Figures 5.2 through 5.4 can be useful in guiding the design effort. Sometimes loop unrolling is used at compile time, to increase the scope of concurrency detection. There is no performance gain in unrolling beyond the point where scope of concurrency detection is not a performance bottleneck any more. Therefore, the information from figures such as Figures 5.2 through 5.4, can also be used to limit the amount of unrolling.

Figures 5.2 through 5.4 are based on the assumption that p_δ is constant. This may be an inaccurate assumption for many programs. Near successors of an instruction are more likely to be dependent on it than the instructions further removed. Thus, p_δ is expected to rise with δ for small values of δ. But, beyond the immediate vicinity, i.e., for large values of δ, p_δ may be fairly constant.

Consider a 4-pipeline superscalar processor system. If due to dependency constraints, only two operations can be issued for 30 percent of the time, then the system behaves effectively as a 2-pipeline system 30 percent of the time. Thus, the effective throughput, G_k, under dependency constraints for a k-pipeline processor is

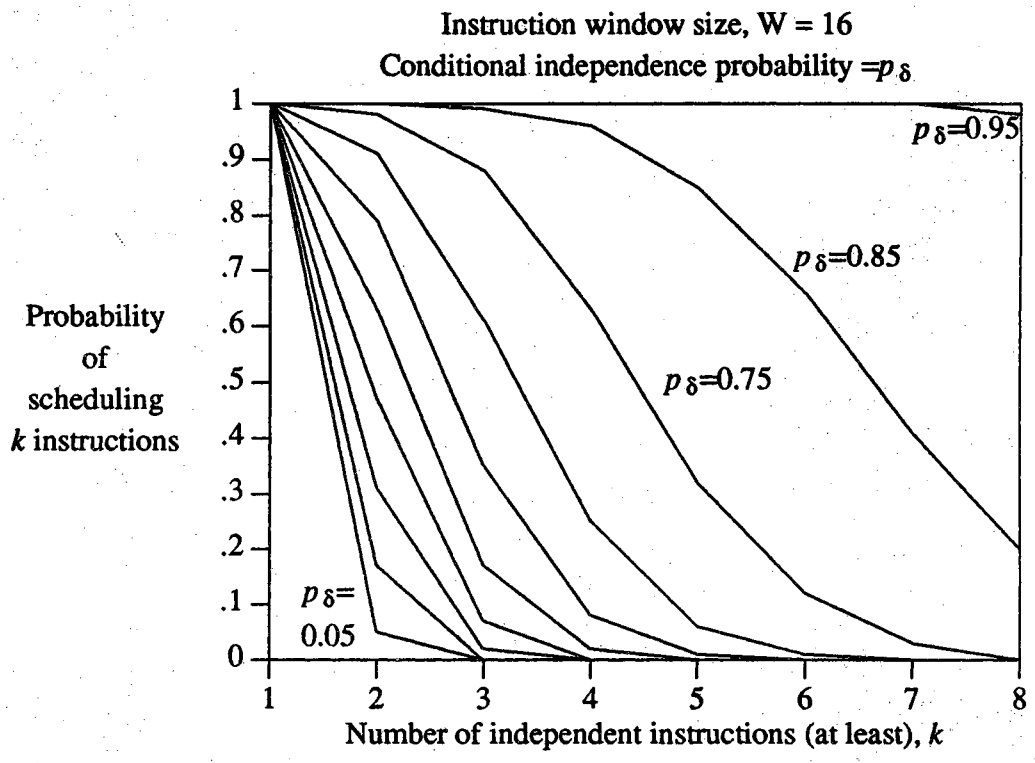


Figure 5.2 Probability of scheduling k instructions for various values of p_δ and a fixed instruction window size of 16.

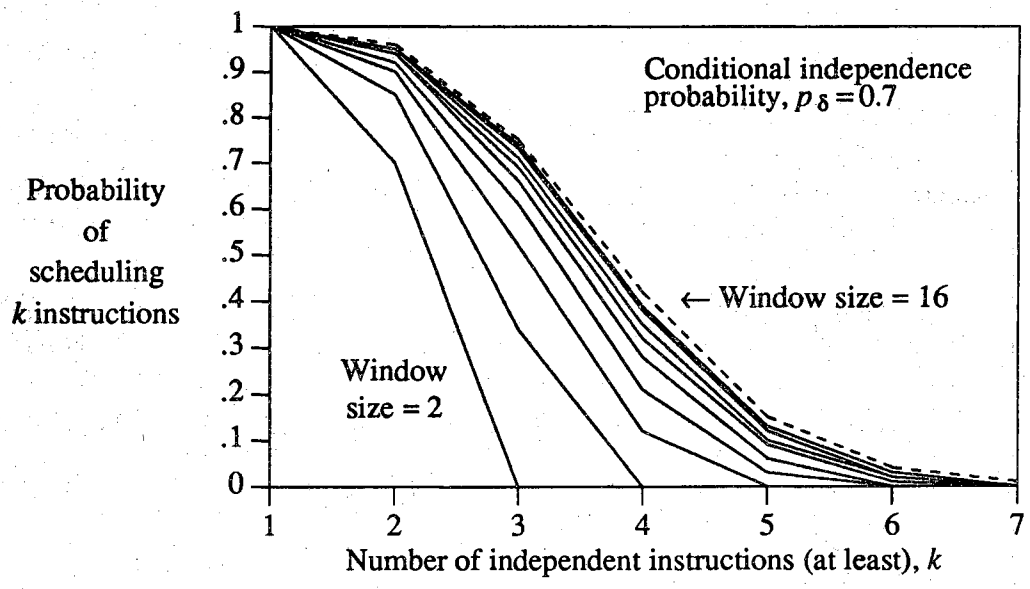


Figure 5.3 Probability of scheduling k instructions for various instruction window sizes and $p_\delta = 0.7$.

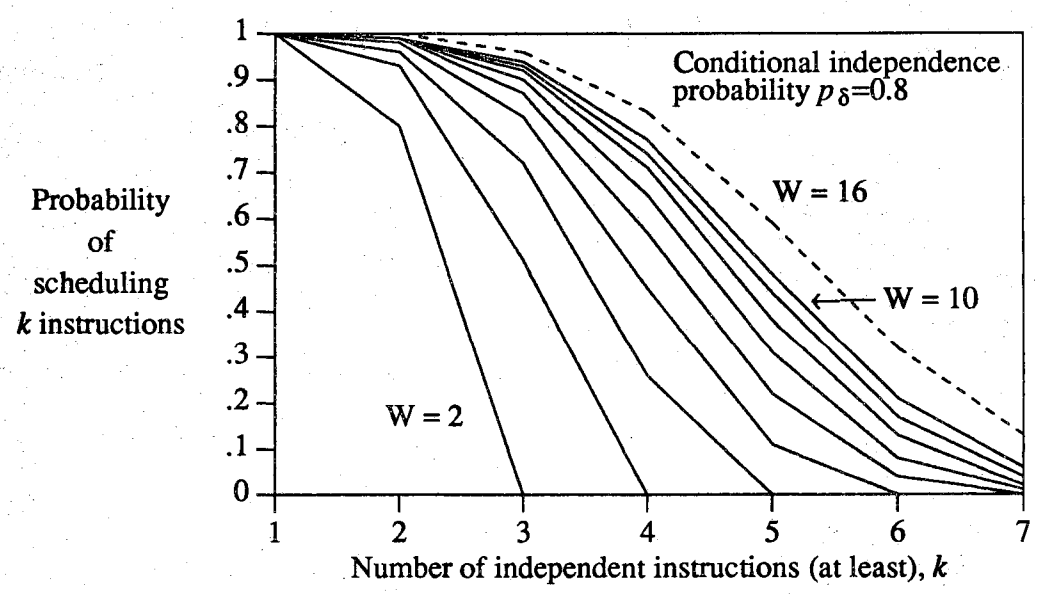


Figure 5.4 Probability of scheduling k instructions for various instruction window sizes and $p_\delta = 0.8$.

$$G_k = \sum_{i=0}^{k-2} g_{i+1} P_i (I_1, I_2, ..., I_W) + g_k P_{\geq k-1} (I_1, I_2, ..., I_W) \ , \qquad \{5.5\}$$

where g_j, for $1 \leq j \leq k$, is the throughput for a j-pipeline processor calculated ignoring any dependency constraint (always having j schedulable instructions), but including such delays as cache misses and conditional branch resolutions. This is same as the throughput computed using Equation (4.4) of Chapter 4.

5.3 Cost of Branches

Let b be the probability that an instruction is a branch instruction. On every clock cycle an instruction packet consisting of k instructions is fetched. Let the cache line size be a multiple of k. If instruction words have fixed length, then in the absence of any branches, all instruction references will be aligned and a fetch bandwidth of k instructions per cycle will be sustained.

The cost of branches may be categorized as follows:

a) *Misprediction delay*. Every time a branch prediction is incorrect, a certain delay is incurred. Let D_i be the average delay associated with each misprediction. In the case of out-of-sequence, beyond-basic-block execution, a branch misprediction means that more than just the execution pipeline may contain incorrect execution: instructions from much earlier may need to be undone. So, misprediction delay may be significant.

b) *Wasted fetch delay*. Every time a branch is detected, the remaining instructions that are part of the packet of k instructions may be wasted. (Delayed branching may reduce this waste.) The wasted execution bandwidth corresponding to these instructions is added as a delay to the branch instruction. Since the total execution bandwidth of a packet of k instruction is 1 clock, the wasted bandwidth of the last j instructions in a packet is j/k. This delay has been studied with respect to a variety of branch strategies for the single pipeline case in Chapter 3 (also in [DuF89, DuF91]).

c) *Misalignment delay*. For normal execution, assume that a cache line can be fetched from the instruction cache every cycle. An instruction reference is considered *misaligned*, and hence requiring an additional fetch, if the group of k instructions spans a line boundary. Every time a branch is predicted to a target such that the corresponding group of k instructions spans a line boundary, a delay of an extra clock results because only one cache line can be read at a time.

d) *Cache miss delay.* The cache miss probability for the branch target may be somewhat more than the typical cache miss probability on instruction fetches.

5.3.1 Calculating Misprediction Delay Resulting from Speculative Execution

Scheduling techniques using a concurrency detection scope extending beyond a basic block incur additional delay for an incorrect prediction because they need to undo the damage, if any, caused by execution of instructions outside the current basic block. The cost of undoing a wrongfully executed instruction is dependent on the specific implementation support for damage undoing, and can be considered independent of the specific instruction type. For example, instructions that allow updates to user memory (interface space) before branch resolution would need to restore the incorrectly updated locations. Let the time cost of undoing the damage of a wrongfully executed instruction be μ cycles.

Consider a program tree where each node represents a basic block and imagine following a program trace using some branch prediction mechanism (see Figure 5.5). Let p_ω be the probability that an instruction is scheduled with an instruction from a basic block that is ω levels up the program tree. Assume p_ω is independent of depth in the program tree. Also assume that the scope of concurrency detection extends to the end of the program (feasible at compile time but not at run time). The branch misprediction cost, D_i^j, associated with the basic block that is j levels deep is

$$D_i^j = \sum_{n=j+1}^{N} \sum_{\omega=n-j}^{n-1} \frac{p_\omega u}{b} , \qquad \{5.6\}$$

where N is the average program tree depth and $1/b$ is the expected size of a basic block. For example, suppose the conditional branch prediction associated with the basic block five levels deep is in error. This implies that all the instructions scheduled from the basic block at depth six and below to basic blocks at depths 5, 4, 3, 2, and 1 need to be undone. Assuming conditional branch prediction at any depth is equally likely to be in error, the average cost of a misprediction is

$$D_i = \frac{1}{N} \sum_{j=1}^{N} \sum_{n=j+1}^{N} \sum_{\omega=n-j}^{n-1} \frac{p_\omega u}{b} . \qquad \{5.7\}$$

Because the probability of scheduling an instruction x levels up cannot exceed that of scheduling $x-1$ levels up, p_ω must be a monotonically decreasing function of ω. Also, since $\sum_{\omega=0}^{N} p_\omega = 1$, p_ω must be a nonlinear function of ω. Assume p_ω can be approximated by a truncated geometric distribution with parameters K and empirically determined q such that

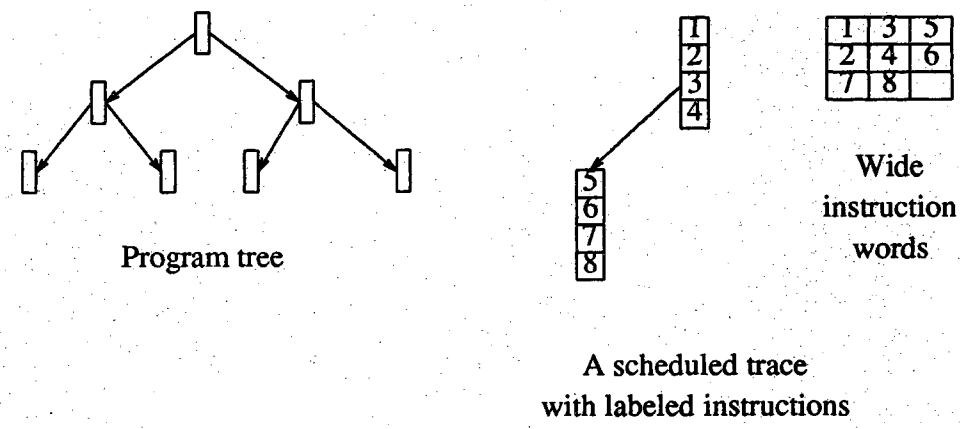


Figure 5.5 Illustration of a program tree, a scheduled trace of execution, and the assembly of wide instruction words with beyond-basic-block scheduling.

$$p_\omega = Kq^\omega (1-q)\ , \text{ where } K = \frac{1}{\sum_{\omega=0}^{N} q^\omega (1-q)}\ .$$

Assuming $K=1$ (as would be the case for large N), intuitively, $(1-q)$ represents the probability of within-basic-block scheduling for an instruction. The experimentally collected results for p_ω for the set of benchmarks exhibit similar distribution characteristics. Thus, Equation (5.7) can be rewritten as

$$D_i = \frac{1}{N} \sum_{j=1}^{N} \sum_{n=j+1}^{N} \sum_{\omega=n-j}^{n-1} \frac{K q^\omega (1-q) u}{b}\ .$$

In closed form

$$D_i = K u \frac{q}{N b (1-q)} \left[N - \frac{q(1-q^N)}{1-q} - \frac{q^{N-1}(q^{-N}-1)}{q^{-1}-1} + N q^N \right]\ . \qquad \{5.8\}$$

Figure 5.6 illustrates D_i as a function of program tree depth, N. For values of q in the range of 0.1 to 0.6 the average misprediction delay, D_i, is essentially constant for $N \geq 20$. Even for q as large as 0.8, D_i is nearly stable for $N \geq 50$. Since the scope of concurrency detection in Equation (5.7) is assumed to be infinite (extending up to the end of the program), D_i as computed above is an *upper* bound. Even in the worst case the average branch misprediction delay is about the same as a typical cache miss processing delay, 10 to 20 clocks.

5.3.2 Alternate Computation for p_ω

Assuming fixed size basic blocks of size $B = \lfloor 1/b + 0.5 \rfloor$, p_ω can be computed from the $P(I_j{:}y)$:

$$p_\omega = \frac{1}{B} \sum_{i=1}^{B} \sum_{j=i+(\omega-1)B}^{i+\omega B-1} P(I_j{:}y)$$

For example, assume b=0.2. Consider the second to last instruction (i=2) in a basic block. If it is scheduled at a distance of 7 to 11, it has been scheduled past two unresolved branches. Hence, contribution to p_ω by the second to last instruction, ω=2 is $0.2 \sum_{j=2+5}^{2+10-1} P(I_j{:}y)$.

The parameter p_ω computed as above would not be as accurate as that empirically collected, because the above calculations are based on the very simplistic assumption of fixed basic block size. This can be improved by using the basic block size distribution instead. Assuming p_δ (and hence $P(I_j{:}y)$) to be independent of the size of basic blocks,

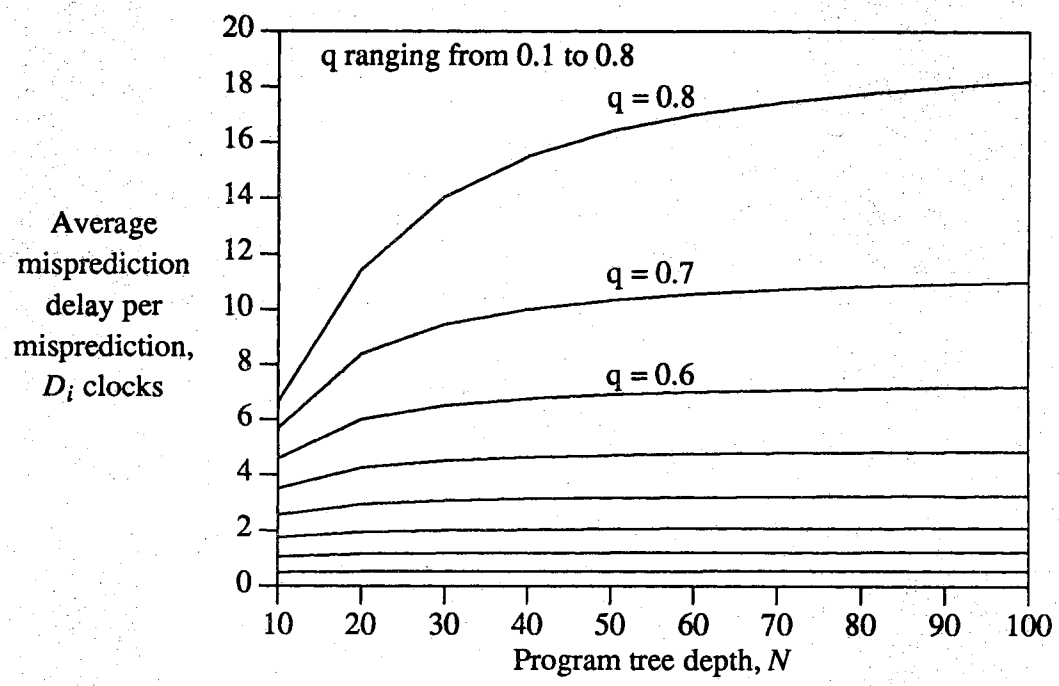


Figure 5.6 Average misprediction delay versus program tree depth for branch frequency, b = 0.2, average cost of damage undoing per percolation, μ = 1, and various percolation-distance distribution parameter, q, values. The parameter q is a measure of beyond-basic-block scheduling probability.

p_ω computed using basic block size distribution should be approximately same as that empirically collected. Thus an alternate characterization of parallelism can be in terms of p_δ and basic block size distribution instead of p_ω.

5.3.3 Dynamic Scheduling with Finite Lookahead

Since a machine can only dedicate a finite amount of chip area for keeping the history of speculatively executed operations, there would be a limit to the amount of look ahead in terms of basic blocks. Let L be the scope of look ahead measured in number of basic blocks ($L=0$ for within-basic-block scheduling). Then the average misprediction delay is approximately

$$D_i = \sum_{j=1}^{L} \sum_{\omega=j}^{L} \frac{\hat{p}_\omega u}{b} \quad . \qquad \{5.9\}$$

where $\hat{p}_\omega$ represents the p_ω distribution truncated at a distance of L basic blocks.

A finite lookahead in terms of basic blocks also implies that the size of instruction window, W, is a variable, as it gets truncated to the size of L basic blocks whenever the combined size of L pending basic blocks is less than W instructions. The distribution for W in such a case, can be computed using the basic block size distribution.

5.4 Experimental Results

There are two key input parameters in the model developed in the previous two sections: p_δ and p_ω. The parameter p_δ provides a measure of how often two instructions at positions δ apart in the instruction stream are found to be independent. As noted, two independent instructions at a fixed distance δ may have a very different cost for simultaneous scheduling, depending on their distance as measured in basic blocks. The parameter p_ω captures this additional cost, giving a more realistic performance estimate.

Experiments have been conducted on a set of benchmarks (see Table 1) using the Multiflow TRACE SCHEDULING compacting C and Fortran 77 compiler on a TRACE computer. This compiler [Fis81] does out-of-order, beyond-basic-block scheduling. The goal of these experiments was two-fold. First, to establish the nature of the p_δ and p_ω parameters and to determine their capability for characterizing program parallelism; second, to show that this characterization can be used to predict their performance under scope and resource constraints.

Table 5.1 Benchmarks used in this study.

Benchmark	Description
stanford	Collection of various application programs also known as *Stanford Integer Suite*
spice	Analog Circuit Simulation Package
fpppp	Quantum chemistry benchmark that measures performance on a two-electron integral derivative computation
tair	Transonic airfoil analysis program
applu	Coupled partial differential equations
cgm	Conjugate gradient solver
fftpde	3-D FFT PDE
mgrid	Simple multigrid solver
mdg	Driver for molecular dynamic simulation of flexible water molecule
mg3d	Nonlinear algebraic systems solvers and ODE solvers for signal processing
bdna	ODE solvers for chemical and physical models

The hardware of the TRACE 28/200 [CNO88] includes four processor boards, each containing of two integer ALUs, one floating point adder, and one floating point multiplier. It can initiate 28 operations per instruction. Thus, the collected values for p_δ and p_ω are not resource constrained when considering less than the available number of resources of the TRACE 28/200.

The Multiflow compiler provides the ability to generate detailed trace schedules, such that the operations being grouped are also tagged to indicate their position in the original sequential instruction stream. This information is used to calculate the *scheduling probability*, $P(I_j:y)$, (see Section 5.2). Collected traces for the benchmarks, are post-processed to simulate a run time scheduling environment. The target environment assumes that on every scheduling cycle W instructions from the dynamic stream of instructions are examined for dependency. Those found independent, say k ($< W$), are scheduled together and dispatched, and another $W-k$ instructions are moved into the instruction window. The post-processing consists of following phases:

1) renumbering instructions in a trace to represent a continuous dynamic sequence,
2) dynamically adjusting distances between instructions as the execution proceeds, and
3) weighing data from each routine in proportion to the fraction of run time spent in that routine.

Figures 5.7 through 5.9 plot $P(I_j:y)$ for the chosen set of benchmarks. Note the distinctive nature of the *fpppp* benchmark. Unlike the others, it has a relatively small scheduling probability for adjacent instructions ($\delta=1$) and has a small but non-negligible probability of scheduling even at distances of more than 512.

The p_δ values can be obtained from the scheduling probabilities in the following manner. Restating Equation (5.1),

$$P(I_j:y)=P(I_{j-1,j})P(I_{j-2,j} \mid I_{j-1,j}) \cdots P(I_{1,j} \mid I_{j-1,j},\ldots,I_{2,j})P(I_{0,j} \mid I_{j-1,j},I_{j-2,j},\ldots,I_{1,j}) .$$

Assuming a stationary distribution yields

$$P(I_j:y) = P(I_{j-2,j-1})P(I_{j-3,j-1} \mid I_{j-2,j-1}) \cdots P(I_{0,j-1} \mid I_{j-2,j-1},\ldots,I_{1,j-1})P(I_{0,j} \mid I_{j-1,j},I_{j-2,j},\ldots,I_{1,j})$$

$$=P(I_{j-1}:y)P(I_{0,j} \mid I_{j-1,j},I_{j-2,j},\ldots,I_{1,j}) .$$

Therefore,

$$P(I_{0,j} \mid I_{j-1,j},I_{j-2,j},\ldots,I_{1,j})=\frac{P(I_j:y)}{P(I_{j-1}:y)} \qquad \{5.10\}$$

$$=p_\delta , \text{ for } \delta=j .$$

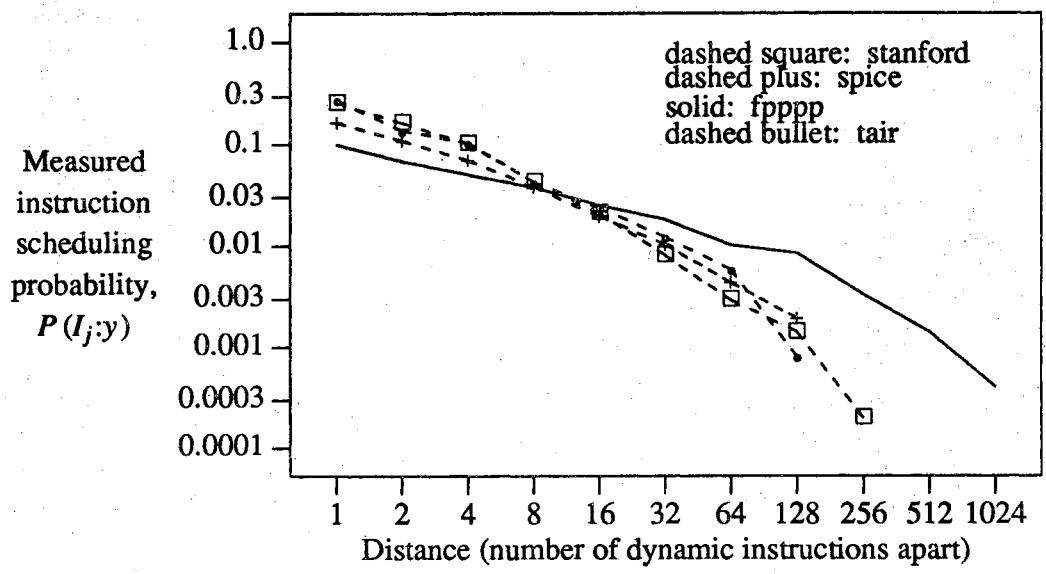


Figure 5.7 Measured instruction scheduling probability versus distance for the stanford, spice, fpppp, and tair benchmarks.

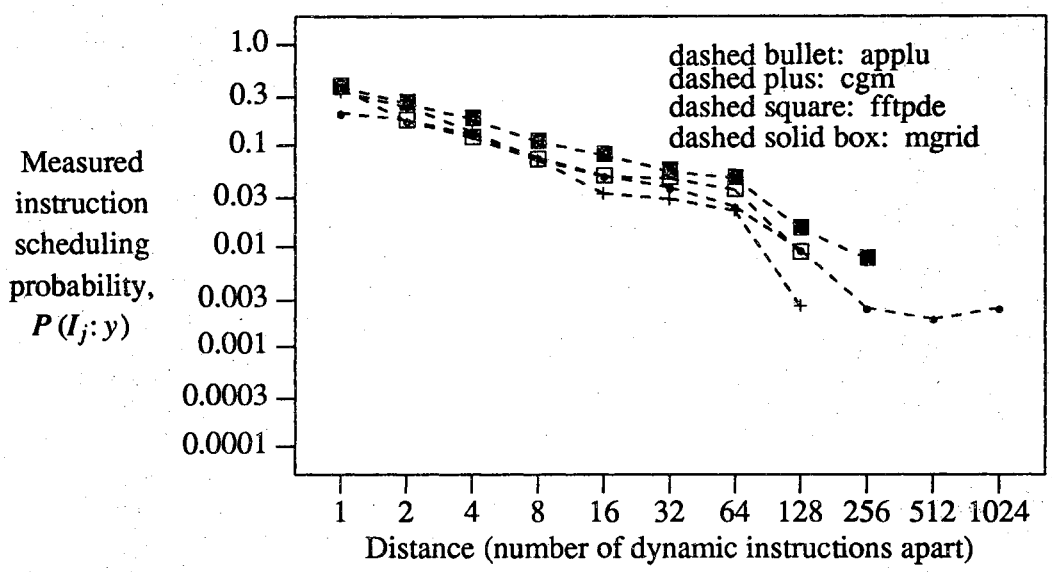


Figure 5.8 Measured instruction scheduling probability versus distance for the applu, cgm, fftpde, and mgrid benchmarks.

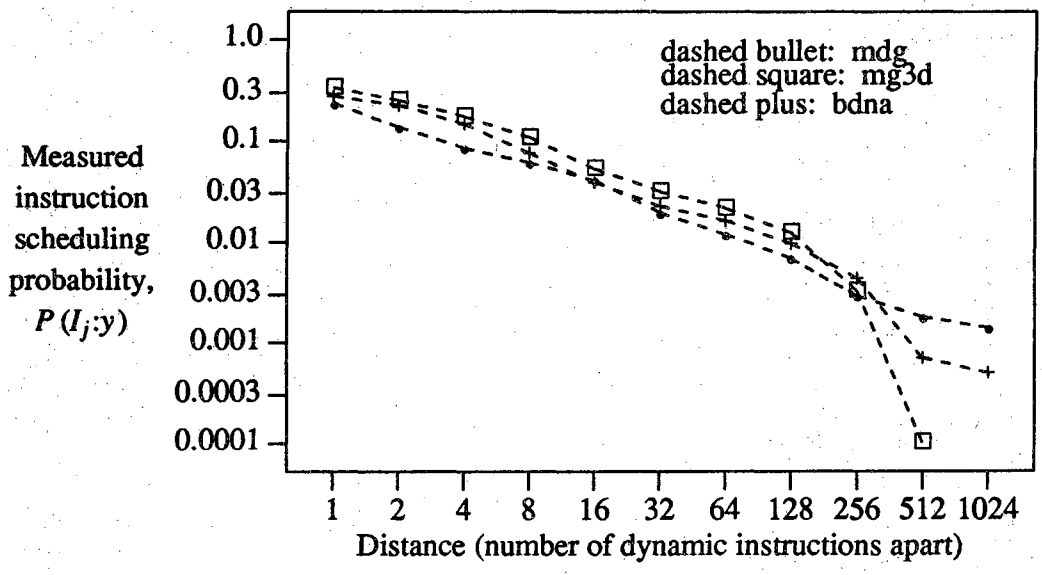


Figure 5.9 Measured instruction scheduling probability versus distance for the mdg, mg3d, and bdna benchmarks.

Figures 5.10 through 5.12 plot the p_ω distribution for the chosen set of benchmarks. The p_δ plot alone is insufficient justification for increased scope for a benchmark; the associated misprediction delay cost given by the p_ω distribution must also be considered. Figures 5.13 through 5.15 provide an estimate of this cost for the chosen set of benchmarks using empirically computed branch probability, b and assuming cost of damage undoing, $\mu = 1$, in Equation (5.9). The worst case misprediction delay is around 20 clocks, as estimated earlier from the analytical calculations in the previous section. Note that for *fpppp* more than 90 percent of time the scheduled instructions are in the same basic block, resulting in very small misprediction cost. Therefore, this benchmark would benefit most from a large scope. On the other hand, although one might be tempted to increase the scope for *tair*, by only looking at the $P(I_j{:}y)$ plot, the misprediction delay estimate (Figure 5.13) for this program would be a strong deterrent to such a decision. Thus, p_δ, the conditional instruction independence probability, and p_ω, a measure of the cost of speculative execution, together provide a complete picture in terms of the amount of available parallelism and the cost of its extraction, respectively.

For programs where the Multiflow compiler generates many short traces probability calculations for larger distances become inaccurate. For example, assume there is one instance when an instruction a distance of 128 was examined for dependence and it was found independent. If such a small sample is used to calculate the scheduling probability at this distance, then the scheduling probability at a distance of 128 would be assigned a probability of one, which is obviously an erroneous conclusion. A simple fix for this problem is to ignore probabilities calculated having too small number of sample data points. To incorporate this fix, of all the SPEC*, NAS Parallel benchmark suite [BBL91] and Perfect benchmarks [CKP90], only the benchmarks** that had at least 200 sample data points in their growing throughput range have been selected. (This is why not all of the SPEC benchmarks, NAS Parallel and Perfect benchmarks are members of the chosen set of benchmarks.)

Figures 5.16 through 5.26 plot the throughput calculated with the model and the measured static throughput (average width of instruction word) from the compiler output as a function of scope of concurrency detection and instruction word width. Model throughput is calculated using the p_δ values input to the analytical model developed in Section 5.2. Measured static throughput is estimated as the average width of schedules (wide instruction words) in the traces output by the Multiflow compiler. The compiler throughput estimate does not take into account run time delays, such as memory delays,

---

*SPEC is a trademark of the Systems Performance Evaluation Cooperative. Purdue University SPEC License No. 310.

**The SPEC benchmarks and NAS parallel benchmarks were compiled using version 1.6.1 of the TRACE C and FORTRAN 77 compilers. The Perfect club benchmarks were compiled using version 2.2 of the TRACE FORTRAN 77 compiler

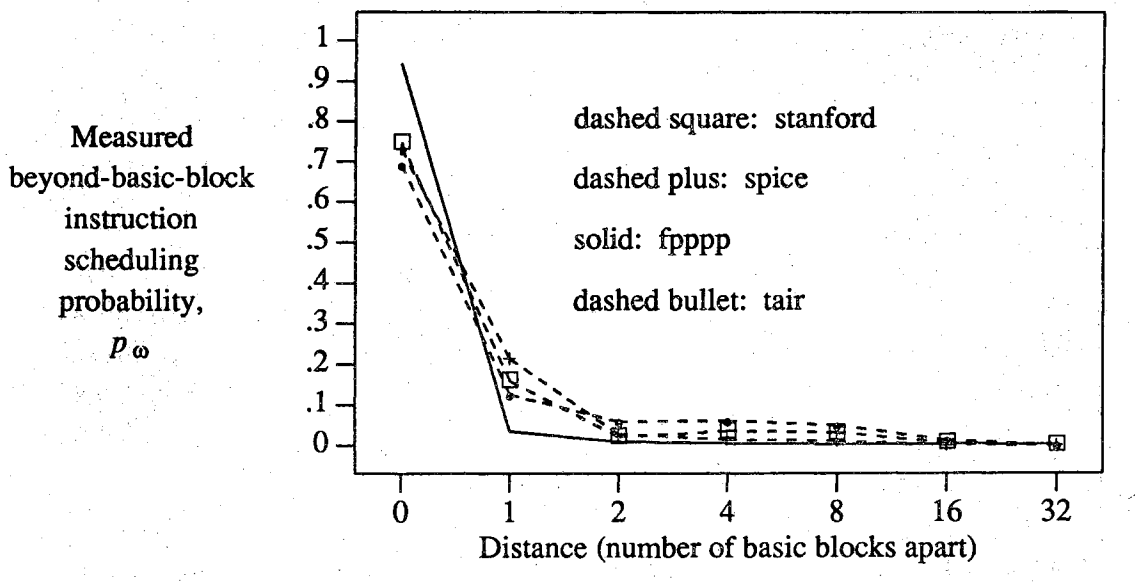


Figure 5.10 Measured beyond-basic-block instruction scheduling probability versus distance for the stanford, spice, fpppp, and tair benchmarks.

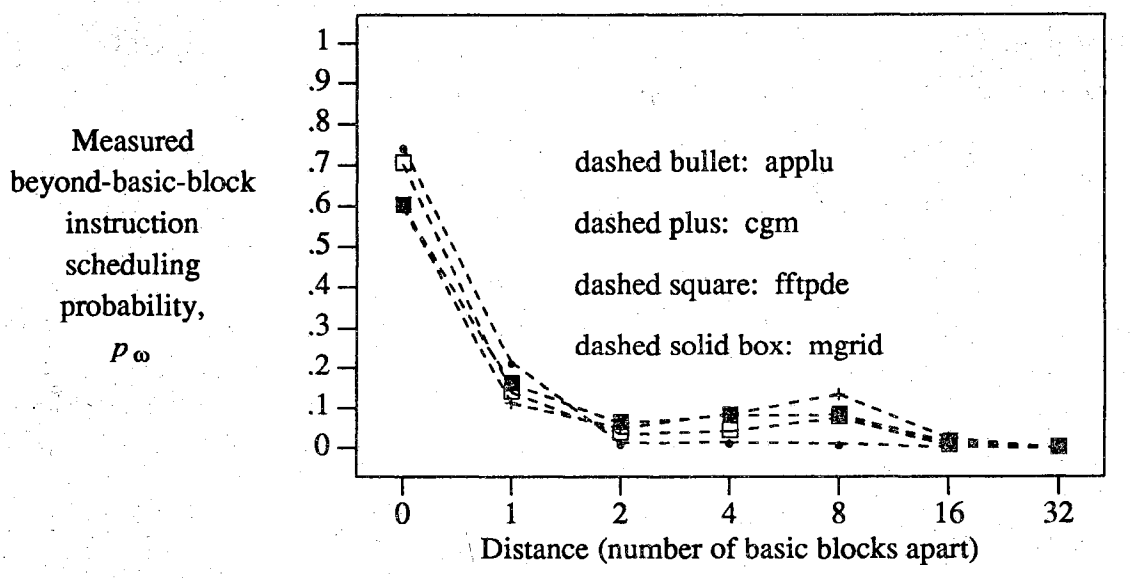


Figure 5.11 Measured beyond-basic-block instruction scheduling probability versus distance for the applu, cgm, fftpde, and mgrid benchmarks.

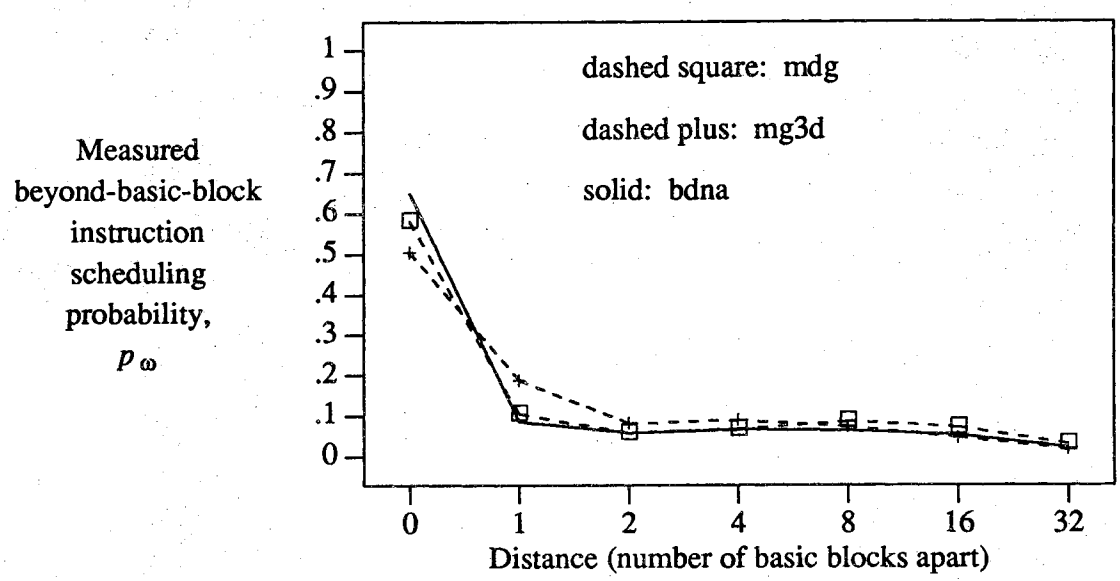


Figure 5.12 Measured beyond-basic-block instruction scheduling probability versus distance for the mdg, mg3d, and bdna benchmarks.

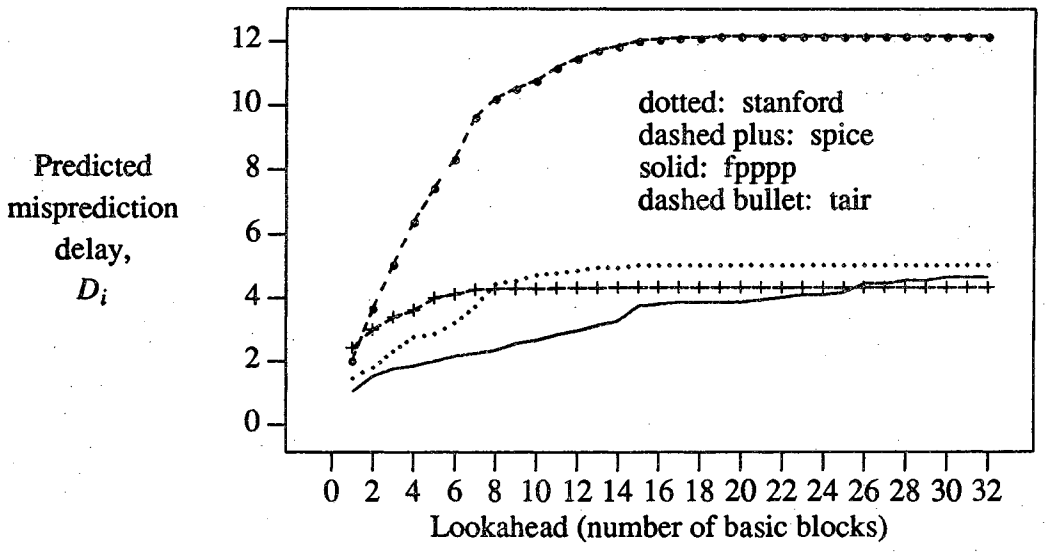


Figure 5.13 Predicted misprediction delay based on the empirically collected p_ω distribution as a function of the amount of dynamic lookahead, in terms of number of basic blocks for the stanford, spice, fpppp, and tair benchmarks.

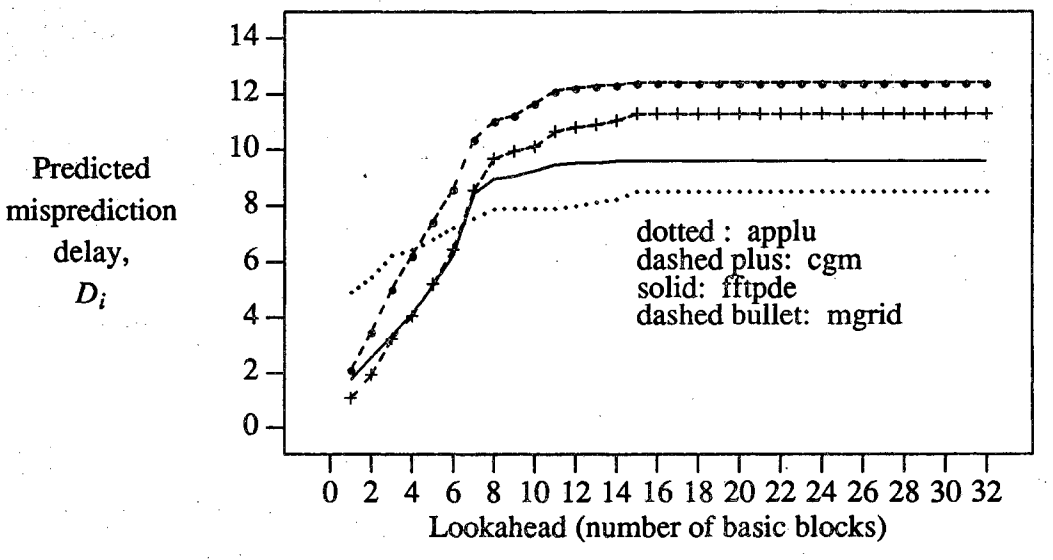


Figure 5.14 Predicted misprediction delay based on the empirically collected p_ω distribution as a function of the amount of dynamic lookahead, in terms of number of basic blocks for the applu, cgm, fftpde, and mgrid benchmarks.

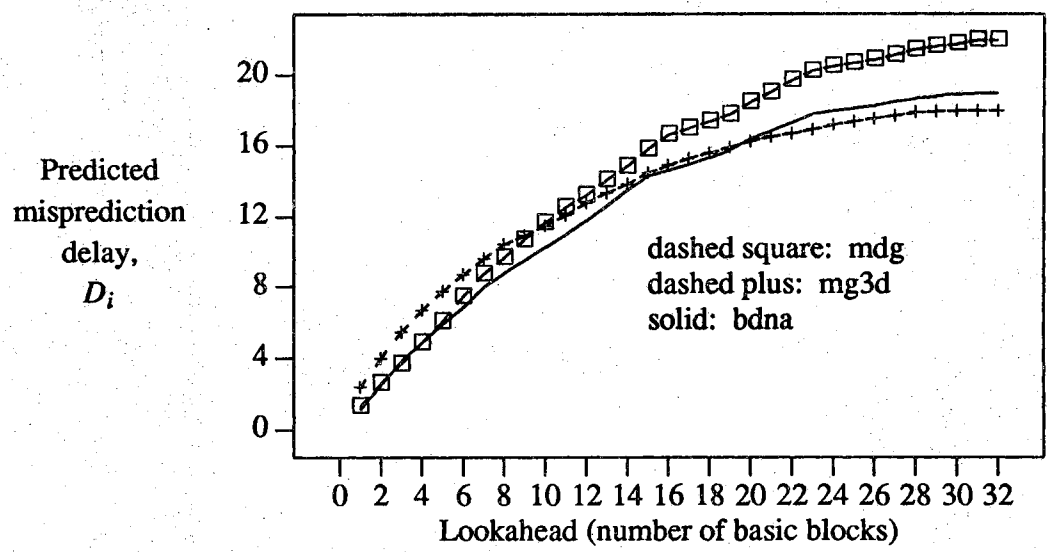


Figure 5.15 Predicted misprediction delay based on the empirically collected p_ω distribution as a function of the amount of dynamic lookahead, in terms of number of basic blocks for the mdg, mg3d, and bdna benchmarks.

which are not part of the analytic model either. An important reason for the discrepancy between the model prediction and the compiler output is due to the basic difference between superscalar and *VLIW* machines. The model is based on a superscalar architecture. Consequently, two instructions that are data-independent of each other are always assumed schedulable on two available resources (pipelines). But in a VLIW environment, such as that of Multiflow, there are additional resource restrictions, as each functional unit is not a complete execution pipeline. For example on a VLIW machine, two independent floating point adds may be forced to wait if only integer adders are available. Such resource constraints are not part of the analytical model. Finally, note that for a window size of 32, the average difference between the model and measured compiler output is 20 percent and the worst case difference is 43 percent; whereas, for a window size of 1024, the average and worst case differences are 47 percent and 79 percent respectively. More importantly, the throughput curves for both the model and the measured values have very similar shape.

Experience with these benchmarks confirms that the longer the traces, the more data points and hence more credible the probabilities and better the performance prediction. For example, *tair* and *fpppp* benchmarks gave relatively longer traces and had better performance prediction. Almost all the benchmarks (an important exception being *fpppp*) attain almost all of the speedup with a scope of about 64 instructions and an instruction word width of 6.

Figures 5.7 through 5.26 have been plotted with distance up to 1024 instructions and for 32 basic blocks of lookahead. Current microprocessors such as, 80x86 or RS6000, however, are just beginning to explore the tradeoffs associated with beyond-basic-block (speculative) execution. Hence, the scope used by near-future generations of such machines is likely to be limited to a few basic blocks and an instruction window size of at most 16 to 32 instructions. With this in mind, , Appendix B contains performance plots for several other benchmarks (Table B.1) to a reduced range of lookahead. Although the benchmarks in Appendix B did not have more that 200 sample data points in their entire speedup range (the previous selection criterion), they all have more than 200 sample data points for a distance of 32 instructions or less. The graphs for these benchmarks are limited in scope to 32 instructions and eight basic blocks (Figures B.1 to B.22).

5.5 Potential Improvements to the Model

There are three predominant sources of inaccuracy in the performance predictions of the analytic model.

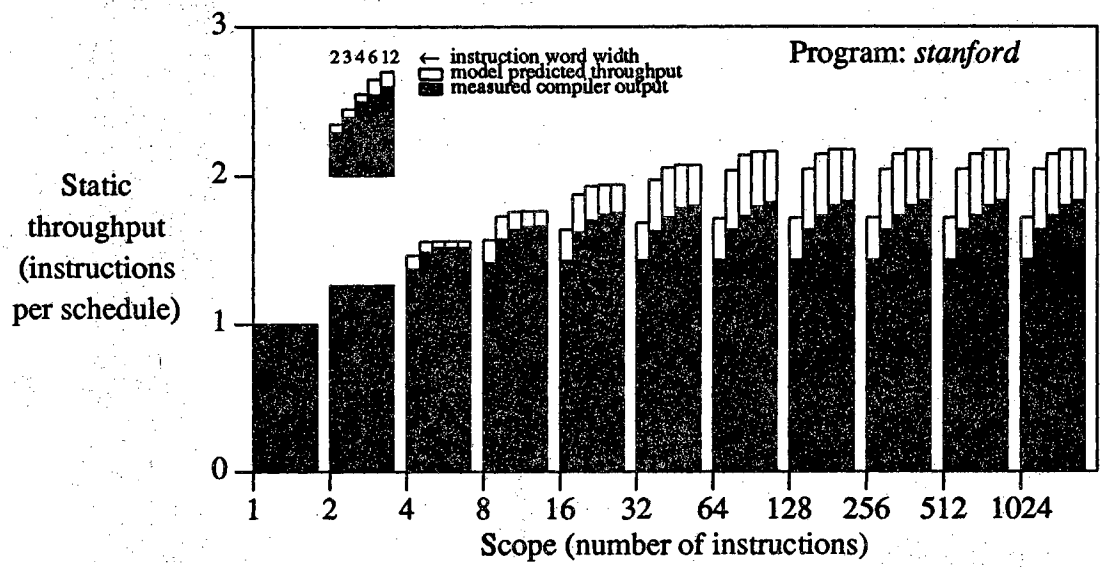


Figure 5.16 Throughput under resource and scope constraints for the *stanford* benchmark; resources varied with instruction word width equal to 2,3,4,6, and 12. The compiler output was influenced by resource constraints that are not part of the model.

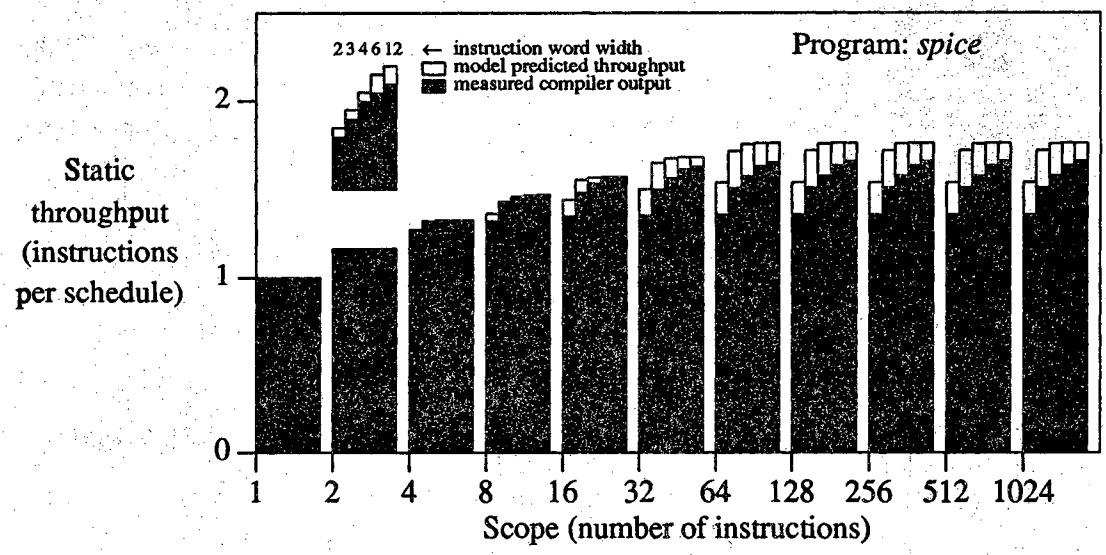


Figure 5.17 Throughput under resource and scope constraints for the *spice* benchmark; resources varied with instruction word width equal to 2,3,4,6, and 12. The compiler output was influenced by resource constraints that are not part of the model.

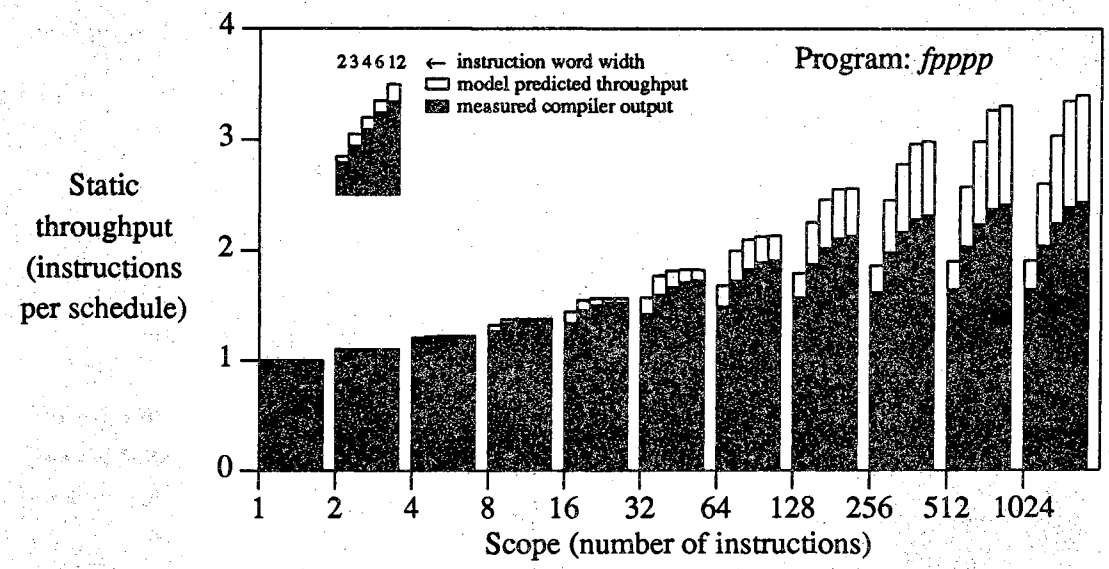


Figure 5.18 Throughput under resource and scope constraints for the *fpppp* benchmark; resources varied with instruction word width equal to 2,3,4,6, and 12. The compiler output was influenced by resource constraints that are not part of the model.

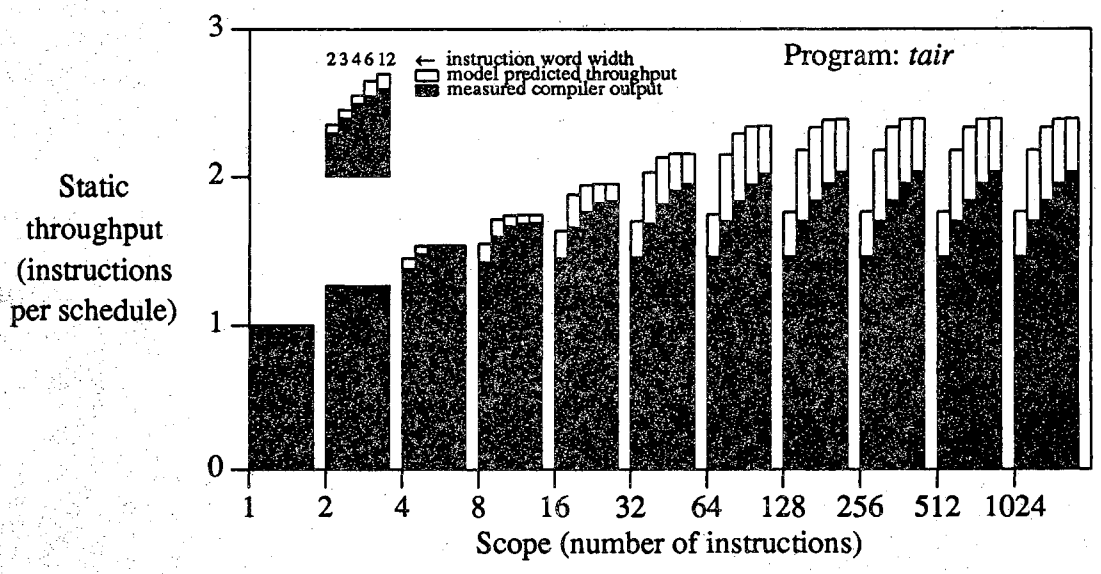


Figure 5.19 Throughput under resource and scope constraints for the *tair* benchmark; resources varied with instruction word width equal to 2,3,4,6, and 12. The compiler output was influenced by resource constraints that are not part of the model.

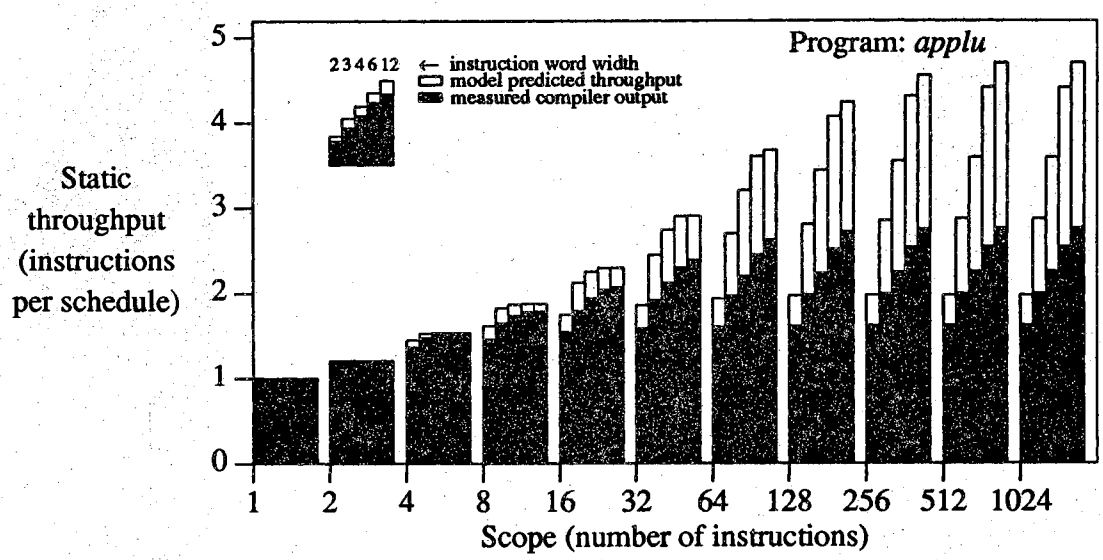


Figure 5.20 Throughput under resource and scope constraints for the *applu* benchmark; resources varied with instruction word width equal to 2,3,4,6, and 12. The compiler output was influenced by resource constraints that are not part of the model.

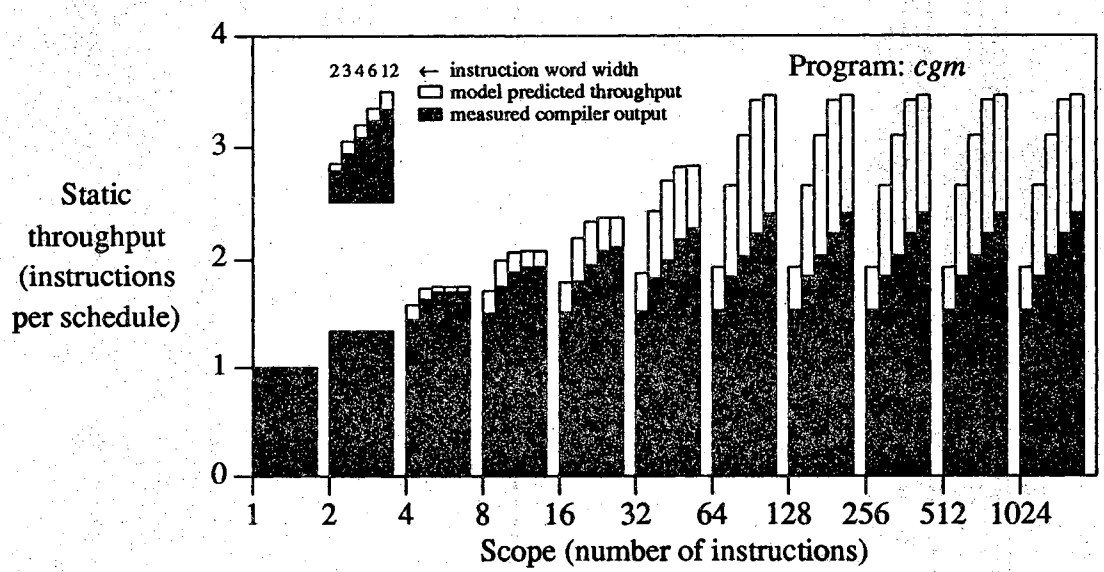


Figure 5.21 Throughput under resource and scope constraints for the *cgm* benchmark; resources varied with instruction word width equal to 2,3,4,6, and 12. The compiler output was influenced by resource constraints that are not part of the model.

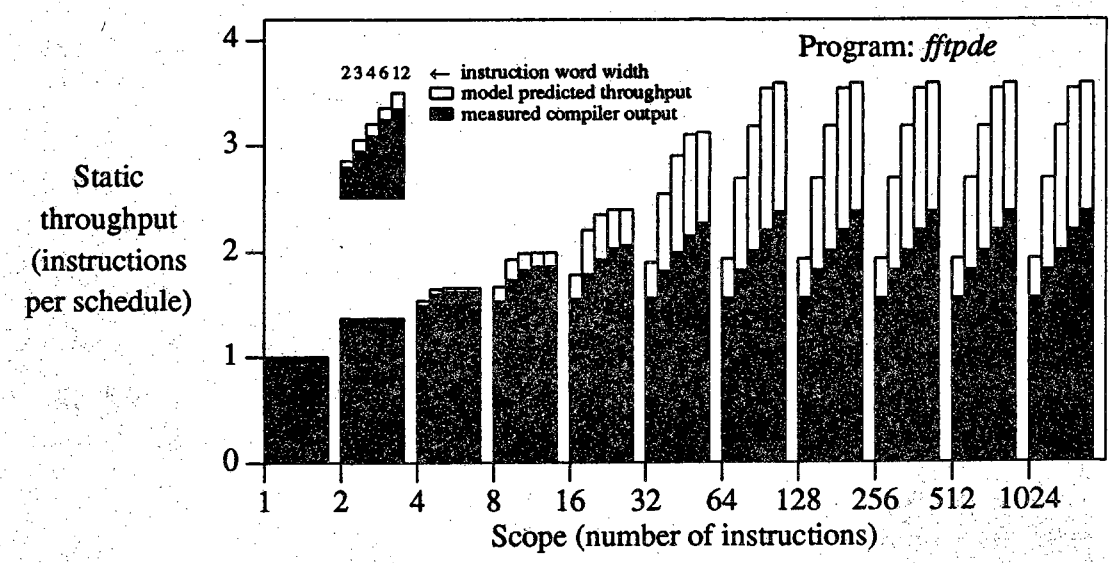


Figure 5.22 Throughput under resource and scope constraints for the *fftpde* benchmark; resources varied with instruction word width equal to 2,3,4,6, and 12. The compiler output was influenced by resource constraints that are not part of the model.

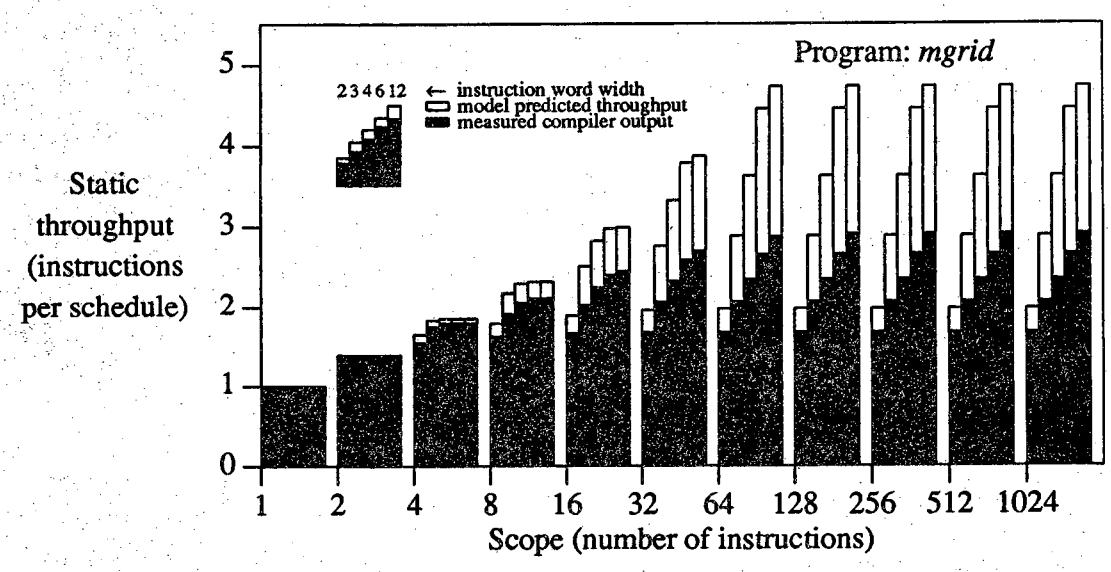


Figure 5.23 Throughput under resource and scope constraints for the *mgrid* benchmark; resources varied with instruction word width equal to 2,3,4,6, and 12. The compiler output was influenced by resource constraints that are not part of the model.

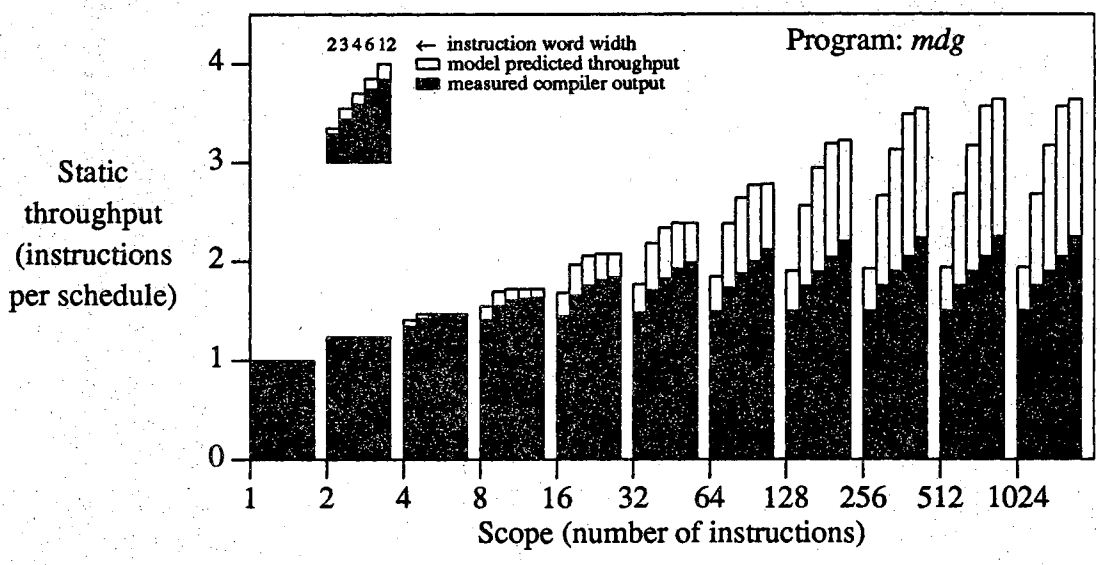


Figure 5.24 Throughput under resource and scope constraints for the *mdg* benchmark; resources varied with instruction word width equal to 2,3,4,6, and 12. The compiler output was influenced by resource constraints that are not part of the model.

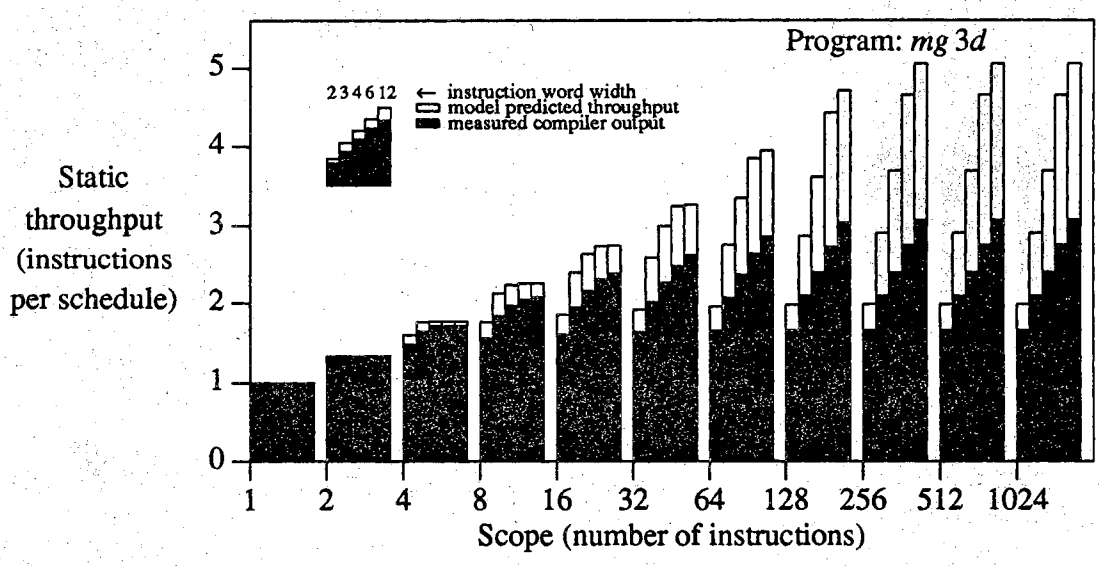


Figure 5.25 Throughput under resource and scope constraints for the *mg3d* benchmark; resources varied with instruction word width equal to 2,3,4,6, and 12. The compiler output was influenced by resource constraints that are not part of the model.

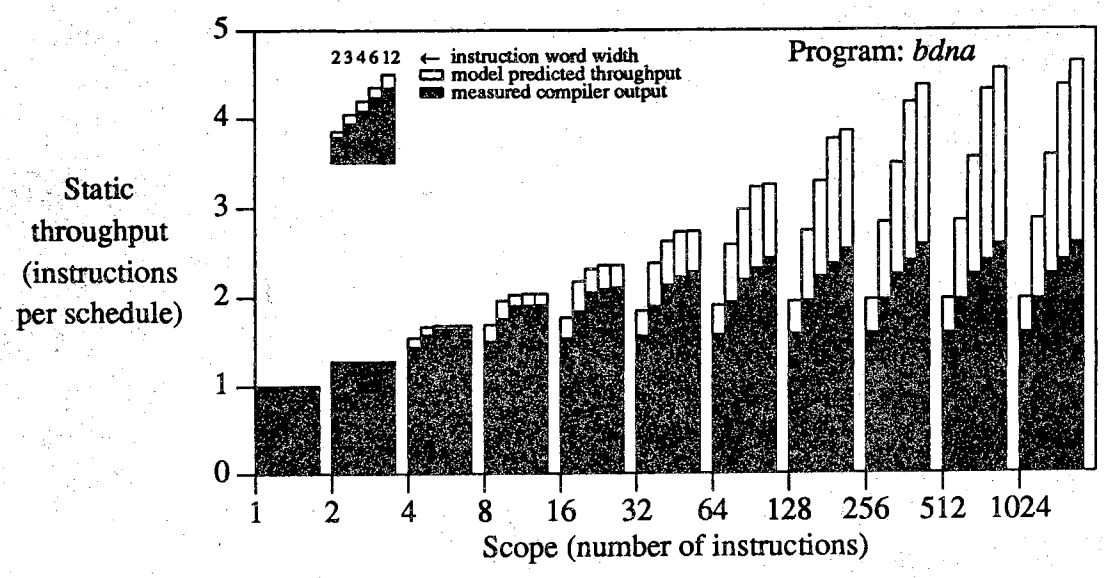


Figure 5.26 Throughput under resource and scope constraints for the *bdna* benchmark; resources varied with instruction word width equal to 2,3,4,6, and 12. The compiler output was influenced by resource constraints that are not part of the model.

First, a trace scheduling compiler must schedule subject to the available hardware resources in the target computer. The analytic model depends on experimentally gathered data for its p_δ probabilities. Because the target of the Multiflow compiler is a finite resource machine the p_δ probabilities inferred from the scheduled VLIW instruction stream omit some cases of instructions schedulable with instruction I_0. Thus, the measured p_δ values are a lower bound on the actual p_δ. We do not know of a parallelizing compiler that assumes an infinite resource environment and provides a detailed schedule, with mapping between the original sequential stream and the new compacted schedule.

Second, the collected set of p_δ statistics is a subset of the total statistics due to missing dependency information from the compiler. For example, when any compiler schedules I_8 with I_0, it can be concluded that independent instruction pairs were detected at a distance of 8, 7, 6, ..., and 1, because I_8 can be scheduled with I_0 only if I_8 is independent of not only I_0 but also of I_7, I_6, ..., and I_1. However, if I_8 is not scheduled with I_0, it is not possible to determine the responsible dependent instruction pair(s). Therefore, the collected statistics for scheduled instructions as a function of distance is a subset of the statistics corresponding to all possible schedulable instruction pairs as a function of their distance.

Finally, the analytic performance prediction model assumes a continuous instruction stream (the dynamic instruction stream) with the given p_δ characteristics, whereas the Multiflow compiler output produces several, potentially many, disjoint traces. It is quite reasonable for the compiler to do this. The analysis combines the p_δ values for all the traces, weighted by the estimated time spent executing each trace. This is an approximation of the dynamic p_δ values.

Fixes for the first and second issues will require dependency analysis tools specially designed for collecting p_δ values. A remedy for the third problem requires generating combined traces or collecting p_δ using dependency analysis on dynamic instruction streams. Analyzing dynamic instruction streams would also fix the problem of short traces mentioned in the previous section.

5.6 Summary

An analytic model for optimizing instruction window size has been presented. The value of this model is its ability to establish whether a performance bottleneck is (1) not having enough resources (number of pipelines), or (2) not having enough parallelism in the instruction stream, or (3) not examining enough instructions to extract the inherent parallelism. The proposed model has been validated by demonstrating that the predicted

throughput for the chosen set of benchmarks is close to the measured throughput from the compiler output. The cost of speculative execution, in terms of the delay required to undo the damage due to wrongfully executed instructions, has been quantified for the set of benchmarks.

The parameters p_δ and p_ω provide a means for characterizing inherent parallelism in an application instruction stream. Intuitively, p_δ corresponds to the inherent parallelism in the application program, and p_ω corresponds to the cost of extracting that parallelism. Although the performance potential of machine architectures can be compared in terms of parameters such as number of pipelines or processors, branch delay, cache miss delay, and so forth, the only common way for comparing two programs such as *spice* and *fpppp* has been in terms of their run time on a certain machine. The parameters p_δ and p_ω are a way of comparing the performance potential of programs in terms of a quantitative measure of their inherent parallelism. The combination of p_δ and p_ω provides quantitative insights (such as the cited difference between *fpppp* and *tair* in Section 5.4) into cost-performance tradeoffs associated with exploiting fine grain program parallelism. In the absence of this insight such tradeoffs have to be postponed to a much later stage during the design process and cost expensive simulation cycles.

One needs to be cautious in comparing the throughput plots of different benchmarks. A better comparison for performance potential of two benchmarks should be in terms of the speedup, i.e, the throughput ratio with respect to single-pipeline sequential execution, rather than in terms of individual throughputs. The baseline throughput might vary considerably with the benchmarks. For example, consider the *stanford* and *fpppp* benchmarks. The former consists of all integer arithmetic; the latter is a floating-point intensive benchmark and, hence, is very likely to have a baseline throughput of much less than one. Therefore, although *stanford* and *fpppp* may both have a throughput of around 1.6 for a window-size of 16, the latter implies a much higher speedup than the former.

Finally, the performance prediction approach presented in this chapter is meant primarily for actual applications (including "dusty decks") as opposed to kernels or small benchmarks. Full applications yield longer traces. The longer the traces, the more credible and meaningful the probability calculations, and hence the better the performance prediction. Predictions about the speed of real applications rather than those of kernels is an advantage of this approach.

CHAPTER 6
SPECTRUM OF CHOICES: SUPERPIPELINED, SUPERSCALAR OR MULTIPROCESSOR?

6.1 Introduction

This final chapter extends the model developed in Chapter 4 to include multiprocessors. The utilization of all three system types, as affected by the inherent parallelism in an instruction stream, is examined.

Recent simulation-based studies suggest that while superpipelines and superscalars are equally capable of exploiting fine-grain concurrency, multiprocessors are better at exploiting coarse-grain parallelism. Lilja and Yew [LiY90] used trace-driven simulations and concluded that the best performance is obtained using a coarse-grain multiprocessor configuration where each individual processor has a parallelism of two to four.

6.2 Delays Associated with Multiprocessors

Assume a system of N processors, each processor having k pipelines, where the pipelines are s stages deep (see Figure 6.1). Consider performing γ iterations of a program loop, each consisting of n/γ identical operations, each operation taking g gate levels of delay. These operations are being pipelined in response to the instructions being scheduled on each pipelined, which in turn is a consequence of source code level iterations being assigned to each processor.

For n a multiple of $s * k$ and γ a multiple of N, the first set of k results in each iteration completes after s clocks and then the remaining $n-k$ results finish in $(n-k)/k$ clocks in groups of k. Thus a total of,

$$\left\lceil s + \frac{\frac{n}{\gamma} - k}{k} \right\rceil \frac{\gamma}{N} \text{ clocks}$$

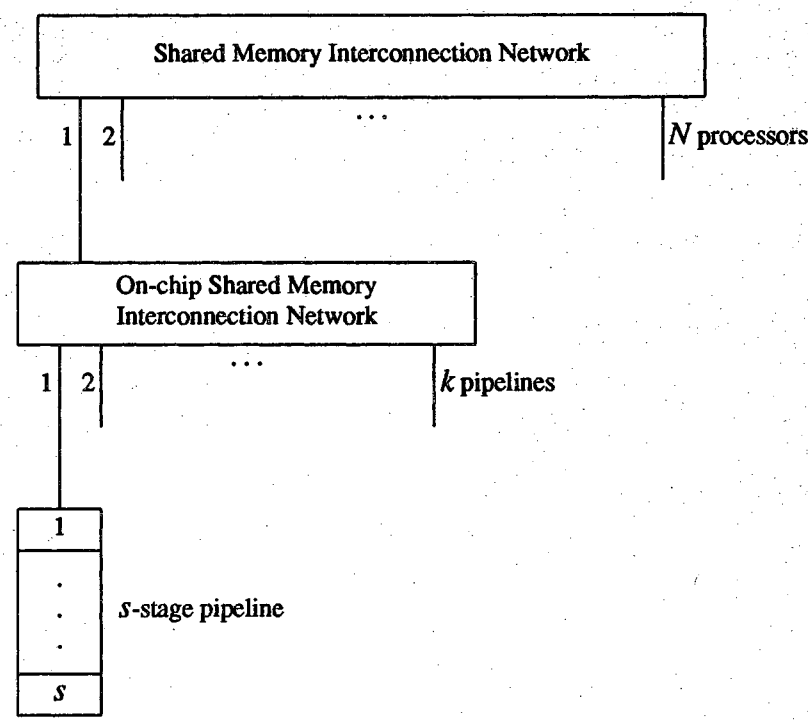


Figure 6.1 Combined system architecture assumed by the models.

are needed. Assuming the clock period is determined by the pipeline hardware, the above expression in terms of clocks can be re-written in terms of gate delays as,

$$\left\lceil \left[s + \frac{\frac{n}{\gamma} - k}{k} \right] \frac{\gamma}{N} \right\rceil * \left[\frac{g + 2(s-1)}{s} \right] \text{ gate-delays} \qquad \{6.1\}$$

The modelling details of dependency delay and operand fetch delay are discussed next. These models are similar to the one used by Cytron [Cyt86] to explain *Doacross* and to that used by Lilja and Yew [LiY90].

6.2.1 Dependency Delay

Suppose iterations are statically scheduled such that processor 1 executes loop iterations 1, 1+N, 1+2N, ...; processor 2 executes iteration 2, 2+N, 2+2N, ...; and so on. Further assume that the parallel iterations have a lexically backwards dependence of distance one, i.e., a certain statement S_i in iteration ω must be executed after a statement S_j ($i<j$) of the previous iteration, ω–1. Let $j-i = l\,n/\gamma$, where l represents the fraction of loop code exhibiting the dependence. Ignoring any delay (due to, for example, branching or cache misses) this fraction of code (from statement S_i to S_j) can be executed in $l\,n/(\gamma k)$ clocks on k pipelines. As shown in Figure 6.2, this implies that a new iteration on a processor is delayed by $(N-1)\,l\,n/(\gamma k)$ clocks. However, this waiting time is also overlapped with the execution of remaining code of the current iteration. The average length of this remaining code is given by $(1-l)\,n/(2\gamma)$, which can be executed in $(1-l)\,n/(2\gamma k)$ clocks. Therefore,

$$\text{dependency delay} = \max\left[0\,,\,(N-1)\frac{l\,n}{\gamma k} - \frac{(1-l)\,n}{2\gamma k}\right] \text{ clocks.}$$

For an inter-iteration dependency distance of δ iterations, the above equation becomes

$$\text{dependency delay} = \max\left[0\,,\,\left\lceil\frac{N-\delta}{\delta}\right\rceil\frac{l\,n}{\gamma k} - \frac{(1-l)\,n}{2\gamma k}\right] \text{ clocks.} \qquad \{6.2\}$$

```
DO i = 1,N
S1:
S2:
S3: B(i) = A(i) + C(i-2)
S4:
S5: C(i) = D(i) * E(i)
S6:
S7:
S8:
S9:
S10:
ENDDO
```

$l = 3/10 = 0.3, \ \delta = 2$

Processor 1	Processor 2	Processor 3	Processor 4
...	...	...	...
S3(1)	S3(2)	...	...
...	...	...	...
S5(1)	S5(2)	delay	delay
...	...	S3(3)	S3(4)
S10(1)	S10(2)	...	...
delay	delay	S5(3)	S5(4)
S3(5)	S3(6)	...	...
...	...	S10(3)	S10(4)

Note: Si(j) denotes j^{th} iteration of the instruction *Si*

Figure 6.2 Inter-iteration dependency

6.2.2 Operand Fetch Delay

Assume a multistage interconnection network between the multi-ported, on-chip cache and the pipelines, and another such network between the processor and the off-chip, shared, global memory. Then operand fetch delay can be modelled as

$$\text{operand fetch delay} = b_0 + b_1 \log_2 k + f(log_2 k, \text{utilization}), \quad \text{for } N = 1 \qquad \{6.3\}$$

$$\text{operand fetch delay} = (1-y)\left[b_0 + b_1 \log_2 k + f(log_2 k, \text{utilization})\right] + y\left[c_0 + c_1 \log_2 N + f'(log_2 N, \text{utilization})\right], \quad \text{for } N > 1. \qquad \{6.4\}$$

Here, y is the probability of accessing the shared global memory. For more than one processor, assume that all the references to shared variables go to the off-chip shared memory. Further, b_0 is the delay encountered for accessing the on-chip cache, c_0 is the delay required for accessing the off-chip shared global memory, and $b_1 \log_2 k$ and $c_1 \log_2 k$ are network access delay for the cache and the global memory, respectively. Functions f and f' represent delays due to interconnection network contention.

Unlike the superpipeline/superscalar model, the simplifications of ignoring on-chip data cache misses and assuming that total cache access time is not constant are made. Let on-chip cache access time, $b_0 = g\, d_c$ and off-chip access time, $c_0 = g\, d_m$. Based on the reported experience with the Cedar system [LiY90], contention delays are assumed to be 50 percent of the network delays, i.e., $(b_1/2)\log_2 k$ and $(c_1/2)\log_2 N$ respectively for the cache and the shared memory. Let the network delay factor for on-chip implementation, $b_1 = g\, d_n^c$ and that for the off-chip memory be, $c_1 = g\, d_n^m$.

Branch delays and operand fetch delays are further reduced by *overlap* factors of π_b and π_o (both less than one), respectively. These factors are determined by how often compiler is able to hide these delays behind execution delays for machines with multi-cycle operations.

6.3 Utilization Constraints

The preceding discussion assumes that there are k instructions available to be scheduled on the k pipelines and N iterations available to be assigned onto the N processors. This is not always true due to data dependency constraints. Similarly, pipeline interlocks can cause freezes of pipeline segments. Each of these effects result in a drop in the utilization of the resources. This decrease can be modelled as a scalar factor reducing utilization or as a modification of the utilization distribution, via a vector

product.

6.3.1 Characteristics of Utilization Curves

The performance decrease due to utilization constraints is simply a manifestation of the scheduling delay of Section 4.2. Sometimes this delay is best modelled using an additional delay term, such as when modelling fetch delay due to branches or when modelling the dependency delay for scheduling different iterations on a multiprocessor. At times when the available information is less precise or less regular, the delay is best modelled as a utilization factor, u. This is distinct from the overall resource utilization introduced in Section 4.2. Let u be the utilization factor, then

$$u = \frac{s_{av}}{s} * \frac{k_{av}}{k} * \frac{N_{av}}{N},$$

where s_{av} refers to the average number of active pipeline segments, k_{av} to the average number of active pipelines for superscalar processors, and N_{av} to the average number of active processors for multiprocessor computers.

The utilization factor, u, should have the following characteristics:

a) Let α be the dynamic fraction of code that must be executed in strict order on a single pipeline, and similarly, let β be the dynamic fraction of code that must be executed serially on a single processor. Two code sequences with the same α (or β) can have different amounts of inherent parallelism. Although α and β place an upper limit on the utilization, they are not a measure of the actual amount of parallelism. For example, the two code sequences shown in Figure 6.3 have the same α of 0.1, but while code sequence (b) would be at its peak performance with 10 processors, code sequence (a) would require 28 processors for peak performance.

Lilja and Yew [LiY90] report different speedups for programs in different α-categories and the same range of speedup for programs in the same α-category. We believe that for the chosen set of programs, different α-categories appear to correspond to different utilization categories and the reported correlation of actual speedups with α-categories is coincidental.

For N processors, u can be expressed as:

$$u = \frac{\beta}{N} + (1-\beta)\, u_{ns}$$

Next look at the characteristics of u_{ns}, which refers to the utilization in the parallelizable or non-scalar portion of the code.

```
******************

S1
S2
S3
S4

for i = 1 to 28
S1
S2:

******************

for i = 1 to 10
S1:
S2:

for i = 1 to 10
S1:
S2:

for i = 1 to 10
S1:
S2:

******************
```

Figure 6.3 Sample code sequences with same α (= 0.1) but different amounts of parallelism

b) For a small number of processors (or pipelines or pipeline segments), $N_{av} \approx N$ (or $k_{av} \approx k$, or $s_{av} \approx s$), that is, the utilization decrease with an additional processor (or pipeline or pipeline segment) is very small. But as the number of processor grows, utilization decreases more significantly with additional processors. For a large number of processors, additional processors do little to increase the average number of active processors. The above holds true analogously for large numbers of pipelines and pipeline segments. Hence as N increases, utilization tends to the curve given by the function $1/N$.

c) Another important characteristic of any utilization curve can be stated as:
If u_{ns} $(for\ N=x) = y$ then u_{ns} $(for\ N=z>x) \geq (y\ x)/N$.
Stated simply, the number of active processors cannot decrease with the addition of a new processor. Such a restriction makes intuitive sense in case of superscalars and multiprocessors. The utilization equation for pipelines in Chapter 2 does not impose this restriction, because, additional segmentation is not as straightforward as the addition of another pipeline or processor, and conceivably, the average number of active segments can decrease with increasing segmentation of a pipeline. This distinction is ignored in this chapter,

d) Finally, in case of superpipelines and multiprocessors, $k_{av} \leq k_{\max}$ and $N_{av} \leq N_{\max}$, respectively, which are determined by the maximum degree of fine-grain (operation-level) parallelism and coarse-grain (iteration-level) parallelism, respectively.

6.3.2 Alternate Characterization of Program Parallelism

In Chapter 5, inherent parallelism in program was characterized using the p_δ and p_ω statistics. The parameters introduced above can provide an alternate characterization of the inherent parallelism in application programs. While α and β determine the portion of code that lacks any parallelism, $k_{\max}$ and $N_{\max}$ limit the maximum parallelism that can be extracted in any instance. These parameters together put an upper limit on the utilization of superscalars and multiprocessors. For example, in the case of multiprocessors, the upper limit is

$$u_{\max} = \frac{\beta}{N} + (1-\beta)\ \min\left[1, \frac{N_{\max}}{N}\right] \qquad \{6.5\}$$

Let u_{ns}^{sp}, u_{ns}^{ss} and u_{ns}^{mp} = $\{low, average, high\}$ depending on whether the parallelism available for the superpipeline, superscalar, or the multiprocessor respectively is *low*, *average*, or *high*. This implies the corresponding utilization curve from Figure 6.4. The bounding curves of Figure 6.4 are the lower and upper bounds of utilization as given by $1/N$ (in case of multiprocessors) and 1, respectively. Moving from applications with a

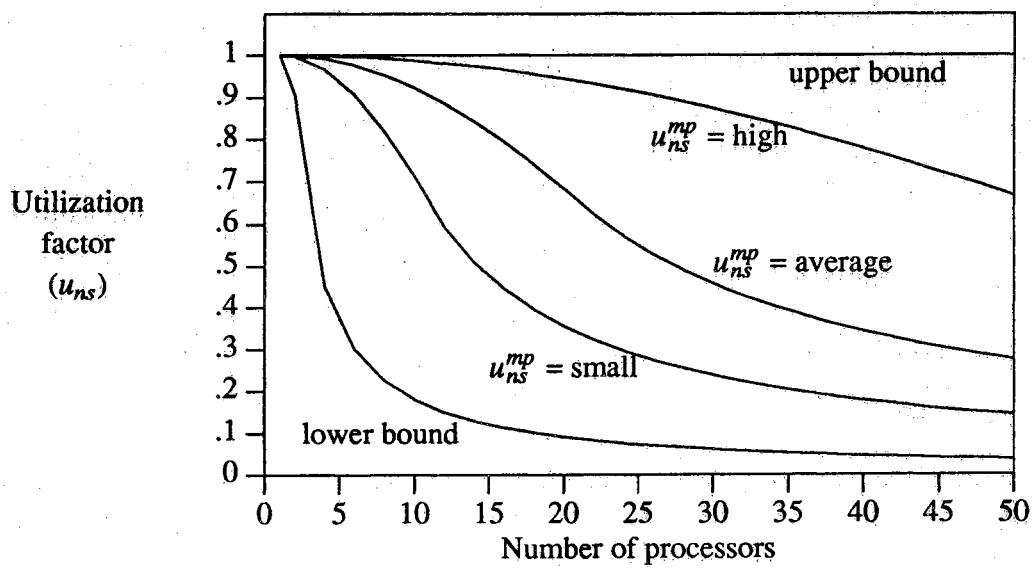
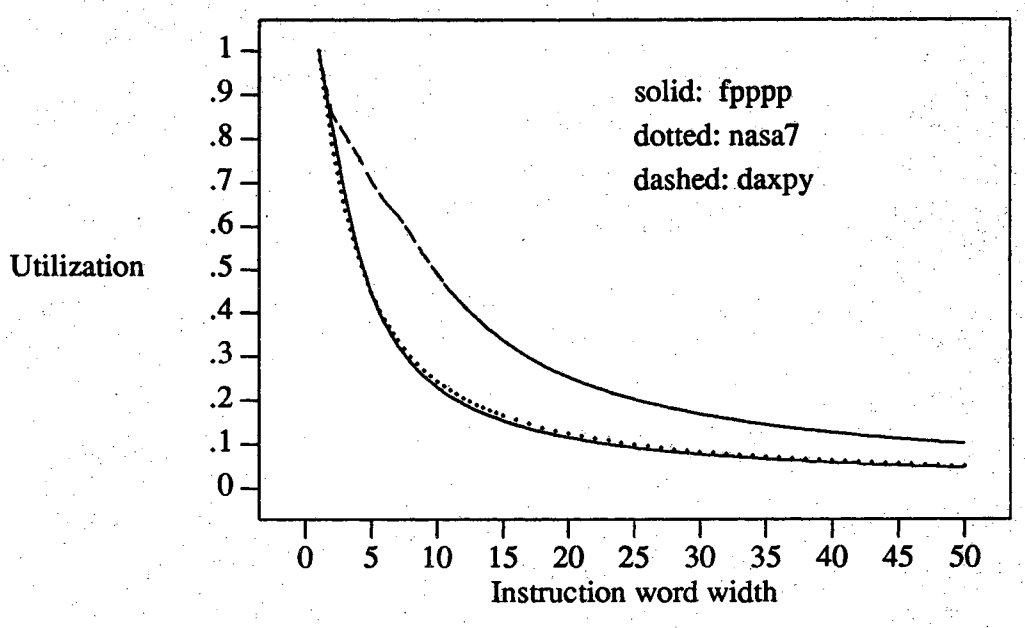
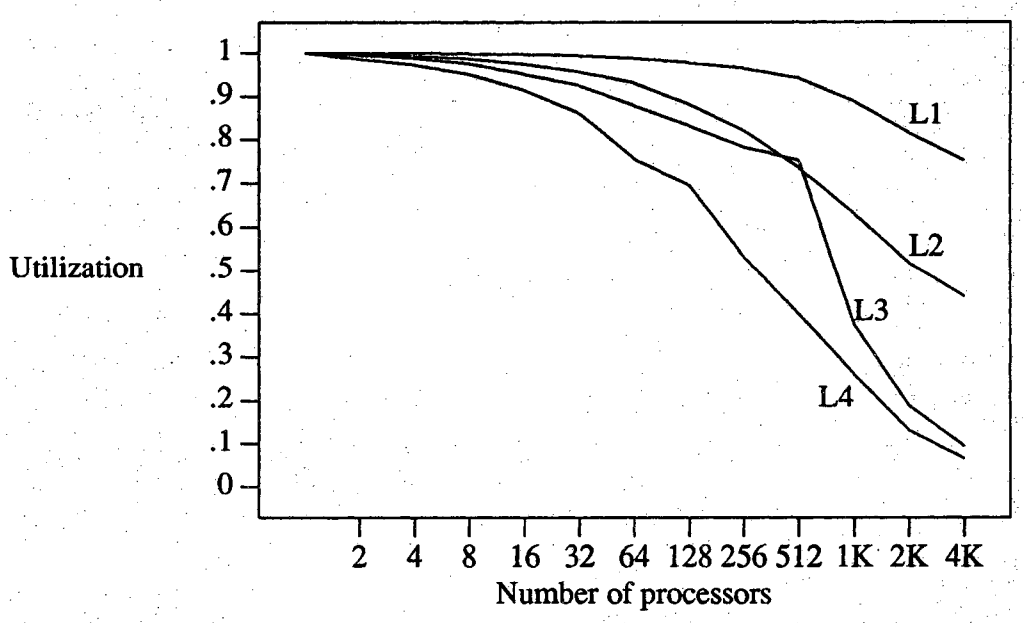


Figure 6.4 Assumed utilization curves.



(a)



(b)

Figure 6.5 (a) Utilization versus instruction word width measured on the Multiflow TRACE 28/200 computer, and (b) utilization curves derived from tables on pp. 214-217 of [Pol86].

large amount of inherent parallelism to those with little or no parallelism, parameters u_{ns}^{sp}, u_{ns}^{ss}, and u_{ns}^{mp} decrease. The level at which the parallelism is available is important. Loop level parallelism is reflected by the value of u_{ns}^{mp}. The parameter u_{ns}^{ss} indicates instruction-level parallelism. A program that is strictly serial would force u_{ns}^{mp} and u_{ns}^{ss} to their lower bounds, but u_{ns}^{sp} may still be very high. Inter-iteration dependencies that limit the utilization of multiprocessors are characterized by l, the fraction of code in a loop body that exhibits dependence and δ, the iteration distance of the dependence.

In order to get a realistic idea of the nature of the utilization curves, data was collected from machines relying on both fine-grain parallelism and iteration-level, coarse-grain parallelism. Figure 6.5 (a) represents the utilization data for the Multiflow TRACE 28/200 machine. Figure 6.5 (b) represents utilization inferred from the speedup results published by Polychronopoulos using guided self-scheduling techniques on certain loops [Pol86]. The nature of these empirical curves conforms with characteristics (b) and (c) above. Based on this combination of experimental and analytical insights, a family of utilization curves have been used that are considered representative of *low*, *average*, and *high* amounts of parallelism in the non-scalar portion of the code, as depicted in Figure 6.4. Let u_{ns}^{sp}, u_{ns}^{ss} and u_{ns}^{mp} = {*low*, *average*, *high*} depending on whether the parallelism available for the superpipeline, superscalar, or the multiprocessor respectively is *low*, *average*, or *high*. For the discussion to follow the sole purpose of these curves is to assess the performance impact of a change in the available amount of parallelism as the program transformation moves from coarse-grain to fine-grain, or as different applications are executed.

6.4 Results

A nominal set of values (see Table 6.1) are assumed to describe the hardware performance characteristics and program characteristics for a hypothetical, but realistic environment. The nominal value of 0.4 for d_m implies a main memory access time of about two to three clocks for a five to six stages deep pipeline, which is typical of current microprocessors. The on-chip cache is assumed to be four times times faster its off-chip counterpart. On-chip network delay factor, d_n^c is chosen such that the network delay is at most twice the access time to the cache. The off-chip network is assumed about two times slower than its on-chip counterpart. Also, 20 percent of memory accesses are assumed to be to shared variables. This fraction may be much higher on some systems due to the main memory traffic to maintain cache consistency, in which case this fraction would be a function of the number of processors and the particular consistency algorithm in use. Finally, 30 percent of the branch and operand fetch delays are assumed to be

Table 6.1 Nominal values of model parameters describing hardware and program characteristics.

Hardware Characteristics:	
Fraction of operation gate delay required for branch resolution, d_b	0.15
Fraction of operation gate delay required for on-chip cache access, d_c	0.1
Fraction of operation gate delay required for off-chip memory access, d_m	0.4
On-chip network access delay factor, d_n^c	1/45
Off-chip network access delay factor, d_n^m	1/25
Program Characteristics:	
Fraction of code that must be serially executed on one pipeline, α	0.1
Fraction of code that must be serially executed on one processor, β	0.1
Branch instruction probability, b	0.1
Probability of memory-reference per operation, w	0.2
Fraction of data accesses to shared variables, y	0.2
Maximum degree of operation (pipeline) level parallelism, k_{max}	50
Maximum degree of iteration (processor) level parallelism, N_{max}	50
Across iteration dependency distance, δ	1
Distance between dependent instructions as a fraction of the size of loop body, l	0.0
Utilization factor for the parallelizable code for pipeline stages, u_{ns}^{sp}	follows
Utilization factor for the parallelizable code for complete pipelines, u_{ns}^{ss}	the average
Utilization factor for the parallelizable code for processors, u_{ns}^{mp}	utilization curve of Figure 6.4
Fraction of branch delays overlapped with execution delays, π_b	0.3
Fraction of operand-fetch delays overlapped with execution delays, π_o	0.3

overlapped with execution delays. This is consistent with the reported figures from compilers for typical RISC machines.

Figure 6.6 demonstrates the impact of utilization on the throughput of superpipelined systems. Recall that because delay is measured in units of gate delays, operand fetch delay is a constant overhead in the absence of any growing network delay. The only growing overhead is that due to the inter-segment buffers. In the analysis range, this is noticeable only for poorly utilized pipelines, which show a very small drop in throughput with an increasing number of segments. For better utilized pipelines the throughput keeps growing, although at a slower rate as observed in the analysis of superpipelines in Chapter 4.

Turning attention to superscalars, there are two major differences with respect to the superpipelined systems. First, the addition of the interconnection network for shared memory access results in a growing operand fetch overhead. Second, utilization for superscalars has an additional factor, α, the fraction of code that must be executed on a single pipeline. Figures 6.7 (a) and (b) plot the maximum throughput attained and the corresponding number of pipelines for various combinations of α and the utilization factor in terms of u_{ns}^{ss}. The effect of α becomes noticeable only for larger values, say $\alpha > 0.1$. Also, for the same value of α, different levels of throughput can be achieved depending on the utilization factor. The optimum number of pipelines shows even more insensitivity towards α, except that for $\alpha > 0.1$, there may be slight increase in the number of pipelines required to achieve the optimum throughput. The optimum throughput, as expected, grows with better utilization factor; so does the number of pipelines required to achieve this optimum. This is in accord with the findings of Lilja and Yew [LiY90] as visible in their category-2 performance plots, where better utilized multiprocessors require a higher degree of parallelism (number of pipelines or processors) than the lesser utilized superscalars to achieve a higher level of speedup.

The impact of memory delay is shown in Figures 6.8 (a) and (b). The two major components are the memory access time d_c and the network access delay factor d_n^c. The number of pipelines where maximum throughput is attained becomes increasingly more dependent on the ratio, d_c/d_n^c, as d_c increases. Since d_n^c controls the rate of growth of operand fetch delay, its impact on how long it takes before the operand fetch delay overruns the advantage of an additional pipeline is to be expected. The stepwise nature of the curves in Figure 6.8 (b) (which results from the log terms in Equation 6.3) is difficult to follow. Hence, the corresponding data is also presented in a tabular form in Table 6.2.

Ignoring dependency overhead, Figures 6.7 and 6.8 also represent multiprocessor performance, except that the optimum throughputs would be somewhat less due to the slower off-chip memory interface. Such graphs can be useful in deciding the incremental benefit of adding a processor (or pipeline). Suppose the curves in Figures 6.7 and 6.8 were used for estimating multiprocessor performance (read u_{ns}^{mp} in place of u_{ns}^{ss}, and *processors* in place of *pipelines*). If a program environment offers a higher level of

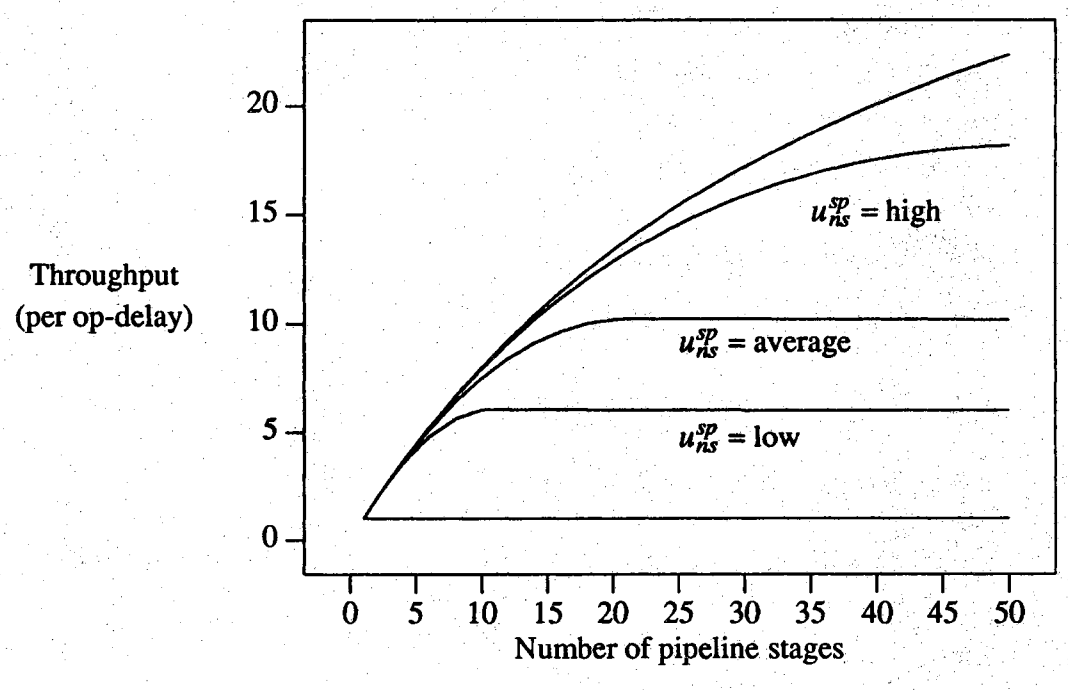
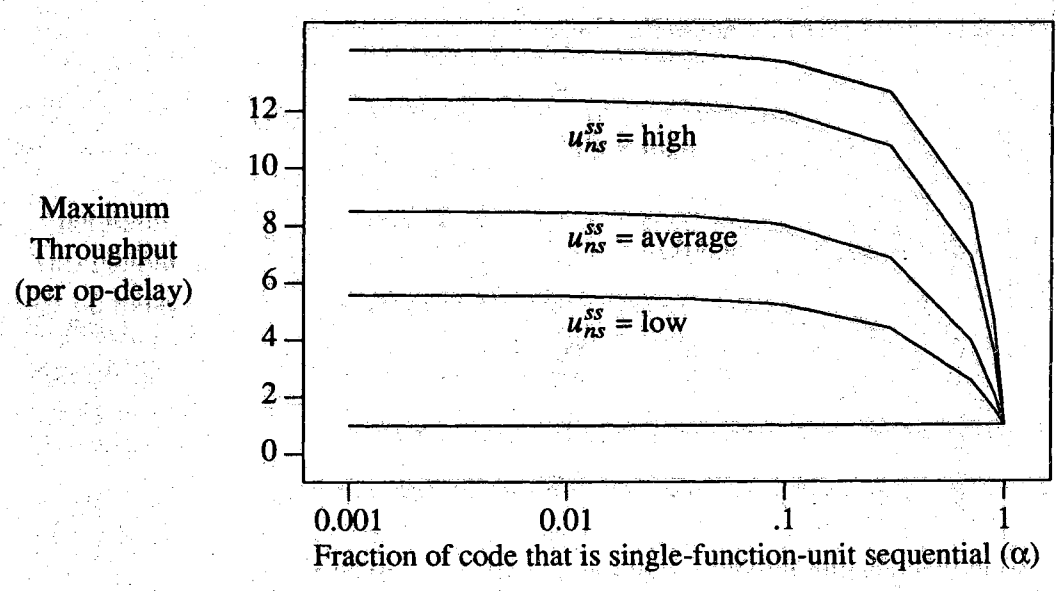
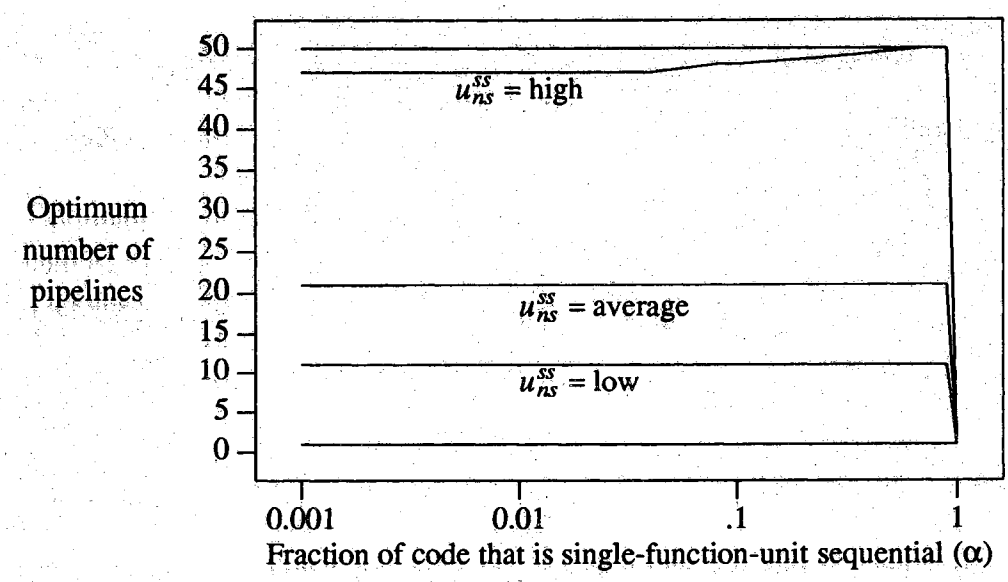


Figure 6.6 Impact of utilization on throughput for superpipelines.

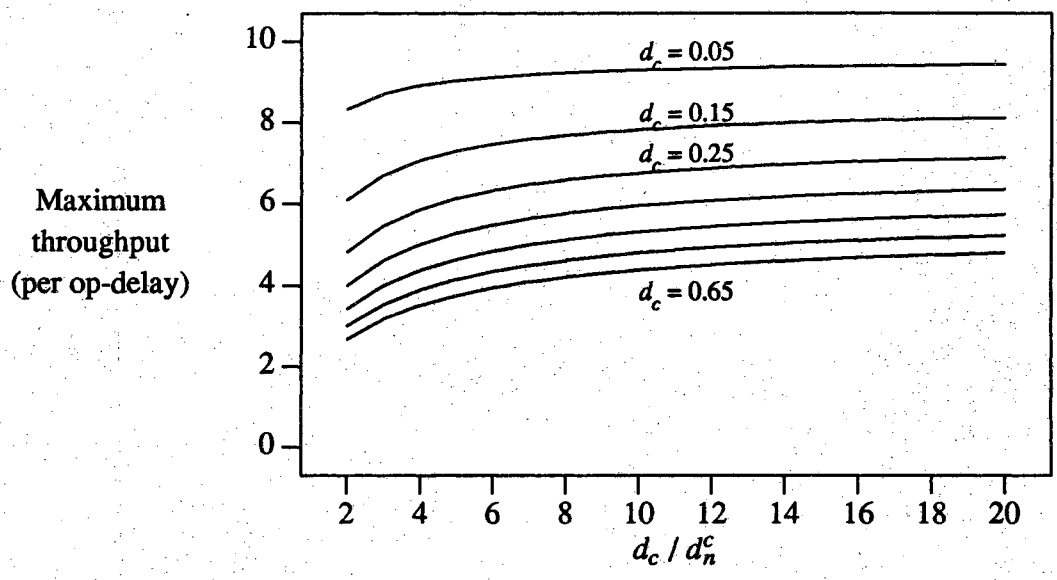


(a)

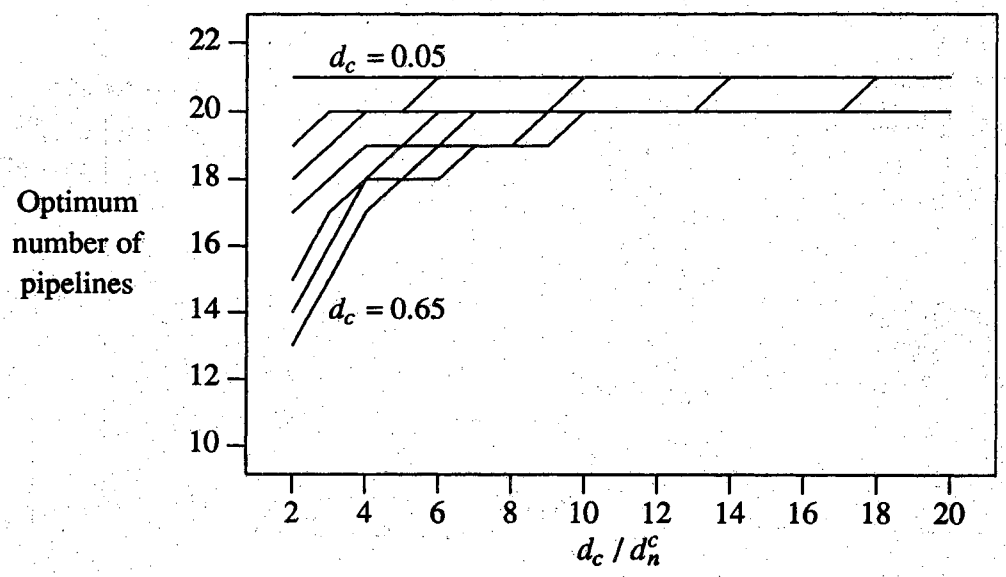


(b)

Figure 6.7 Maximum throughput (a) and optimum number of pipelines (b) as a function of the fraction of code that must be executed on a single function unit (pipeline).



(a)



(b)

Figure 6.8 Maximum throughput (a) and optimum number of pipelines (b) versus ratio of memory access delay (d_c) to network access delay factor (d_n^c); d_c values shown are 0.05 to 0.65 in increments of 0.1.

Table 6.2 Optimum number of pipelines versus ratio of memory access delay, d_c, to network access delay factor, d_n^c.

$\frac{d_c}{d_n^c}$	Optimum number of pipelines						
	d_c=0.05	d_c=0.15	d_c=0.25	d_c=0.35	d_c=0.45	d_c=0.55	d_c=0.65
2	21	19	18	17	15	14	13
3	21	20	19	18	17	16	15
4	21	20	20	19	18	18	17
5	21	20	20	19	19	18	18
6	21	21	20	20	19	19	18
7	21	21	20	20	20	19	19
8	21	21	20	20	20	19	19
9	21	21	20	20	20	20	19
10	21	21	21	20	20	20	20
11	21	21	21	20	20	20	20
12	21	21	21	20	20	20	20
13	21	21	21	20	20	20	20
14	21	21	21	21	20	20	20
15	21	21	21	21	20	20	20
16	21	21	21	21	20	20	20
17	21	21	21	21	20	20	20
18	21	21	21	21	21	20	20
19	21	21	21	21	21	20	20
20	21	21	21	21	21	20	20

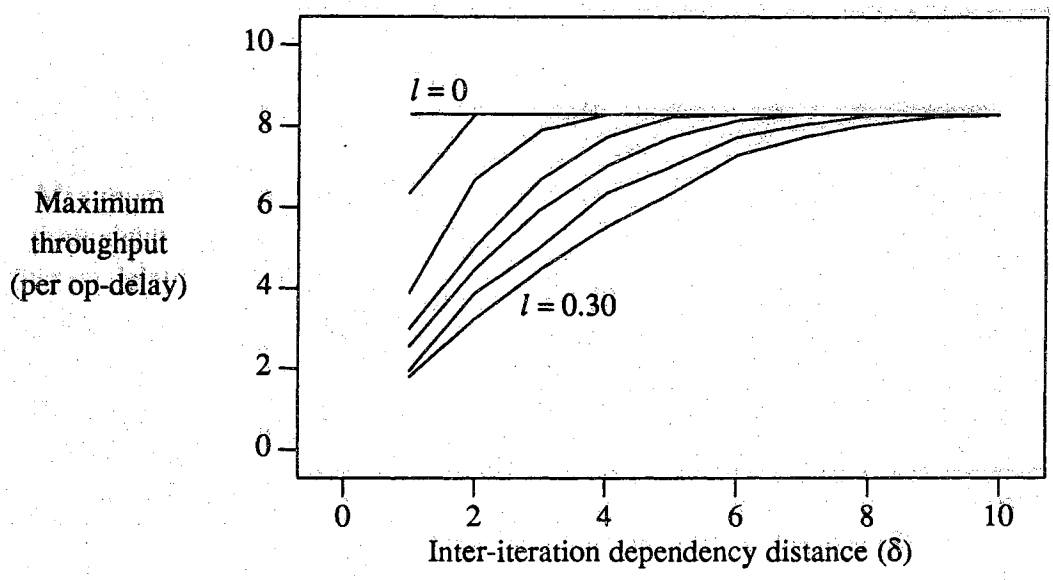
coarse-grain parallelism such that u_{ns}^{mp} increases from *average* to *high*, this implies an approximate increase of about 45 percent in optimum throughput but an increase of about 125 percent in the number of processors required to attain that throughput.

Dependency overhead has two important variables: δ, the distance in number of iterations between dependent instructions, and l, the distance between the dependent instructions in the same iteration as a fraction of the loop body length. Before new iteration on a processor executes, all the previous iterations on which it is dependent must have completed up to the point of dependency (Figure 6.2). An increase in δ has an effect in two ways. First, the number of prior iterations that must complete to the point of the dependency is reduced. For example, on a 10 processor system if $\delta = 1$, a new iteration has to wait for the nine previous iterations; whereas, if $\delta = 2$, there are only four iterations to wait for. Second, an increase in δ retards the onset of dependency delay, as there is no dependency delay for less than δ processors. The distance between dependent instructions, l, also has a two-fold impact. As l grows, the larger dependency region yields a longer wait for initiation of a new iteration. A larger l also implies there is on average a lesser remaining portion of loop body to hide the dependency delay. Figures 6.9 (a) and (b) plot the optimum performance and the corresponding number of processors for varying combination of l and δ. As a function of δ, notice the nonlinear nature of optimum throughput curves in Figure 6.9 (a), whereas, the optimum number of processors changes almost linearly (Figure 6.9 (b)).

6.5 Combined Systems

Finally, consider the performance issues of combined systems, such as superpipelined multiprocessors and superscalar multiprocessors, which are obtained by using clusters of superpipelined and superscalar processors, respectively. Assume a single processor system with a pipeline that is 32 stages deep. (This implies an issuing capacity of 32 instructions during the length of the pipeline.) Assume that the application stream has a significant amount of coarse-grain parallelism. Therefore, trading off some pipeline stages for additional processors is expected to improve the performance.

Consider Figure 6.10 and assume $u_{ns}^{sp} = low$ and $u_{ns}^{mp} = high$. This means significantly more coarse-grain parallelism (resulting in a better utilized multiprocessor configuration) than the amount of fine-grain parallelism (causing a poorly utilized pipeline configuration). Keeping a constant issuing capacity of 32, add more processors. Initially, performance improves significantly. As more processors yet are added, multiprocessor performance starts to level off and begins to approach the lower limits of



(a)

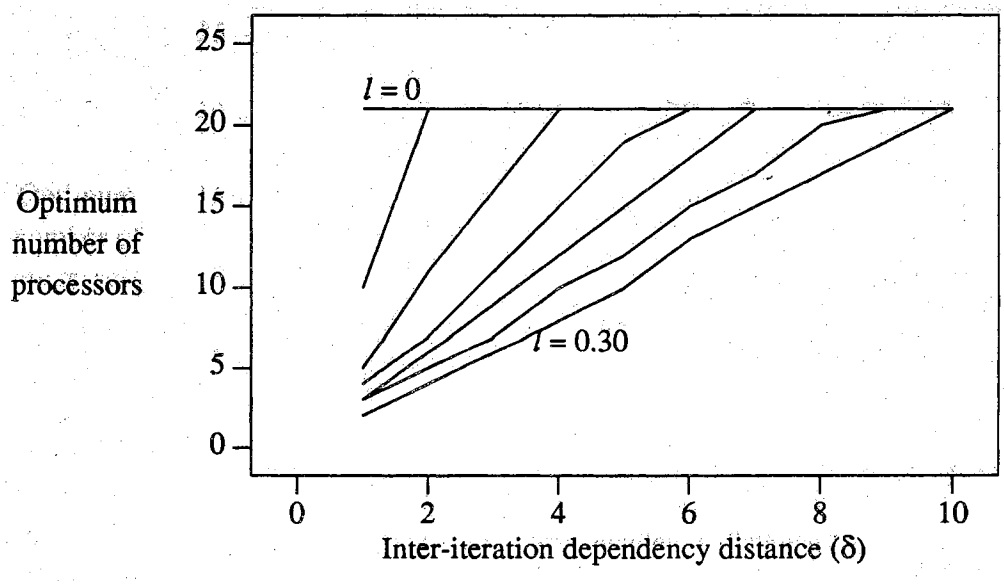


Figure 6.9 Maximum throughput (a) and optimum number of processors (b) versus inter-iteration dependency distance; l ranges from 0 to 0.30 in increments of 0.05.

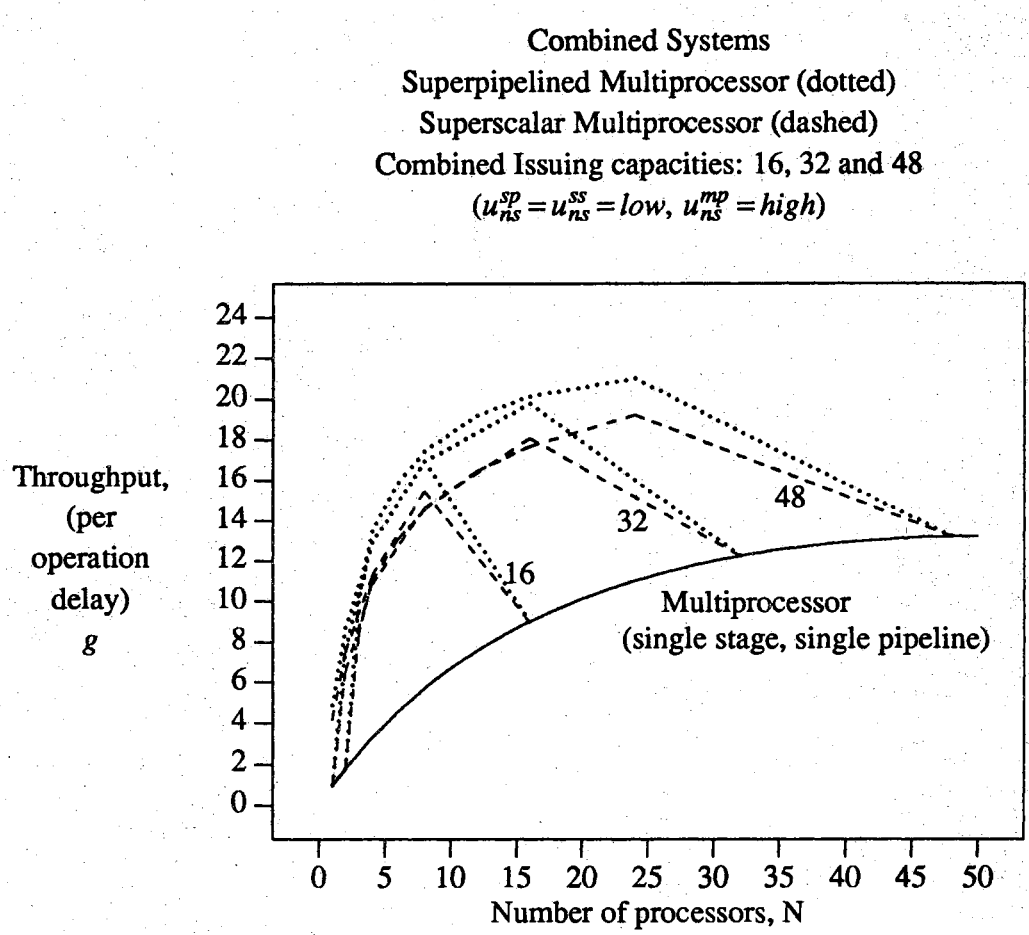


Figure 6.10 Throughput plot of combined system performance.

superpipeline performance curves, where the performance loss due to reduced number of stages becomes increasingly significant. This results in an optimum at 2-stage pipelines and 16 processors. Performance plots for issuing capacities of 16 and 48 are also shown. This is repeated with superscalar multiprocessor systems. It is intersting that in all these cases the optimum performance is obtained when the combined system is a multiprocessor with processors using 2- or 3-stage pipelines. This observation agrees well with the trace-driven simulation-based findings of Lilja and Yew [LiY90]. Also, as the difference between the utilization factors (u_{ns}^{sp} and u_{ns}^{ss} or u_{ns}^{mp}) shrinks, the optimum shifts more in favor of pipelines as observed by Lilja and Yew [LiY90].

6.6 Summary

The analytical models developed in previous chapters and extended here allow easy, comparative evaluation of superpipelines, superscalars, and multiprocessors.

The extended model although simplistic in nature is shown capable of deriving some useful results. It is shown that maximum throughput is not sensitive to the ratio of memory access time to network access delay. However, the number of pipelines (or processors) at which the maximum throughput is obtained is increasingly sensitive to this ratio as the memory access time increases. As a function of inter-iteration dependency distance, optimum throughput varies nonlinearly, whereas the corresponding optimum number of processors does vary linearly. Finally, for programs with more coarse-grain parallelism, optimum performance is obtained in the multiprocessor configuration where each processor has hardware supporting fine-grain parallelism of degree two to four.

CHAPTER 7
CONCLUSIONS

This research has presented analytical approaches to optimal processor design. The model developed during this research is primarily targeted to, although not limited to, superscalar processors with dynamic scheduling. Starting with single pipeline optimization, the model was gradually refined to gain insights into various performance tradeoffs associated with multiple-pipeline systems. Special attention was paid to the understanding of delays associated with different branch strategies, misprediction delay during beyond-basic-block execution, and the loss of throughput due to inherent dependencies in the source code.

Throughout the dissertation, the model development and/or enhancement consists of three generic steps. First, a model is proposed based on known and/or expected performance characteristics of the system. Second, the proposed model is validated by correlating its predictions with published results and/or experimentally gathered performance measurements. Third, the validated model is used to gain new insights into performance limiting factors.

7.1 Summary

First, a survey of the existing machines and literature was presented with a proposed classification of various approaches for exploiting fine-grain concurrency. Optimization of a single pipeline is discussed based on an analytical model. The predicted nature of performance curves is found to be in close proximity with published results using simulation techniques. A model is also developed for comparing different branch strategies for single-pipeline processors, in terms of their effectiveness in reducing the branch delay. Additional instruction traffic generated by the different branch strategies is also studied and is shown to be a useful criterion for choosing between equally well performing strategies.

Such analytical techniques are extended to processors with multiple pipelines to study the tradeoffs associated with deeper versus multiple pipelines. An analytical model is developed for optimizing the size of an instruction window for machines with dynamic

scheduling. The cost associated with beyond-basic-block execution is examined via probability distributions that characterize the inherent parallelism in the instruction stream. The throughput prediction of the analytic model under resource and scope constraints is shown to be close to the measured static throughput of the compiler output for 24 benchmarks chosen from the SPEC, NAS, and Perfect benchmark suites. Further experiments provide misprediction delay estimates for these benchmarks under scope constraints, assuming beyond-basic-block, out-of-order execution and run-time scheduling. These results were derived using traces from the Multiflow TRACE SCHEDULING™ compacting C and FORTRAN 77 compilers.

A simplified extension to the model to include multiprocessors is also proposed. The extended model is used to analyze combined systems, such as superpipelined multiprocessors and superscalar multiprocessors. It is shown that the number of pipelines (or processors) at which the maximum throughput is obtained is increasingly sensitive to the ratio of memory access time to network access delay, as memory access time increases. Further, as a function of inter-iteration dependency distance, optimum throughput is shown to vary nonlinearly, whereas the corresponding optimum number of processors varies linearly. The predictions from the analytical model agree with similar results published using simulation-based techniques.

7.2 Contributions

The contributions of this research can be summarized as follows:

a) Simulation-based performance predictions for single-pipeline optimizations and those for combined system optimizations, have been analytically correlated. Relative to the previous simulation-based studies, this analytical approach is less time consuming, more flexible, and offers additional insights into the performance issues.

b) A comparative analysis of different branch strategies has been presented on a common analytical platform. Also, the additional instruction traffic associated with the branch strategies has been analyzed on a comparative basis. This aspect of branch strategies has not been reported in published literature to this date.

c) A validated model has been presented for optimizing the size of an instruction window for superscalar processors with beyond-basic-block, dynamic scheduling. Window sizes as large as 1024 instructions or more can be analyzed quickly. The published material on this tradeoff [AKT86, SJH89, and Joh91] has been solely based on simulation-derived findings. The model developed can also offer insights into where a performance bottleneck might be: insufficient resources to exploit

discovered parallelism, insufficient instruction stream parallelism, or insufficient scope of concurrency detection.

d) Although the performance potential of machine architectures can be compared in terms of parameters such as number of pipelines or processors, branch delay, cache miss delay, and so forth, the only common way for comparing programs has been in terms of their run time on a certain machine. This research proposes certain parameters, p_δ and p_ω, as a way of comparing the performance potential of programs in terms of a quantitative measure of their inherent parallelism. The combination of p_δ and p_ω provides quantitative insights into cost-performance tradeoffs associated with exploiting fine grain program parallelism.

e) Throughput estimates for a variety of benchmarks have been provided under under resource and scope constraints, assuming out-of-sequence, beyond-basic-block execution. For some of the benchmarks, data has been provided for a scope as large as 1024 instructions. Previous studies have either been limited to within basic blocks [AKT86] or limited to simulations up to a window size of 32 instructions [SJH89, Joh91]. Assuming dynamic scheduling, the research also provides misprediction delay estimates for the analyzed benchmarks up to a lookahead of 32 basic blocks.

Most of the published work on processor performance has been based on simulations, which are a valuable tool for providing accurate performance estimates for the simulated program traces. The research presented in this dissertation seeks to complement previous work by providing an approach based on relatively simple, validated analytical models. We hope the contributions of this research will be appealing enough for some processor architect to try out some of the models for some future processor design.

CHAPTER 8
FUTURE RESEARCH

This chapter offers extensions of some of the ideas presented in previous chapters for future research work in this area.

8.1 Out-of-sequence Execution Versus Locality of Operand References

A sequence of successive memory requests that are from logically related dependent operations exhibit locality of reference, both spatial and temporal. This locality is obscured when independent, logically unrelated operations are grouped together for simultaneous execution. This was ignored in Chapter 4, since the cache miss rate, $1-h$, was assumed unchanged as more and more operations from different pipelines were issued together. If the scope of concurrency detection is small, independent operations grouped together are likely to be of the same working set [Den70]. For example, during loop unrolling, if $A[i]$ is in cache then $A[i+1]$ is likely to be in cache also. Conversely, if the scope is large enough to group independent operations belonging to different working sets, it is unreasonable to expect a simultaneous cache hit for all of these references.

Consider for example, the two execution scenarios listed in Figure 8.1. The sequential case corresponds to purely sequential execution, whereas the parallel case allows out-of-sequence execution. For the sake of simplicity, the latter differs from the former in only that it permits simultaneous fetch of two unrelated operands A and D. In the parallel case, Fetch D can result in a cache miss and displace the line containing C. Subsequently, a miss on Store C may displace the line containing D, causing another miss during Store D. Both Store C and Store D could have been cache hits in the sequential case.

Let, N_s be the number of misses in the sequential code and N_p be the number of misses in the parallel code. The discussion below is divided into three steps.

a) First, the impact of moving Fetch D on its own cache hit/miss probability, i.e., the hit/miss probability of operand D, is explained. Call it Impact-A.

X:	Fetch A	Fetch A / Fetch D
	Fetch B	Fetch A
	..	..
	compute C := f(A,B)	compute C := f(A,B)
	..	..
	Store C	Store C
Y:	Fetch D	
	..	..
	compute D := f(C,D)	compute D := f(C,d)
	..	..
	Store D	Store D
	..	..
	Sequential	Parallel

Figure 8.1 Two execution scenarios

b) Second, the impact of removing Fetch D on the operand fetches in the vicinity following Y is analyzed; where Y is the program location associated with Fetch D in the sequential code. This impact is referred to as the Impact-B in the following discussion.

c) Finally, the impact of introducing Fetch D on the cache miss probability of operand fetches in the vicinity following X is discussed; where X refers to the program location where Fetch D is moved to in the parallel code. This effect is referred to as the Impact-B in the following discussion.

To keep things simple, it is also assumed that the references moved up during parallelization do not influence the cache hit/miss probability of each other. In other words, they are independent of each other. There may be some references to an operand that were scattered in the sequential code but get grouped together in the parallel code. And hence after parallelization, these may change from cache miss to cache hit due to mutual influence. This effect is ignored.

Impact-A. The following possibilities exist regarding Fetch D, that is moved up in the parallel code:
a) cache miss in sequential, cache miss in parallel,
b) cache hit in sequential, cache hit in parallel,
c) cache hit in sequential, cache miss in parallel, and
d) cache miss in sequential, cache hit in parallel.

Cases (a) and (b) do not change the number of misses, N_p with respect to N_s. Therefore only the remaining two cases need to be examined. Considering impact-A alone,

$$N_p = N_s + R * [Prob\ (hit\ in\ sequential\ but\ miss\ in\ parallel\ code) - Prob\ (hit\ in\ parallel\ but\ miss\ in\ sequential\ code)] \quad \{8.1\}$$

where, R = average number of operand references that are scheduled out-of-sequence. Before proceeding further, following observation may be useful.

An Observation: Consider the following more generalized version of the code sequences given above:

X_i:		X_i:	*op* D
X_{i+1}:		X_{i+1}:	
..		..	
..		..	
X_{i+n}:	*op* D	X_{i+n}:	
Sequential		Parallel	

Let *op* = *Fetch*. In the parallelized code, since the read reference to operand D can be is moved up from from X_{i+n} to X_i, it implies that program locations X_{i+n-1} to X_{i+1} do not contain any write reference to the operand D. Otherwise, it would mean violation of essential data dependency. These intermediate locations are not likely to contain any read references to operand D either for the following reasons:

a) If there is a preceding Fetch D, the compiler is expected to move that reference instead of the one at X_{i+n},

b) If there are some intermediate read references to the operand D, it would not be a smart register allocation scheme. Since the register being used in the parallelized code to hold the operand from X_i to X_n could also have been used in the sequential code to get rid of the intermediate read references.

Let *op* = *Store*. Again, program locations X_{i+1} through X_{i+n-1} can not contain any read or write references to operand D, because that would imply that the move in the parallelized code is in violation to order and output dependency respectively. Note that renaming techniques for bypassing these dependencies are being ignored here.

Therefore, it can be concluded that when a particular operand reference gets moved up during parallelization, there are no additional references in the sequential code to that operand during the scheduling distance. Also note that if one were solely limited by data dependencies, an operand reference being moved up can only be stopped by another reference to the same operand and therefore would always be a cache hit. In other words, if scheduling were to be only restricted by data dependencies and not by resource or control dependencies, all the relocated references should be cache hits due to their grouping with their previous references. This can be used to calculate a lower bound for N_p.

Assume that a particular memory location is bound to a specific operand all through the program. Refer to Fig. 8.2. A *quiet-period* is defined as the time period between successive references to the same location. Typically, minimum quiet period would be determined by compiler's inability to hold on to a temporary (intermediate) result, which in turn is a function of the number of available registers etc. Hence it is likely to show identical distribution on a given system (at least for the same type of variables, like global, local etc.). On the other hand, maximum or actual quiet-period would be a

function of the specific program context and hence may not have a distribution invariant across different programs.

Now consider a main memory location's multiple entries and exits to the cache as depicted in Fig. 8.3. An operand reference is considered a *virgin-hit* if $t_0 > 0$. In other words, first hit to a prefetched variable is called a virgin-hit. Note that, $t_0 = 0$ for a variable fetched on a cache miss. Assume that an average distribution exists for the parameters shown in Figs. 8.2 and 8.3.

Using the terminology developed so far, the probability terms associated with Equation (8.1) can be calculated.

$$Prob(\text{hit in seq. but miss in para.}) =$$

$$Prob(\text{miss in para.}) * Prob(\text{hit in seq.} \mid \text{miss in para.}) \qquad \{8.2\}$$

$$Prob(\text{miss in para.}) = Prob(\text{scheduling distance} > E[t_0]) - \varepsilon \qquad \{8.3\}$$

where,

The conditional probability is given by the virgin-hit probability,

$\varepsilon = n * E[cache\text{-}period / cache\text{-}cycle]_{quiet\text{-}period}$, and,

n = expected number of cache-cycles in a time period = (scheduling distance $- t_0$)

The *quite-period* subscript above implies that the ratio statistics for cache-period to cache-cycle should preferably be collected during the quiet period of the variable.

$$Prob(\text{miss in seqential but hit in parallel}) =$$

$$Prob(\text{hit in parallel}) * Prob(\text{miss in seqential} \mid \text{hit in parallel}) \ . \qquad \{8.4\}$$

Unlike the previous case, hit in the parallel code does not qualify the sequential miss in any particular way, so the conditional probability in Equation (8.4) above can be considered same as the normal cache miss probability. Whereas,

$$Prob(\text{hit in parallel}) = \varepsilon \ \text{ with } t_0 = 0 \ . \qquad \{8.5\}$$

Impact-B and Impact-C. Assume an LRU (least recently used) replacement policy for the cache. Cache hits can be classified into those to the most recently used line (MRU) and those to a not most recently used line (NMRU). If the hit in the sequential case (the parallel case) is a *MRU* hit, the original (reference that is scheduled out-of-sequence) operand reference has no impact on the immediate surrounding. On the other hand, both the *NMRU* hit and a miss can have an impact on the following surrounding. Every

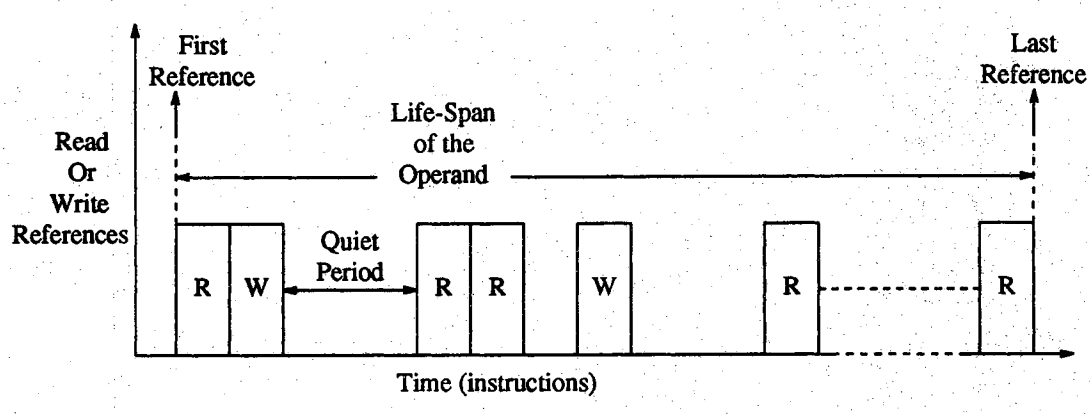
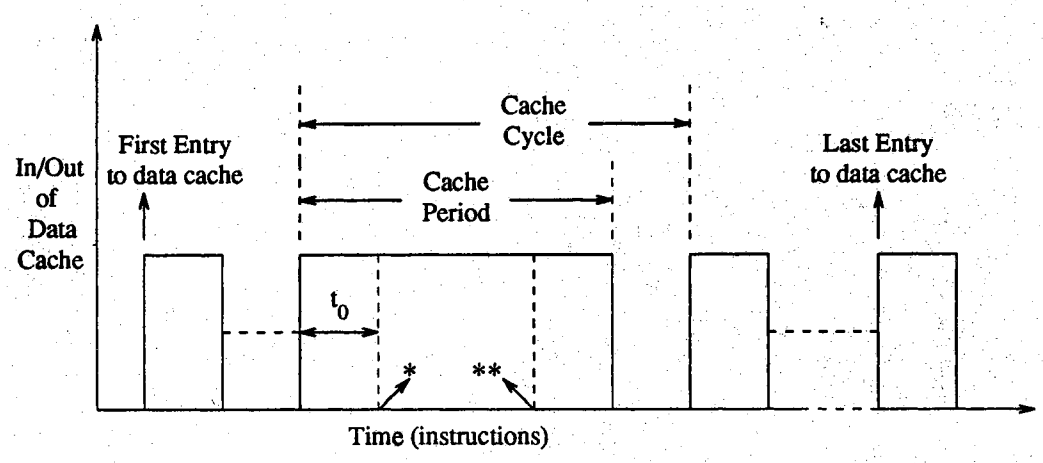


Figure 8.2 Reference pattern for a memory location bound to certain logical operand.



* refers to the time of first reference to the location since the cache-line entry
** refers to the time of last reference to the location before the cache-line exit

Figure 8.3 Entries and exits out of data cache for a memory location bound to certain logical operand.

reference that is scheduled out-of-sequence adds following number of misses:

$$\gamma_1 = Prob\text{ (hit in para.)} * \alpha_1 + Prob\text{ (miss in para.)} * \alpha_2 - Prob\text{ (hit in seq.)} * \alpha_1 - Prob\text{ (miss in seq.)} * \alpha_2 \quad , \qquad \{8.6\}$$

where

$\alpha_1 = Prob\ (NMRUhit) *$

fraction of dead lines in NMRU lines *

average number of misses per dead line

$\alpha_2 =$ fraction of misses that fetch dead lines *

average number of misses per dead line

Note that, hit/miss probabilities for the sequential and the parallel case are same as those calculated during impact-A calculations. A cache line is considered *dead* if it is going to be flushed out before its next reference, else it is called a *live* line. Also note that the probability of a dead line in the sequential case becoming a live line after the relocated reference hit in the parallel case has been ignored.

To calculate the additional number of hits, note that an *NMRU* hit to a live line does not add any hits, since the line would have become most recently used anyway on its subsequent hit. On the other hand, a cache miss that results in fetching a live line does add hits. Therefore, additional number of hits per relocated reference is given by:

$$\gamma_2 = Prob\text{ (miss in para.)} * \alpha_3 - Prob\text{ (miss in seq.)} * \alpha_3 \qquad \{8.7\}$$

where

$\alpha_3 =$ fraction of misses that fetch live lines * average number of hits per live line

As a result, considering impacts B and C only,

$$N_p = N_s + R * (\gamma_1 - \gamma_2) \qquad \{8.8\}$$

where, γ_1 and γ_2 are as given by Equations (8.6) and (8.7). Finally, Equations (8.1) and (8.8) can be combined to yield the complete picture. Note that the scheduling distance probability in Equation (8.3) can be computed using the approach described in Chapter 5. The remaining probability terms in Equations (8.3) through (8.7) can be computed using a cache simulator, modified to compute the virgin hit probability also.

8.2 Cost/Performance Tradeoffs for Concurrency Detection in Different Execution Phases

As explained in Chapter 1 before execution, any end user task goes through stages of transformation from the level of algorithm formulation to high level language specification, followed by assembly and possibly microcode translations. The available amount of parallelism increases as the transformation proceeds from the algorithm level to the microcode level. But detection of the additional amount of available parallelism at the later stages of transformation has additional cost associated with it. This implies a cost-performance design tradeoff aimed at extracting large amount of parallelism without incurring prohibitive cost.

Machines have been built with parallelism detection and scheduling at all of the stages of transformations (refer to Chapter 1). The ability to quantify the amount of parallelism and the cost of its extraction, as discussed in Chapter 5 (using p_δ and p_ω distributions) can be a useful tool in analyzing the cost-performance tradeoff for the proper level of concurrency detection. Chapter 5 provides the plots for these statistics for a VLIW machine (Multiflow) and efforts are underway to collect the same data at the level of high level language specification.

There are two other factors that influence the available amount of parallelism and the associated cost. First, the language used for specification at the high level or at the assembly level. There may be built-in dependencies in the specification syntax. For example if there are separate instructions used for setting the condition code and branching, then almost invariably the condition code setting would be followed by a branch that is dependent on the preceding instruction that set the condition code. Such built-in dependencies would limit the available amount of parallelism for a given scope. By collecting the p_δ and p_ω distributions for a variety of languages on a common set of application tasks, a quantitative comparison can be made on the basis of the amount of parallelism exposed and the associated cost. Future research is being targeted at comparing different high level languages (such as FORTRAN and C) as well as some assembly level instruction sets (such as the Intel x86, the Motorola 68x, the IBM RS6000, and the Sun Sparc).

Second, whether the concurrency detection is done at compile time or run time, has an impact on the amount of available parallelism and the cost of its extraction. For example, at compile time, even a scope as large as several hundreds of instructions is feasible without a very high cost in terms of space and compilation time. But at run time, there is a significant cost associated with a large scope, in terms of the number of instructions that simultaneously need to be examined and the number of pending branches.

8.3 Other Measures for Distance Between Instruction Pairs

This dissertation is primarily aimed at analyzing the instruction window size tradeoffs for a machine with dynamic (run time) scheduling and speculative execution. The typical input for such run time schedulers is the dynamic instruction sequence. Consequently the density of available parallelism and the cost of its extraction (as measured using p_δ and p_ω, respectively) have been estimated as a function of the number of such instructions being examined (W) and the number of pending branches (L). There is another reason for describing p_δ and p_ω in terms of the number of intervening instructions. As indicated in Chapter 5, intuitively one would assume that the output of an instruction is more likely to be consumed in the immediate vicinity than much farther. At the assembly or microcode level, this immediate vicinity can be quantified in terms of the number of following instructions. But at the level of high level languages, this may not be a good measure of immediate vicinity. For example, consider a machine capable of directly executing instructions specified in a high level language, such as the two pieces of code in Figure 8.4. Intuitively one would assume B[i,j] to be equally likely to be dependent on A[i-1,j] and A[i,j-1]. But in terms of number of run time intervening instructions, the dependent instructions in Example (i) are separated by 6 instructions, whereas, those in Example (ii) are separated by 51 instructions. Although different iteration instances of an instruction may be at varying distances, the probability of dependence for any pair should be expected to be close if they are equidistant along any one of the dimensions. Thus at the level of high level language specification, when multi-dimensional references are involved, the immediate vicinity may be better characterized using some measure that treats equally every dimension. Such measures of parallelism can also be used as heuristics in choosing the dimensions that would be most profitable to unroll in *loop quantization* techniques discussed in [Nic88]. Finally, for machines doing static scheduling, a better distance measure between two instructions may be the number of arcs in the uncompacted program flow graph.

8.4 Recursive Performance Modelling

As mentioned earlier in Chapter 1, given a certain end-user task, the most obvious performance measure is the amount of real time spent in performing the task, as measured (or perceived) by the end-user. Assume a synchronous computer system with a global clock. The frequency of this clock is determined by the peak rate at which the system is designed to deliver the results. Imagine the user monitoring the system output every clock cycle for computing the actual throughput. During every cycle, either a

```
for i=1 to 10
for j=1 to 10
1:
2:
3: A[i,j] := C[i,j]
4: B[i,j] := A[i,j-1]
5:
end;
end;
```

Example - i

```
for i=1 to 10
for j=1 to 10
1:
2:
3: A[i,j] := C[i,j]
4: B[i,j] := A[i-1,j]
5:
end;
end;
```

Example - ii

Figure 8.4 Dependence across iterations

result is available or just a bubble (implying: *no result*). The frequency of these bubbles at the system output is enough to compute the actual system throughput. The performance modelling approach described ahead offers some suggestions for recursively computing the probability of receiving bubbles at the system output.

The probability of *bubble-transmission*, p_t from a system stage to its successor can be computed using the probability of *bubble-generation*, p_g and that of *bubble-reception*, p_r. A stage is said to receive a bubble if the preceding stage does not provide any intermediate result during a cycle. A stage is said to generate a bubble if the duration of its computation on some input from preceding stage exceeds the clock cycle. A stage transmits bubbles either if it generates one or if it receives a bubble when its not generating one. Mathematically,

$$p_t = p_g + (1 - p_g) p_r$$

One can compute p_g using a detailed model for that stage and p_r is same as p_t from the preceding stage. Thus recursively the probability of bubbles being transmitted to the user (which determines the user-perceived system throughput) can be computed. For example, the model developed in Chapter 2 can be used as the basis for computing p_g for individual pipelines, or, p_δ information from Chapter 5 can be used for computing the bubbles generated by the scheduler.

One advantage of such an approach lies in the fact that while a low-level detailed model can be used for computing the p_g for a given stage, the low-level model can then be abstracted using p_t information to the next stage. As a result, an analytical model for the entire system (i.e., processor, memory and I/O combined) may also be feasible. The alternative approach would be a simulation-based low-level model for the entire system, which would be many times slower (most likely, prohibitively slow) than this proposed analytic approach.

LIST OF REFERENCES

[AKT86] R.D. Acosta, J. Kjelstrup, and H. C. Torng, "An instruction issuing approach to enhancing performance in multiple functional unit processors", *IEEE Trans. on Computers*, vol. C-35, Sep. 1986, pp. 815-828.

[AgC87] T. Agerwala and J. Cocke, "High performance reduced instruction set processors", *Technical Report RC12434 (#55845)*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, Jan. 1987.

[ArG82] Arvind and K. P. Gostelow, "The U interpreter", *IEE Computer* vol. 15, Feb. 1982, pp. 42-50.

[AST67] D. W. Anderson, F. J. Sparcio, and R. M. Tomasulo, "The IBM System/360 Model 91: Machine philosophy and instruction handling", *IBM Journal of Research and Development* vol. 11, Jan. 1967, pp. 8-24.

[BBE70] J. L. Baer, D. P. Bovet, and G. Estrin, "Legality and other properties of graph models of computations", *J. Ass. Comput. Mach.*, vol. 17, July 1970, pp. 543-552.

[BBL91] D. Bailey, J. Barton, T. Lasinski, and H. Simon, "The NAS parallel benchmarks", *Report RNR-91-002*, NASA Ames Research Center, Jan 1991.

[Ban79] U. Banerjee, *Speedup of Ordinary Programs*, Ph.D. Dissertation, Dept. of Computer Science, Univ. of Illinois, Oct. 1979.

[CGL89] R. Cohn, T. Gross, M. Lam, and P. S. Tseng, "Architecture and compiler tradeoffs for a long instruction word microprocessor", *Proc. of ASPLOS III*, April 1989, pp. 2-14.

[CKP90] G. Cybenko, L. Kipp, L. Pointer, and D. Kuck, "Supercomputer performance evaluation and the perfect benchmarks" CSRD Rept. No. 965 University of Illinois, Urbana, IL, March 1990.

[CNO88] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth, and P. K. Rodman, "A VLIW architecture for a trace scheduling compiler", *IEEE Trans. on Computers*, vol. C-37, Aug. 1988, pp. 967-979.

[Cot65] L. W. Cotten, "Circuit implementations of high-speed pipeline systems", AFIPS Fall Joint Computer Conference, 1965, pp. 489-504.

[Cyt86] R. Cytron, "Doacross: Beyond vectorization for multiprocessors (Extended Abstract)", *1986 International Conference on Parallel Processing*, pp. 836-844.

[DeL87] J. A. DeRosa and H. M. Levy, "An evaluation of branch architectures", *Proc. 14th Annual Symposium on Computer Architecture*, June 1987, pp. 10-16.

[Den70] P. J. Denning, "Virtual memory", *Computing Surveys*, vol. 2, Sep. 1970, pp. 153-188.

[DeM74] J. B. Dennis and D. P. Misunas, "A preliminary architecture for a basic data-flow processor", *Proc. 2nd Annual Symposium on Computer Architecture*, 1974, pp. 126-132.

[DiM87] D. R. Ditzel and H. R. McLellan, "Branch folding in CRISP microprocessor", *Proc. 14th Annual Symposium on Computer Architecture*, June 1987, pp. 2-9.

[DuF89] P. Dubey and M. J. Flynn, "Branch strategies: Modelling and optimization", *Technical Report No. CSL TR 90-411*, Computer Systems Laboratory, Stanford University, Feb. 1990.

[DuF90] P. K. Dubey and M. J. Flynn, "Optimal pipelining", *Journal of Parallel and Distributed Computing*, Jan. 1990, pp. 10-19.

[DuF91] P. K. Dubey and M. J. Flynn, "Branch strategies: modelling and optimization", *IEEE Trans. on Computers*, to appear.

[Faw75] B. K. Fawcett, *Maximal Clocking Rates for Pipelined Digital Systems*, M.S. Thesis, Dept. of Elec. Eng., University of Illinois at Urbana-Champaign, 1975.

[Fis81] J. A. Fisher, "Trace Scheduling: A technique for global microcode compaction", *IEEE Trans. on Computers*, vol. C-30, July 1981, pp. 478-490.

[Fis83] J. Fisher, "VLIW architectures and the ELI-512", *Proc. 10th Annual Symposium on Computer Architecture*, June 1983, pp. 140-150.

[Fly72] M. J. Flynn, "Some computer organizations and their effectiveness", *IEEE Trans. on Computers* vol. C-21, no. 9, Sep. 1972, pp. 948-960.

[FlH79] M. J. Flynn and W. L. Hoevel, "A theory of interpretive architectures: ideal language machines", *Technical Report 170*, Computer Systems Laboratory, Stanford University, Feb. 1979.

[FoR72] C. C. Foster and E. M. Riseman "Percolation of code to enhance parallel dispatching and execution", *IEEE Trans. on Computers*, vol. C-21, Dec. 1972, pp. 1411-1415.

[GaH80] L. C. Garcia and T. Huynh, "Storage fetch contention reduction using instruction branch prediction", *IBM Technical Disclosure Bulletin*, vol. 23, no. 6, 1980.

[GaJ79] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman Publishing Co., 1979.

[GrH86] T. R. Gross and J. Hennessey, "Optimizing delayed branches", *Proc. 15th Workshop on Microprogramming*, 1986.

[GKT90] G. F. Grohoski, J. A. Kahle, L. E. Thatcher, and C. R. Moore, "Branch and fixed-Point instruction execution units", IBM RISC System/6000 Technology, Publication No. SA23-2619, IBM Corporation, 1990.

[Gro83] T. Gross, "Code optimizations of pipeline constraints", *Technical Report No. CSL TR 83-255*, Computer Systems Laboratory, Stanford University, Dec. 1983.

[HaF72] T. G. Hallin and M. J. Flynn, "Pipelining of arithmetic functions", *IEEE Trans. on Computers*, Vol. C-21, No. 8, Aug. 1972, pp. 880-886.

[HwP87] W-M. Hwu and Y. N. Patt, "Checkpoint repair for high performance out-of-order execution machines", *Proc. 14th Annual Symposium on Computer Architecture*, June 1987, pp. 18-26.

[HsD86] P. Y. T. Hsu and E. S. Davidson, "Highly concurrent scalar processing", *Proc. 13th Annual Symposium on Computer Architecture*, June 1986, pp. 386-395.

[Joh91] W. M. Johnson, *Superscalar Microprocessor Design* Prentice Hall, 1991.

[JoW89] N. P. Jouppi and D. W. Wall, "Available instruction-level parallelism for superscalar and superpipelined machines", *Proc. of ASPLOS III*, April 1989, pp. 272-282.

[KaM66] R. M. Karp and R. E. Miller, "Properties of a model for parallel computations: Determinacy, termination, queueing", *SIAM J. of Applied Math.*, Nov. 1966, pp. 1390-1411.

[KMC72] D. Kuck, Y. Muraoka, and S. Chen, "On the number of operations simultaneously executable in Fortran-like programs and their resulting speedup", *IEEE Trans. on Computers*, vol. C-21, Dec. 1972, pp. 1293-1310.

[Kun82] H. T. Kung, "Why systolic architectures ?", *IEEE Computer*, Jan. 1982, pp. 37-46.

[KuS86] S. R. Kunkel and J. E. Smith, "Optimal pipelining in supercomputers", *Proc. 13th Annual Symposium on Computer Architecture*, 1986, pp. 404-411.

[Lam88] M. Lam, "Software pipelining: An effective scheduling technique for VLIW machines", *Proc. SIGPLAN '88 Conf. Prog. Lang. Design and Implementation*, June 1988, pp. 318-328.

[LeS84] J. K. Lee and A. J. Smith, "Branch prediction strategies and branch target buffer design", *IEEE Computer*, vol. 17, Jan. 1984, pp. 6-22.

[LiY90] D. J. Lilja and P. C. Yew, "Comparing parallelism extraction techniques: Superscalar processors, pipelined processors, and multiprocessors", *International Conference on Parallel Processing*, 1990, pp. I-563 - I-564, and "The performance potential of fine-Grain and coarse-grain parallel architectures", *Report No. 954*, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, June 1990.

[McH86] S. McFarling and J. Hennessey, "Reducing the cost of branches", *Proc. 13th Annual Symposium on Computer Architecture*, June 1986, pp. 396-403.

[Mil73] R. E. Miller, "A comparison of some theoretical models of parallel computation", *IEEE Trans. on Computers*, Aug. 1973, pp. 710-717.

[NiF84] A. Nicolau and J. Fisher, "Measuring the parallelism available for very long instruction word architectures", *IEEE Trans. on Computers*, vol. C-33, Nov. 1984, pp. 968-976.

[Nic85] A. Nicolau, "Uniform parallelism exploitation in ordinary programs", *Proc. International Conference on Parallel Processing*, Aug. 1985, pp. 614-618.

[Nic88] A. Nicolau "Loop Quantization: A generalized loop unwinding technique" *Journal of Parallel and Distributed Computing*, vol. 5, Oct. 1988, pp. 568-586.

[Nic89] A. Nicolau, "Run-time disambiguation: coping with statically unpredictable dependencies", *IEEE Trans. on Computers*, vol. C-38, May 1989, pp. 663-678.

[Nil80] N. J. Nilsson, *Fundamentals of Artificial Intelligence*, Tioga Publishing Co., 1980.

[PeS77] B. L. Peuto and L. J. Shustek, "Current issues in the architecture of microprocessors", *IEEE Computer*, Feb. 1977, pp. 20-25.

[PHS85] Y. N. Patt, W-M. Hwu, and M. Shebanow, "HPS, A new microarchitecture: Rationale and introduction", *Proc. 18th Annual Workshop on Microprogramming*, Dec. 1985, pp. 103-108.

[PaD76] J. H. Patel and E. S. Davidson, "Improving the throughput of a pipeline by insertion of delays", *Proc. 3rd Annual Symposium on Computer Architecture*, June 1976, pp. 159-164.

[Pol86] C. D. Polychronopoulos, *On Program Restructuring, Scheduling and Communication for Parallel Processor Systems*, Ph.D. dissertation, Dept. of Computer Science, Univ. of Illinois, Aug. 1986.

[RaG81] B. R. Rau and C. D. Glaeser, "Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing", *Proc. 14th Annual Workshop on Microprogramming*, Oct. 1981, pp. 183-198.

[RYY89] B. Rau, D. Yen, W. Yen and R. A. Towle, "The Cydra 5 departmental supercomputer", *Computer*, vol. 22, Jan. 1989, pp. 12-35.

[RiF72] E. M. Riseman and C. C. Foster, "The inhibition of potential parallelism", *IEEE Trans. on Computers*, vol. C-21, Dec. 1972, pp. 1405-1411.

[Sha77] H. D. Shapiro, "A comparison of various methods for detecting and utilizing parallelism in a single instruction Stream", *Proc. International Conference on Parallel Processing*, Aug. 1977, pp. 67-76.

[Smi81] J. E. Smith, "A study of branch prediction strategies", *Proc. 8th Annual Symposium on Computer Architecture*, May 1981, pp 135-148.

[SmP85] J. E. Smith and A. R. Pleszkun, "Implementation of precise interrupts in pipelined processors", *Proc. 12th Annual Symposium on Computer Architecture*, June 1985, pp. 36-47.

[SJH89] M. D. Smith, M. Johnson, and M. A. Horowitz, "Limits on multiple instruction issue", *Proc. of ASPLOS III*, Boston, MA, April 1989, pp. 290-302.

[SoV87] G. S. Sohi and S. Vajapeyam, "Instruction issue logic in high-performance interruptible pipelined processors", *Proc. 14th Annual Symposium on Computer Architecture*, June 1987, pp. 27-34.

[SoV89] G. S. Sohi and S. Vajapeyam "Tradeoffs in instruction format design for horizontal architectures", *Proc. of ASPLOS III*, Boston, MA, April 1989, pp. 15-25.

[TjF70] G. S. Tjaden and M. J. Flynn, "Detection and parallel execution of independent instructions", *IEEE Trans. Computers*, Vol. C-19, Oct. 1970, pp. 889-895.

[Tja72] G. S. Tjaden, *Representation and Detection of Concurrency Using Ordering Matrices*, Ph.D. dissertation, Johns Hopkins Univ., Baltimore, MD, 1972.

[TjF73] G. S. Tjaden and M. Flynn, "Representation of concurrency with ordering matrices", *IEEE Trans. on Computers*, Aug. 1973, pp. 752-761.

[Tho70] J. E. Thornton, *Design of a Computer: The Control Data 6600*, Glenview, IL, Scott, Foresman and Co., 1970.

[Tom67] R. M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units", *IBM Journal of Research and Development* vol. 11, Jan. 1967, pp. 25-33.

[Uht86] A. K. Uht, "An efficient hardware algorithm to extract concurrency from general-purpose code", *Proc. of the Nineteenth Annual Hawaii International Conference on System Sciences*, 1986, pp. 41-50.

[Wed82] R. G. Wedig, *Detection of Concurrency in Directly Executed Language Instruction Streams*, Ph.D. dissertation, Stanford University, Stanford, CA, June 1982.

[WeS84] S. Weiss and J. E. Smith, "Instruction issue logic in pipelined supercomputers", *Proc. 11th Annual Symposium on Computer Architecture*, June 1984, pp. 110-118.

[WeS87] S. Weiss and J. E. Smith, "A study of scalar compilation techniques for pipelined supercomputers", *Proc. of ASPLOS II*, Palo Alto, CA, Oct. 1987, pp. 105-109.

Appendix A

A.1 Computation of Probabilities in Table 3.2

The four probabilities in Table 3.2, $p_{n,n}$, $p_{n,b}$, $p_{b,n}$ and $p_{b,b}$ can be computed in terms of probability of branch-to-be-taken prediction, p_t and probability of correct prediction, p_c, using the following equations:

$$p_{n,n} = (1 - p_t) * p_c$$

$$p_{n,b} = (1 - p_t) * (1 - p_c)$$

$$p_{b,n} = p_t * (1 - p_c)$$

$$p_{b,b} = p_t * p_c$$

Consider a branch strategy, which on an average predicts 6 out of every 10 branches as likely to be taken and where 2 out of every 10 predictions are incorrect. This yields $p_t = 0.6$, and $p_c = 0.8$, which leads to $p_{n,n} = 0.32$, $p_{n,b} = 0.08$, $p_{b,n} = 0.12$ and $p_{b,b} = 0.48$. This means on an average out of every 100 branches, 32 are not taken as predicted, 8 are taken in spite of not-to-be-taken prediction, 12 are not taken though predicted as likely to be taken, and 48 are taken in accordance with the prediction. Therefore, 56 out of every 100 branches are taken. The number of actually taken branches is independent of the employed branch strategy. It is a characteristic of the program environment under execution and can be expressed as

$$p_{sb} = (1 - p_t) * (1 - p_c) + p_t * p_c \ .$$

For a branch prediction strategy (such as *BTB*) with a certain correct prediction probability, p_c, the probability of to-be-taken predictions can be written using the above equation as,

$$p_t = \frac{p_{sb} - (1 - p_c)}{2p_c - 1} \ .$$

A.2 Additional Instruction Traffic Calculation Under Freeze Conditions

Let,

D_a = average number of clocks spent during a target-address-calculation freeze

D_f = average number of clocks spent in case of a page-fault during target fetch

p_a = probability of a target-address-calculation freeze

p_f = probability of a target fetch freeze

N = maximum possible instruction fetches assuming no freeze

$pos(l) = l$ for $l > 0$

$= 0$ for $l \leq 0$

The additional instruction traffic

$$I^+ = m_1 + m_2 + m_3 + m_4$$

where

m_1 = wasted instruction fetches, assuming address calculation freeze as well as target fetch freeze

$= pos\,(N - D_a - D_f) * p_a * p_f$

m_2 = wasted instruction fetches, assuming address calculation freeze but no target fetch freeze

$= pos\,(N - D_a) * p_a * (1 - p_f)$

m_3 = wasted instruction fetches, assuming no address calculation freeze but target fetch freeze

$= pos\,(N - D_f) * (1 - p_a) * p_f$

m_4 = wasted instruction fetches, assuming no address calculation freeze and no target fetch freeze

$= N * (1 - p_a) * (1 - p_f)$

For the sake of brevity, in the following sections the above calculation will be written

$$I^{+} = N \sim p_a * D_a \sim p_f * D_f \ ,$$

where $\sim$ refers to the probability-based reduction in N as explained above.

The above calculation assumes that no additional instruction traffic is generated during the freeze conditions. Excess instruction traffic would be generated if the freeze conditions require any instruction fetches (for example, during page-fault handling). Incorrect predictions not only result in wasted instruction fetches but may also result in unnecessary operand fetches. For this analysis, the increase in data traffic or any interference of operand fetches with the instruction fetches is ignored.

A.3 Calculation of Branch Delay and Wasted Instruction Traffic

Table A.1 contains the symbols to be frequently used in the calculations to follow. Some of these symbols have been introduced before in Chapter 3 and are reproduced here for easy reference.

Predict branch never taken (PBNT):

$$p_t = 0$$

$$K_{n,n} = 0$$

$$K_{n,b} = (s_b - 1) + p_a * D_a + p_f * D_f$$

$$I^{+}_{n,n} = 0$$

$$I^{+}_{n,b} = s_b - 1$$

Loop buffers (LB):

$$p_t = 0$$

$$K_{n,n} = 0$$

$$K_{n,b} = ((s_b - 1) + p_a * D_a + p_f * D_f) * (1 - p_{lh}) + ((s_b - 2) + p_a * D_a) * p_{lh}$$

$$I^{+}_{n,n} = 0$$

$$I^{+}_{n,b} = (s_b - 1) * (1 - p_{lh})$$

Table A.1 Commonly used symbols.

Average branch frequency	b
Overall fraction of successful branches (conditional and unconditional combined)	p_{sb}
Number of pipeline stages until branch resolution	s_b
Number of pipeline stages until unconditional branch resolution	s_{bu}
Number of pipeline stages until conditional branch resolution	s_{bc}
Number of buffer substages in the instruction fetch stage	s_f
Probability of freeze during target address formation	p_a
Duration of target address calculation freeze	D_a
Probability of freeze during target fetch	p_f
Duration of target-fetch freeze	D_f
Probability of loop buffer hit	p_{lh}
Probability of BTB hit	p_{th}
Probability of correct target address prediction from BTB	p_{ct}
Probability of BTB hit for a non-branch instruction	p_w
Average number of delay slots filled in delayed branch approach	u
Correct prediction probability	p_c
Branch-to-be-taken prediction probability	p_t

Pre-calculate target address (PTA):

$$p_t = 0$$

$$K_{n,n} = 0$$

$$K_{n,b} = (s_b - 1) + p_a * pos(D_a - (s_b - 1 - s_f)) + p_f * D_f$$

$$I^+_{n,n} = 0$$

$$I^+_{n,b} = s_b - 1$$

Target-fetch in the OF-slot (FTOF):

$$p_t = 0$$

$$K_{n,n} = (1 - p_a) * (1 - p_f) + (1 - p_a) * p_f$$

$$K_{n,b} = \alpha_1 * p_a * p_f + \alpha_2 * p_a * (1 - p_f) + \alpha_3 * (1 - p_a) * p_f + \alpha_4 * (1 - p_a) * (1 - p_f)$$

where

$$\alpha_1 = s_b - 1 + D_a + D_f$$

$$\alpha_2 = s_b - 1 + D_a$$

$$\alpha_3 = s_b - 1 + D_f$$

$$\alpha_4 = s_b - 2$$

$$I^+_{n,n} = (1 - p_a) * (1 - p_f)$$

$$I^+_{n,b} = \beta_1 * p_a * p_f + \beta_1 * p_a * (1 - p_f) + \beta_2 * (1 - p_a) * p_f + \beta_2 * (1 - p_a) * (1 - p_f)$$

where

$$\beta_1 = s_b - 1$$

$$\beta_2 = s_b - 2$$

Predict branch always taken (PBAT):

$$p_t = 1$$

$$K_{b,n} = s_b - 1 - s_f$$

$$K_{b,b} = s_f + p_a * D_a + p_f * D_f$$

$$I^{+}_{b,n} = (s_b - 1 - s_f) - p_a * D_a - p_f * D_f$$

$$I^{+}_{b,b} = s_f$$

Predict branch always taken with target copy (PTTC):

$$p_t = 1$$

$$K_{b,n} = s_b - 1$$

$$K_{b,b} = p_a * pos(D_a - (s_b - 1 - s_f)) + p_f * D_f$$

$$I^{+}_{b,n} = s_b - 1$$

$$I^{+}_{b,b} = 0$$

Fetch both the paths (FBP):

$$p_c = 1$$

$$K_{n,n} = 0$$

$$K_{b,b} = s_f + p_a * D_a + p_f * D_f$$

$$I^{+}_{n,n} = (s_b - 1 - s_f) - p_a * D_a - p_f * D_f$$

$$I^{+}_{b,b} = s_b - 1$$

Delayed branch (DB):

$$p_t = 0$$

$$K_{n,b} = pos(s_b - 1 - u) + p_a * D_a + p_f * D_f$$

$$K_{n,n} = 0$$

$$I^{+}_{n,b} = pos(s_b - 1 - u)$$

$$I^{+}_{n,n} = 0$$

Taken/Not-taken switch in the decode stage (TNTD):

$$K_{b,n} = s_b - 1 - s_f$$

$$K_{b,b} = s_f + p_a * D_a + p_f * D_f$$

$$K_{n,b} = s_b - 1 + p_a * D_a + p_f * D_f$$

$$K_{n,n} = 0$$

$$I^+_{b,n} = (s_b - 1 - s_f) - p_a * D_a - p_f * D_f$$

$$I^+_{b,b} = s_f$$

$$I^+_{n,b} = s_b - 1$$

$$I^+_{n,n} = 0$$

Under the Branch target buffer (BTB) scheme, instruction fetch addresses are associatively matched with the buffer contents and in case of a *hit* BTB predicts the most likely branch outcome as well as the most recent target address (see Fig. 3.5). As a result, target fetch does not need to wait for the branch decode and target address calculation. If the branch is likely to be taken, the first target instruction fetch immediately follows the branch instruction fetch. Following branch decode, at the completion of actual target address calculation, a comparison is made with the predicted target address. A mismatch here flushes any fetches made from the incorrect target, aborts any freeze in the incorrect target path and restarts target fetch at the calculated address. It is also assumed that this comparison output is available along with the actual target, without any additional clock overhead. Correct target prediction probability, p_{ct} depends on the frequency of branch target changes.

A hit in the branch target buffer (BTB) means that the fetch address contains a branch instruction. In the case of writable code segments, there is a small likelihood, p_w, that a non-branch instruction gets predicted as a branch instruction. To make things worse, if such an instruction is predicted as a branch likely to be taken then it has an impact on the system throughput even in the absence of any branch instruction as it blocks the sequential address fetch during the following cycle until the actual instruction decode. This throughput deterioration is modelled using the following modified version of the throughput equation in Table 3.2:

$$G = 1/(1 + K * b + s_f * p_w * p_t) \ .$$

Similarly, this probability of a BTB hit with branch-to-be-taken prediction for a non-branch instruction also modifies the computation for wasted instruction traffic in Table 3.2, which so far included additional instruction traffic only due to branch instructions.

The following equation reflects an additional wasted instruction fetch in case a non-branch instruction is predicted as a to-be-taken-branch and there is no target fetch freeze :

$$I^{+} = I_b^{+} * (1-p_w) + (s_f \sim D_f * p_f) * p_t * p_w \ ,$$

where I_b^{+} refers to the excess instruction traffic due to branch instructions given by the equation in Table 3.2 and ~ refers to the probability-based reduction explained in Section A.2.

In case of a miss in BTB, branch instructions are handled in a manner similar to the *PBNT* strategy. In other words, branch is assumed as not likely to be taken by default, in case of a BTB miss. The overhead involved in BTB-updates is ignored. Therefore, equivalently, this strategy can be considered as a combination of two strategies, one as described above with BTB hit probability p_{th} and the other the same as in the case of the *PBNT* scheme with BTB miss probability.

Calculation for different excess instruction traffic parameters is involved in this case and, hence, their derivation is described below in qualitative terms before giving the mathematical details.

i) If a branch is predicted as likely to be taken, target fetch begins immediately from the predicted address and target address calculation starts soon after the decode. If the calculated target address does not match with the predicted address, the instructions fetched so far from the incorrect address are wasted and instruction fetches begin from the calculated (actual) target address.

ii) In the previous case, if the branch is not taken then in addition to the fetches made from the predicted target, instructions fetched from the actual target address are also wasted.

iii) If the branch is predicted as not likely to be taken, it is assumed that no attempt is made to calculate the target address and instruction fetch continues from the sequential path. If this prediction turns out to be false, sequential instructions fetched are discarded, target fetch immediately begins at the predicted address and the actual target calculation starts simultaneously. If the calculated target does not match with the predicted target, this fetched sequence is also wasted.

Finally, the above set of parameters are used to calculate the average branch delay, K_h, and excess instruction traffic, I_h^{+}, where the subscript refers to the BTB hit case. In the case of BTB miss, the corresponding parameters K_m and I_m^{+} are calculated from the components given for the *PBNT* case. Combining these cases gives

$$K = K_h * p_{th} + K_m * (1 - p_{th})$$

and

$$I_b^{+} = I_h^{+} * p_{th} + I_m^{+} * (1 - p_{th}) \ .$$

The computation of different parameters follows.

$$K_{b,n} = s_b - 1$$

$$K_{b,b} = (p_f * D_f) * p_{ct} + (s_f + p_a * D_a + p_f * D_f) * (1 - p_{ct})$$

$$K_{n,b} = ((s_b - 1) + p_f * D_f) * p_{ct} + ((s_b - 1) + p_a * D_a + p_f * D_f) * (1 - p_{ct})$$

$$K_{n,n} = 0$$

$$I^{+}_{n,n} = 0$$

The additional instruction traffic, when branch is predicted as to-be-taken and is actually taken is

$$I^{+}_{b,b} = (\sigma_1 + \sigma_2 + \sigma_3 + \sigma_4) * (1 - p_{ct}) \ .$$

Assume

$$\min(a,b) = a \ \text{ if } a \le b$$

$$= b \ \text{ if } b < a$$

$$\sigma_1 = \min(pos(s_f + D_a - D_f), (s_b - 1)) * p_a * p_f$$

$$\sigma_2 = \min((s_f + D_a), (s_b - 1)) * p_a * (1 - p_f)$$

$$\sigma_3 = pos(s_f - D_f) * (1 - p_a) * p_f$$

$$\sigma_4 = s_f * (1 - p_a) * (1 - p_f)$$

Consider the additional instruction traffic, when branch is predicted as to-be-taken but turns out to be an incorrect prediction : Let p_{fp} and D_{fp} refer to the freeze potential and freeze duration respectively at the predicted target, and, let p_{fc} and D_{fc} refer to the same at the actual calculated target. This distinction is made only for better understanding of the following details. Numerically, $p_{fp} = p_{fc} = p_f$ and $D_{fp} = D_{fc} = D_f$. The discarded instruction fetches in this case would be

$$I^{+}_{b,n} = ((s_b - 1) - p_f * D_f) * p_{ct} + \delta * (1 - p_{ct}) \ ,$$

where

$$\delta = \delta_1 + \delta_2 + \delta_3 + \delta_4 + \delta_5 + \delta_6 + \delta_7 + \delta_8$$

$$\delta_1 = (pos(s_f - D_{fp}) + s_b - 1 - s_f) * p_{fp} * (1 - p_{fc}) * (1 - p_a)$$

$$\delta_2 = (pos(s_f - D_{fp}) + pos(s_b - 1 - s_f - D_{fc})) * p_{fp} * p_{fc} * (1 - p_a)$$

$$\delta_3 = (s_b - 1) * (1 - p_{fp}) * (1 - p_{fc}) * (1 - p_a)$$

$$\delta_4 = (s_f + pos(s_b - 1 - s_f - D_{fc})) * (1 - p_{fp}) * p_{fc} * (1 - p_a)$$

$$\delta_5 = (min\ (pos(s_f + D_a - D_{fp}), (s_b - 1)) + pos(s_b - 1 - s_f - D_a)) * p_{fp} * (1 - p_{fc}) * p_a$$

$$\delta_6 = (min\ (pos(s_f + D_a - D_{fp}), (s_b - 1)) + pos(s_b - 1 - s_f - D_a - D_{fc})) * p_{fp} * p_{fc} * p_a$$

$$\delta_7 = (s_b - 1) * (1 - p_{fp}) * (1 - p_{fc}) * p_a$$

and

$$\delta_8 = (min\ ((s_f + D_a), (s_b - 1)) + pos(s_b - 1 - s_f - D_a - D_{fc})) * (1 - p_{fp}) * p_{fc} * p_a$$

The additional instruction traffic when the branch is predicted as not-to-be-taken but is actually taken is

$$I^{+}_{n,b} = (s_b - 1) + \gamma * (1 - p_{ct}) \ ,$$

where

$$\gamma = \gamma_1 * p_a * p_f + \gamma_2 * p_a * (1 - p_f) + \gamma_3 * (1 - p_a) * p_f + \gamma_4 * (1 - p_a) * (1 - p_f)$$

and

$$\gamma_1 = \min\ (pos(D_a - D_f), s_b - 1)$$

$$\gamma_2 = \min\ (D_a, s_b - 1)$$

and

$$\gamma_3 = \gamma_4 = 0$$

Finally, calculations for the four hybrid cases follows.

Predict branch always taken with target-copy and delayed branch (TTCDB):

$$p_t = 1$$

$$K_{b,n} = pos(s_b - 1 - u)$$

$$K_{b,b} = p_a * pos(D_a - (s_b - 1 - s_f)) + p_f * D_f$$

$$I^{+}_{b,n} = pos(s_b - 1 - u)$$

$$I^{+}_{b,b} = 0$$

Predict branch always taken with target-copy, delayed branch and loop buffer (TTDLB):

$$p_t = 1$$

$$K_{b,n} = pos(s_b - 1 - u) * (1 - p_{lh}) + pos(s_b - 2 - u) * p_{lh}$$

$$K_{b,b} = p_a * pos(D_a - (s_b - 1 - s_f)) * p_{lh} + (p_a * pos(D_a - (s_b - 1 - s_f)) + p_f * D_f) * (1 - p_{lh})$$

$$I^+_{b,n} = pos(s_b - 1 - u) * (1 - p_{lh})$$

$$I^+_{b,b} = 0$$

Taken/Not-taken switch in the decode stage with loop buffer (TNTLB):

$$K_{b,n} = s_b - 1 - s_f$$

$$K_{b,b} = ((s_f - 1) + p_a * D_a) * p_{lh} + (s_f + p_a * D_a + p_f * D_f) * (1 - p_{lh})$$

$$K_{n,b} = ((s_b - 1) + p_a * D_a + p_f * D_f) * (1 - p_{lh}) + ((s_b - 2) + p_a * D_a) * p_{lh}$$

$$K_{n,n} = 0$$

$$I^+_{b,n} = ((s_b - 1 - s_f) - p_a * D_a - p_f * D_f) * (1 - p_{lh})$$

$$I^+_{b,b} = s_f * (1 - p_{lh})$$

$$I^+_{n,b} = (s_b - 1) * (1 - p_{lh})$$

$$I^+_{n,n} = 0$$

The model parameters for the case of taken/not-taken switch in the decode stage with branch target buffer (TNBTB) are the same as those in the case of BTB, except that the average branch delay and excess instruction traffic parameters K_m and I^+_m in the case of a BTB miss are calculated using the *TNTD* case.

A.4 Some Additional Performance Plots

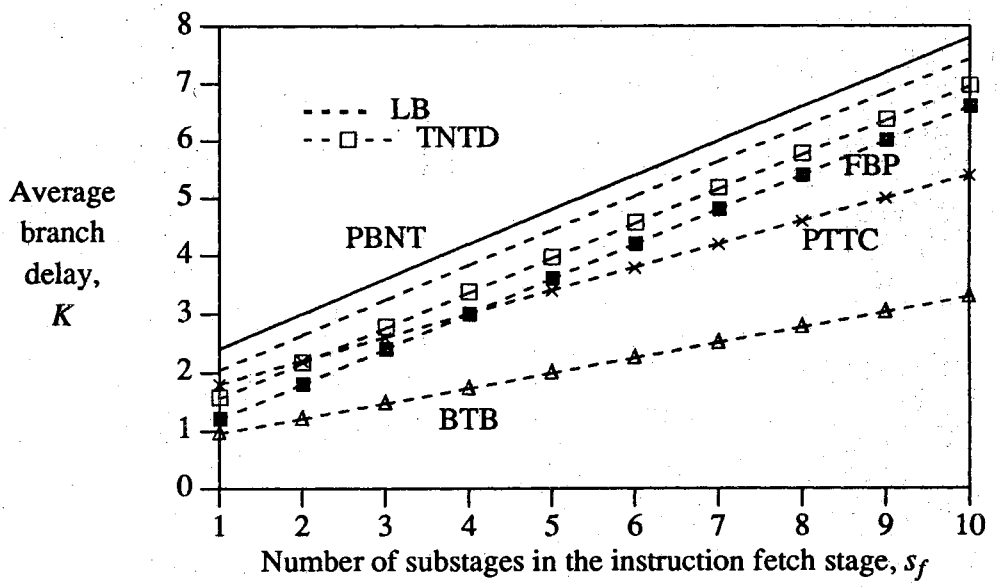


Figure A.1 Average branch delay versus number of substages in the instruction fetch stage for *PBNT*, *LB*, *PTTC*, *FBP*, *TNTD*, and *BTB* strategies.

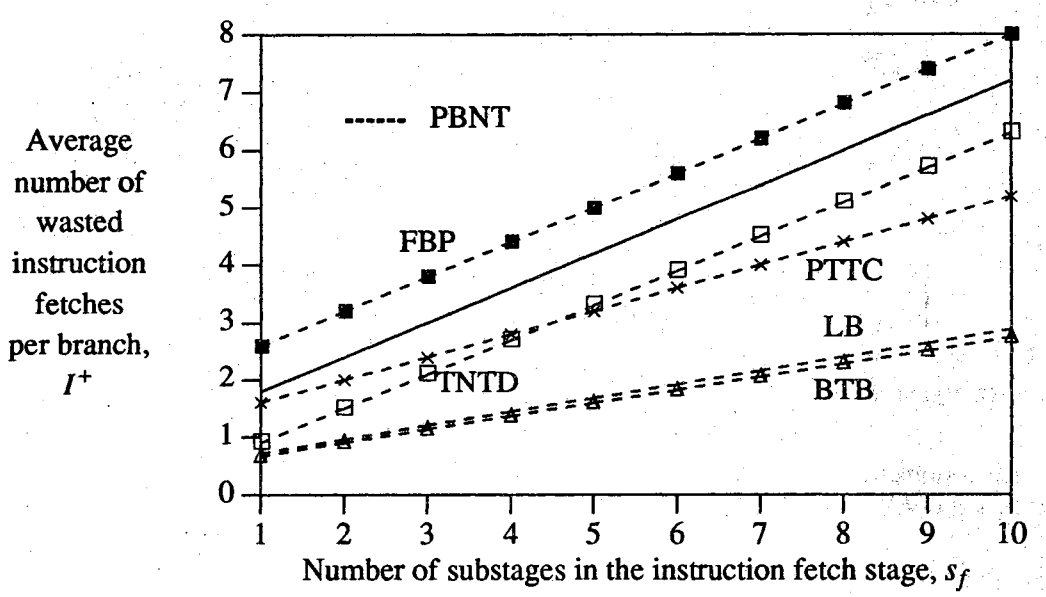


Figure A.2 Average number of wasted instruction fetches per branch versus number of substages in the instruction fetch stage for *PBNT*, *LB*, *PTTC*, *FBP*, *TNTD*, and *BTB* strategies.

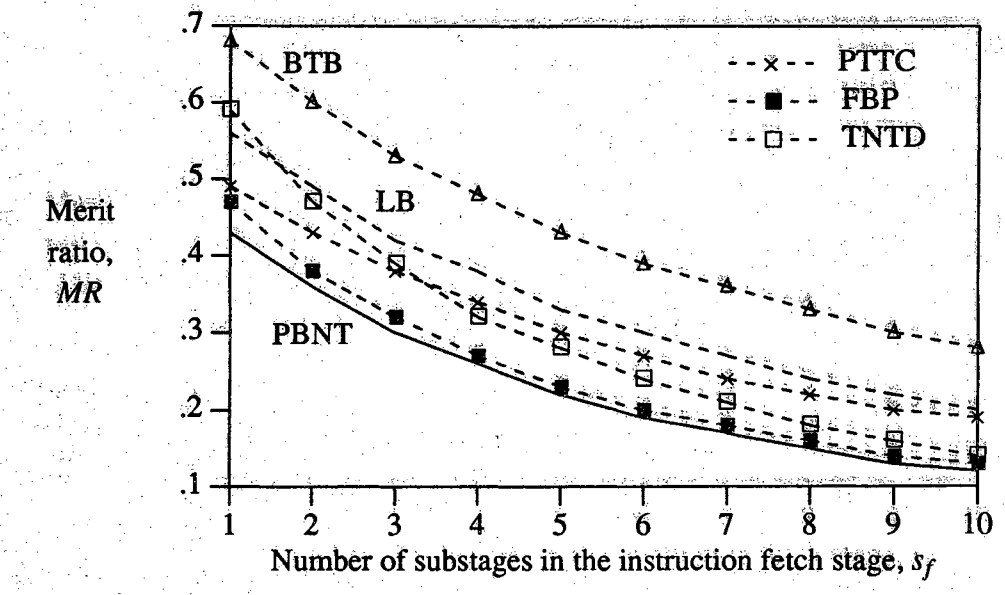


Figure A.3 Merit ratio versus number of substages in the instruction fetch stage for *PBNT*, *LB*, *PTTC*, *FBP*, *TNTD*, and *BTB* strategies.

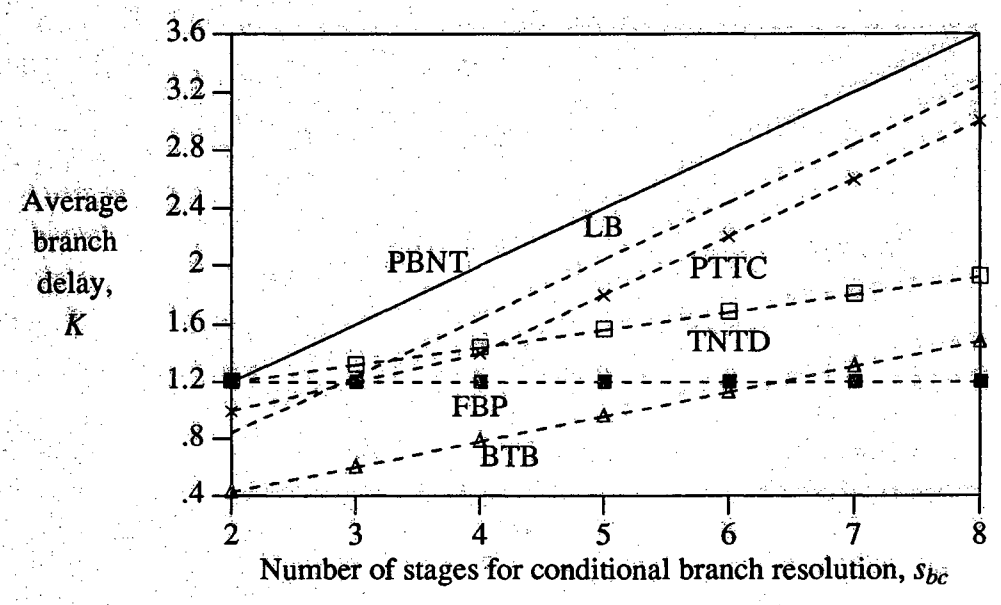


Figure A.4 Average branch delay versus number of stages for conditional branch resolution for *PBNT*, *LB*, *PTTC*, *FBP*, *TNTD*, and *BTB* strategies.

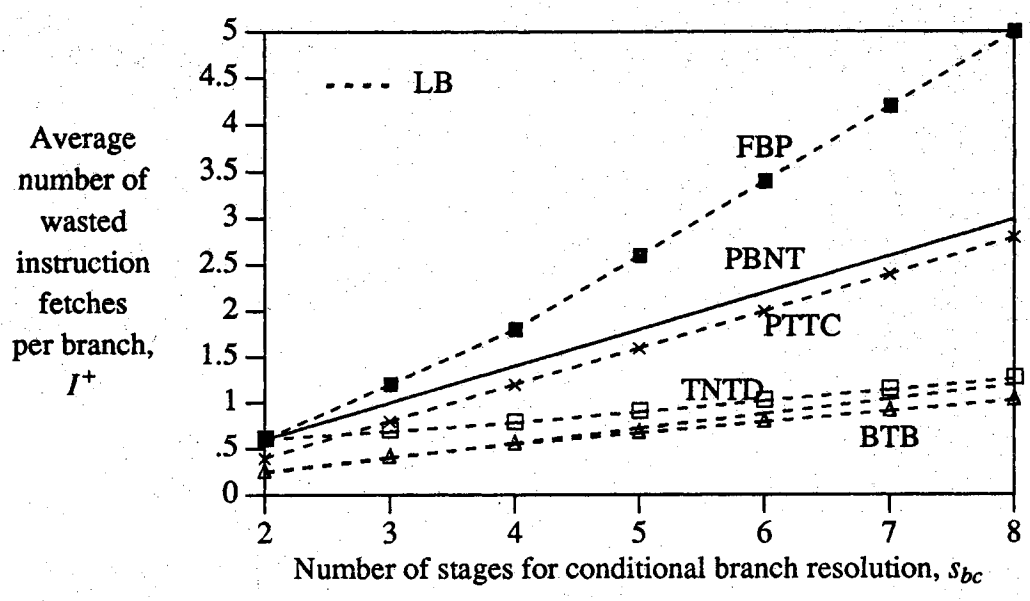


Figure A.5 Average number of wasted instruction fetches per branch versus number of stages for conditional branch resolution for *PBNT*, *LB*, *PTTC*, *FBP*, *TNTD*, and *BTB* strategies.

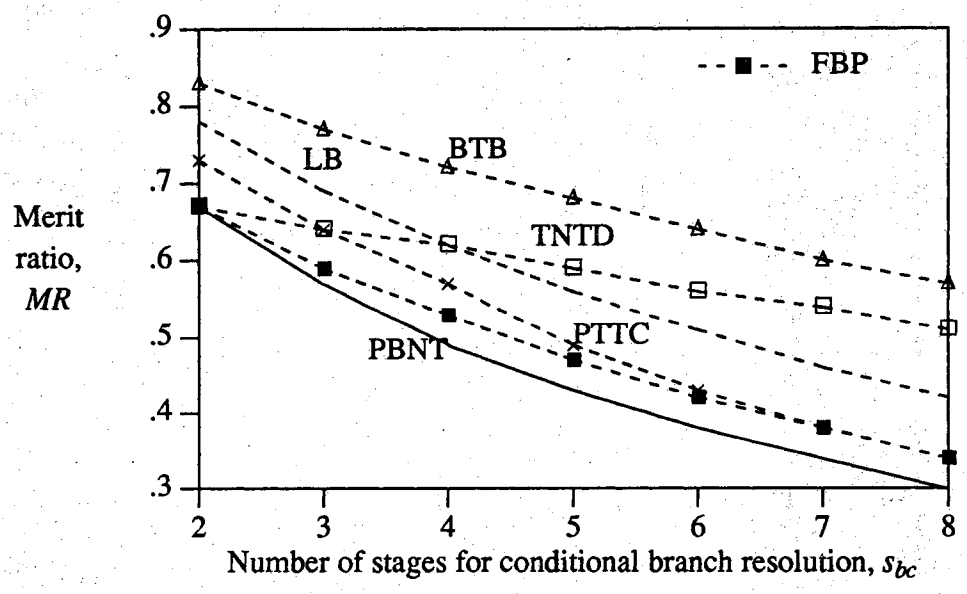


Figure A.6 Merit ratio versus number of stages for conditional branch resolution for *PBNT*, *LB*, *PTTC*, *FBP*, *TNTD*, and *BTB* strategies.

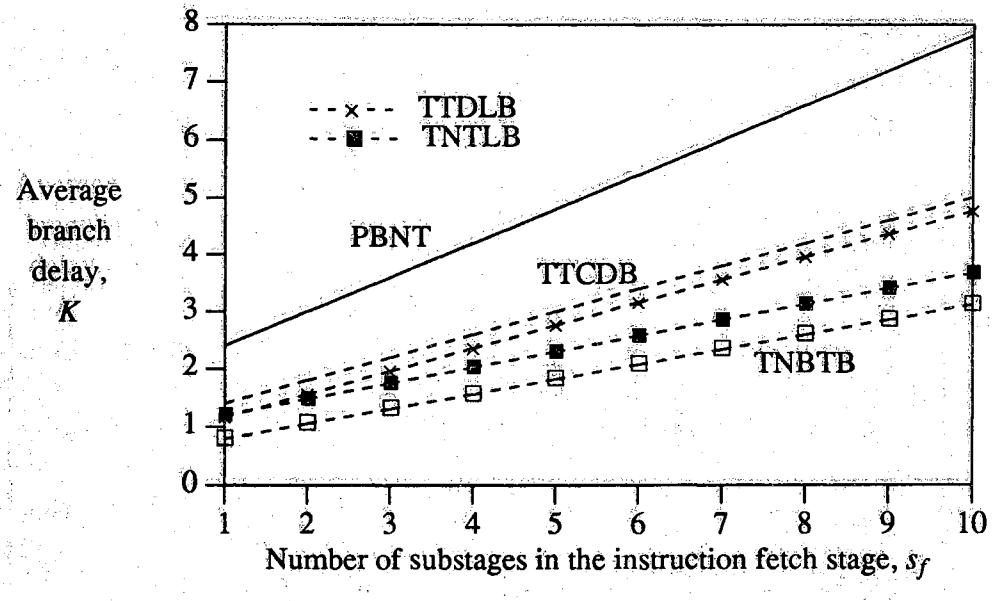


Figure A.7 Average branch delay versus number of substages in the instruction fetch stage for *PBNT*, *TTCDB*, *TTDLB*, *TNTLB*, and *TNBTB* strategies.

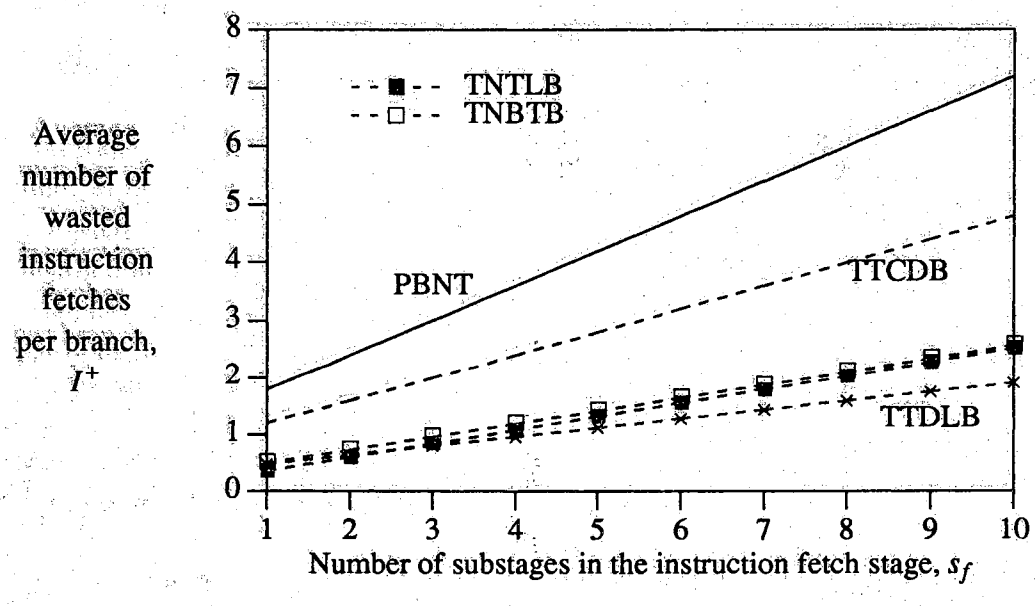


Figure A.8 Average number of wasted instruction fetches per branch versus number of substages in the instruction fetch stage for *PBNT*, *TTCDB*, *TTDLB*, *TNTLB*, and *TNBTB* strategies.

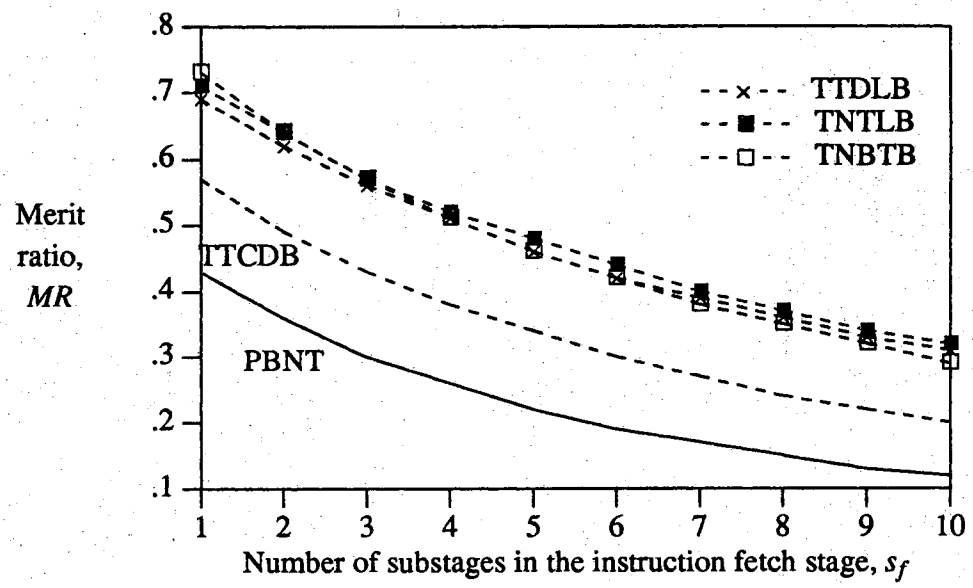


Figure A.9 Merit ratio versus number of substages in the instruction fetch stage for *PBNT*, *TTCDB*, *TTDLB*, *TNTLB*, and *TNBTB* strategies.

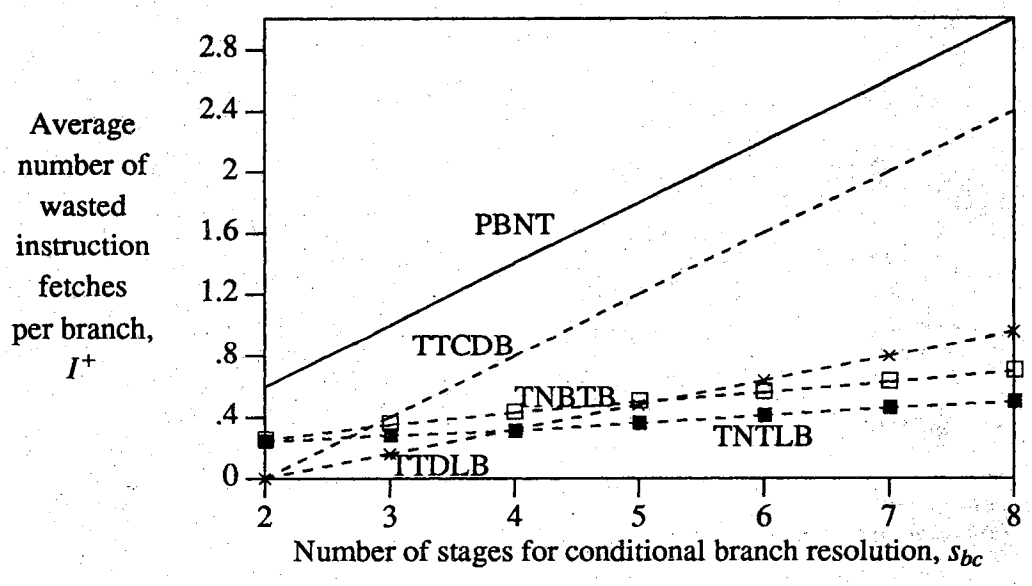


Figure A.10 Average number of wasted instruction fetches per branch versus number of stages for conditional branch resolution for *PBNT*, *TTCDB*, *TTDLB*, *TNTLB*, and *TNBTB* strategies.

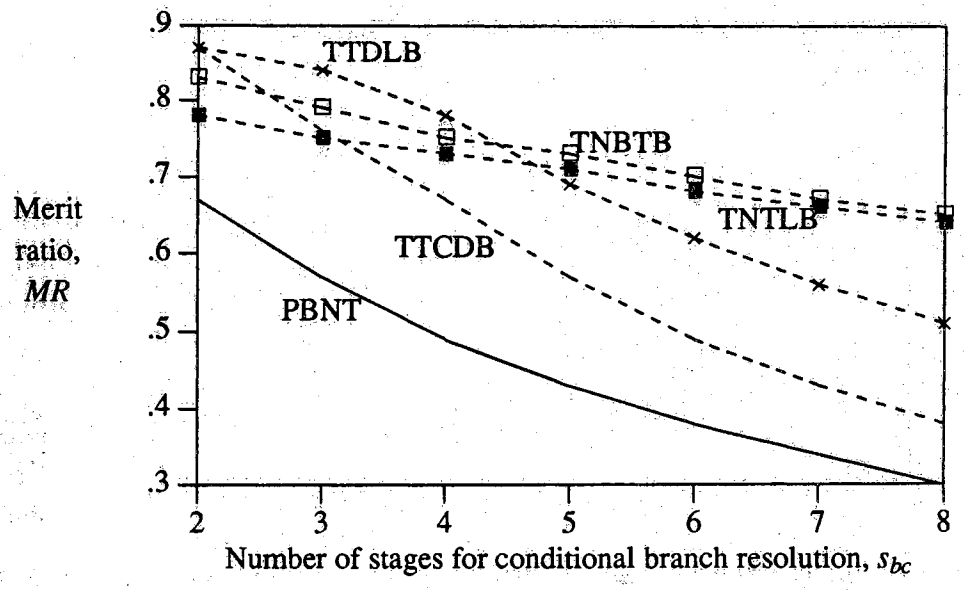


Figure A.11 Merit ratio versus number of stages for conditional branch resolution for *PBNT*, *TTCDB*, *TTDLB*, *TNTLB*, and *TNBTB* strategies.

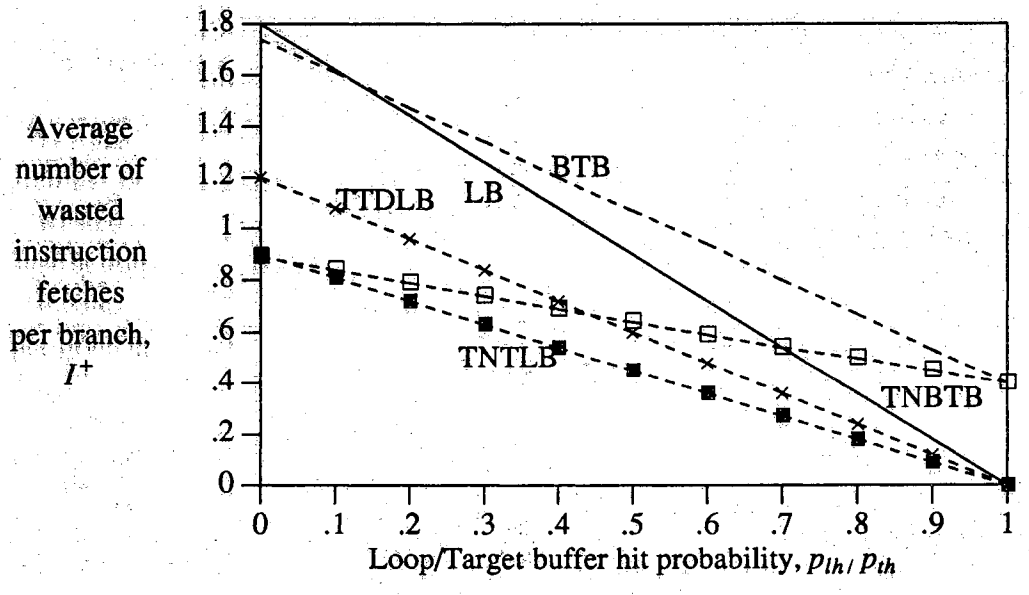


Figure A.12 Average number of wasted instruction fetches per branch versus Loop/Target buffer hit probability for *LB*, *BTB*, *TTDLB*, *TNTLB*, and *TNBTB* strategies.

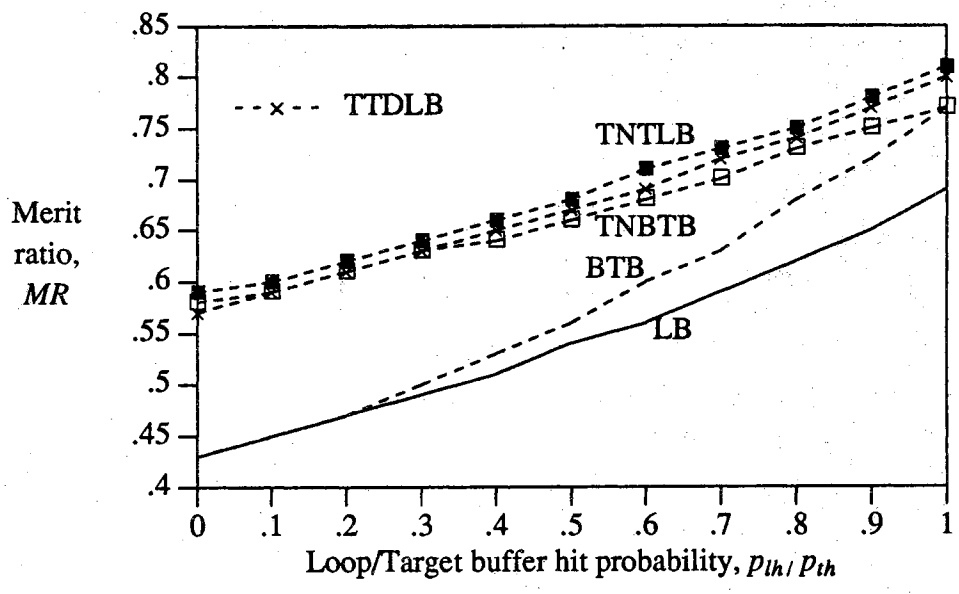


Figure A.13 Merit ratio versus Loop/Target buffer hit probability for *LB*, *BTB*, *TTDLB*, *TNTLB*, and *TNBTB* strategies.

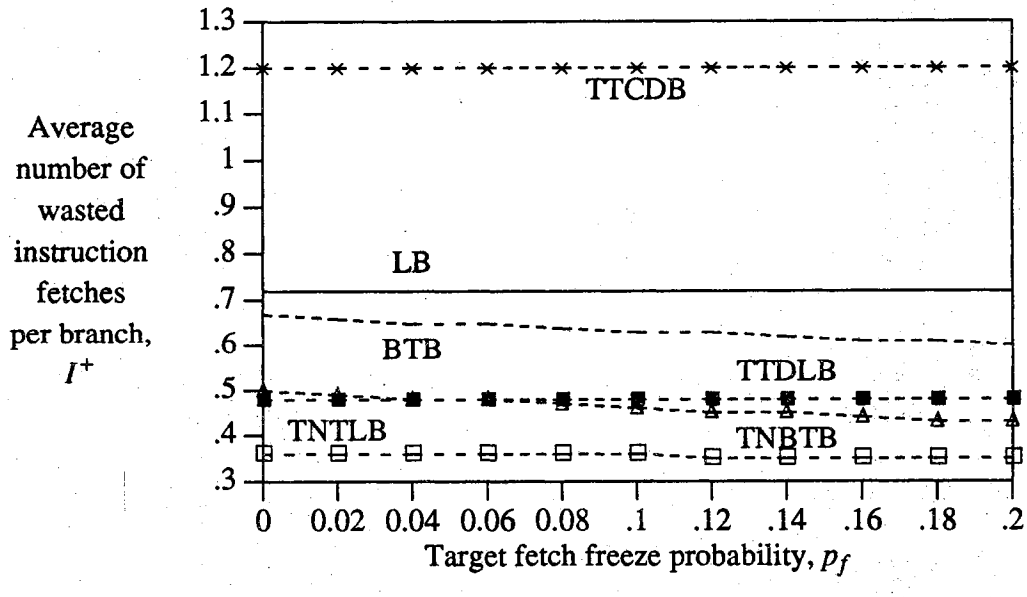


Figure A.14 Average number of wasted instruction fetches per branch versus target fetch freeze probability for *LB*, *BTB*, *TTCDB*, *TTDLB*, *TNTLB*, and *TNBTB* strategies.

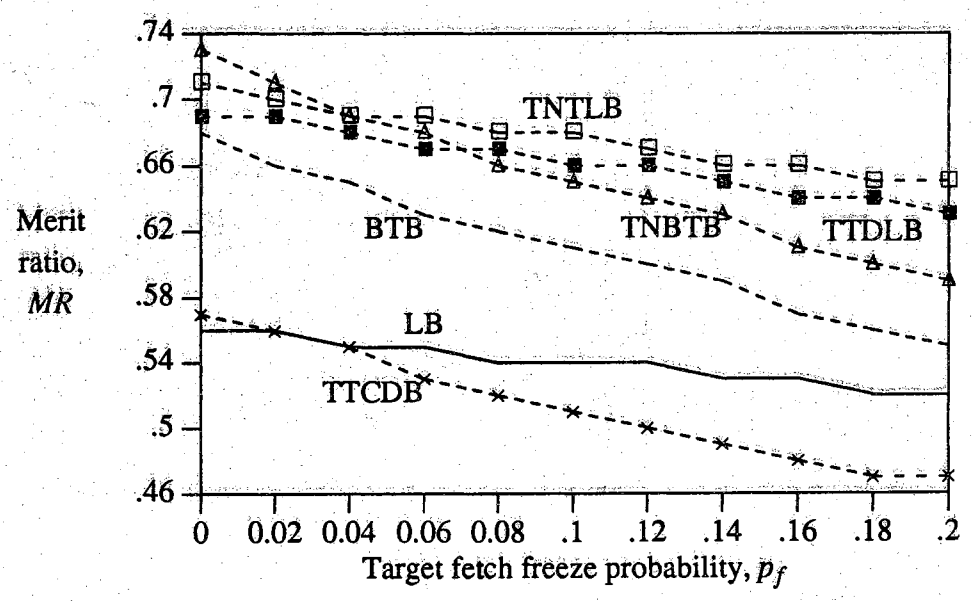


Figure A.15 Merit ratio versus target fetch freeze probability for *LB*, *BTB*, *TTCDB*, *TTDLB*, *TNTLB*, and *TNBTB* strategies.

Appendix B

Table B.1 Additional benchmarks used in this study.

Benchmark	Description
whetstone	Synthetic mix of floating-point and integer arithmetic
tomcatv	Vectorizable floating-point Fortran benchmark that does little I/O
appbt	Coupled partial differential equations
appsp	Coupled partial differential equations
buk	Bucket sort
adm	FFTs
dyfesm	ODE solvers, nonlinear algebraic systems and sparse linear systems solver
flo52	Multigrid schemes, ODE solvers
ocean	FFTs
qcd	Monte Carlo schemes
spec77	FFTs, rapid elliptic problem solvers
track	Convolution
trfd	Integral transforms

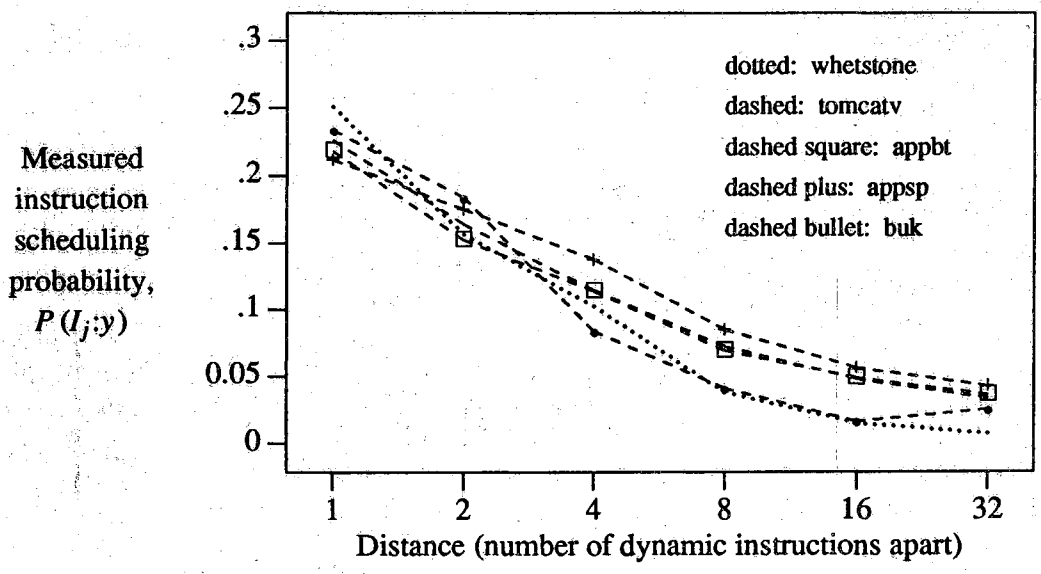


Figure B.1 Measured instruction scheduling probability versus distance for whetstone, tomcatv, appbt, appsp, and buk benchmarks.

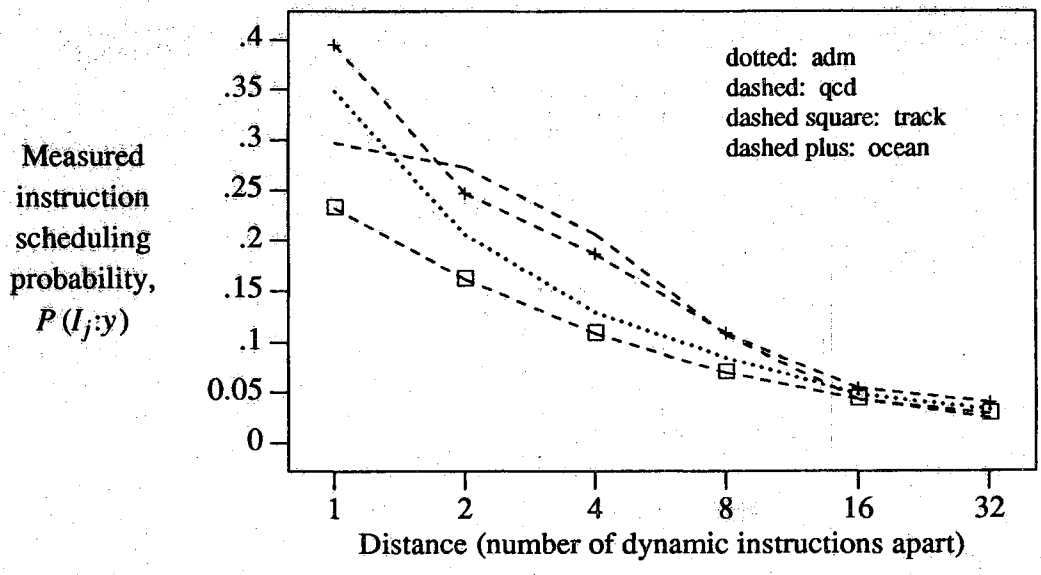


Figure B.2 Measured instruction scheduling probability versus distance for adm, qcd, track, and ocean benchmarks.

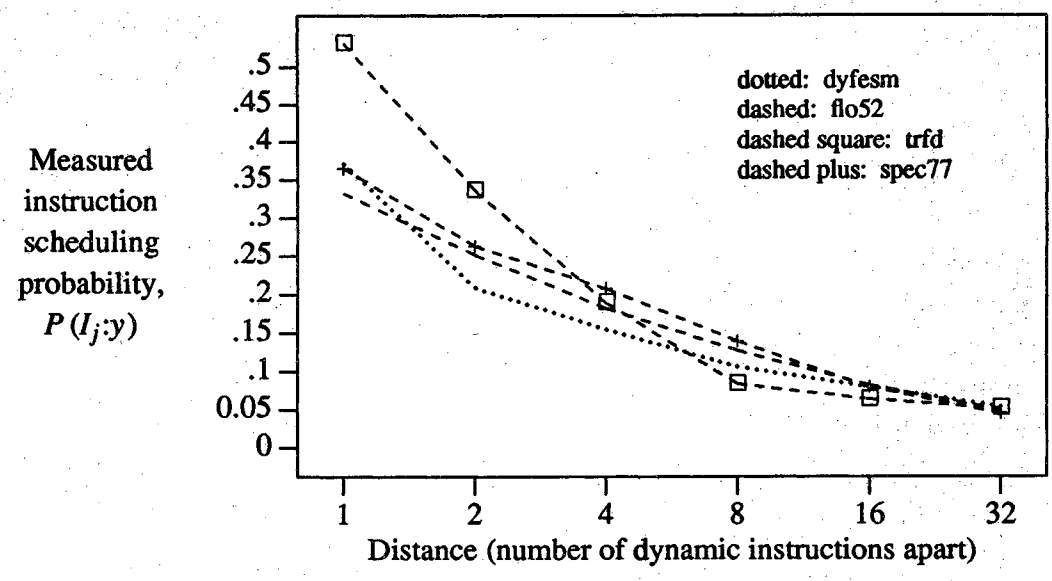


Figure B.3 Measured instruction scheduling probability versus distance for dyfesm, flo52, trfd, and spec77 benchmarks.

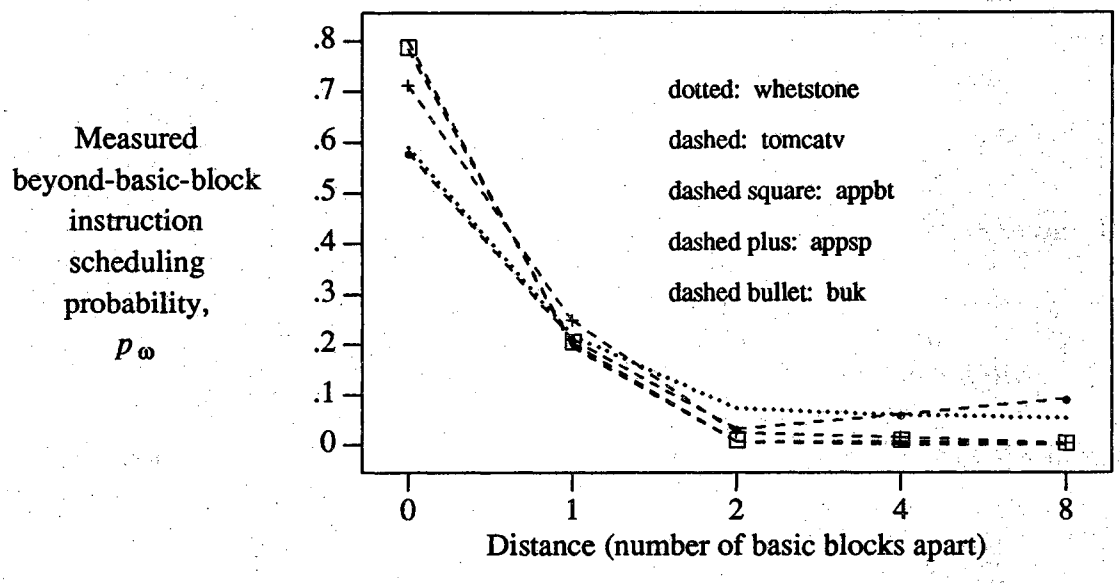


Figure B.4 Measured beyond-basic-block instruction scheduling probability versus distance for whetstone, tomcatv, appbt, appsp, and buk benchmarks.

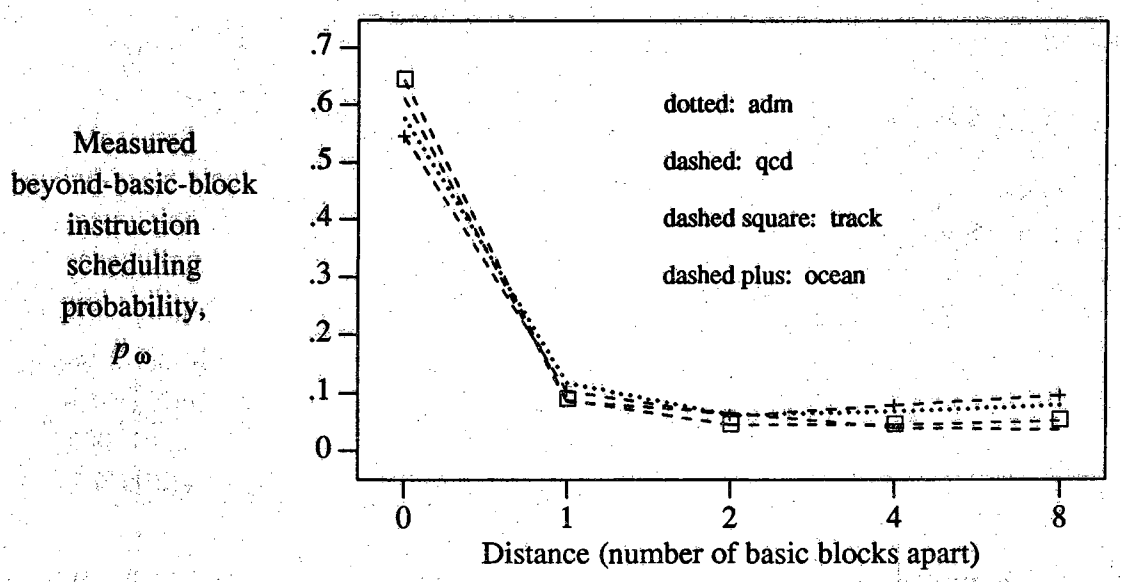


Figure B.5 Measured beyond-basic-block instruction scheduling probability versus distance for adm, qcd, track, and ocean benchmarks.

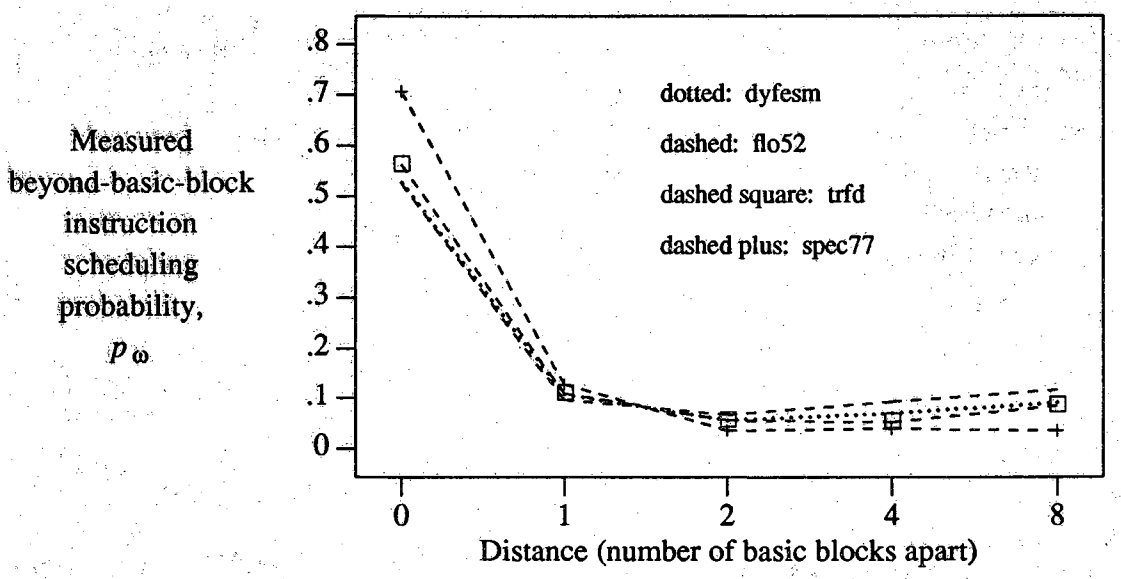


Figure B.6 Measured beyond-basic-block instruction scheduling probability versus distance for dyfesm, flo52, trfd, and spec77 benchmarks.

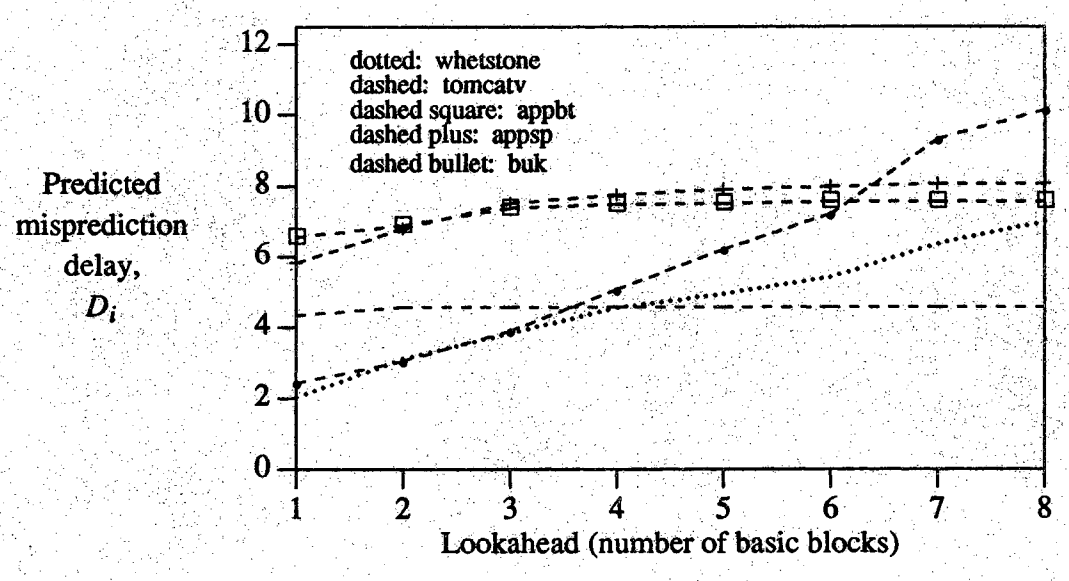


Figure B.7 Predicted misprediction delay based on the empirically collected p_ω distribution as a function of the amount of dynamic lookahead, in terms of number of basic blocks for whetstone, tomcatv, appbt, appsp, and buk benchmarks.

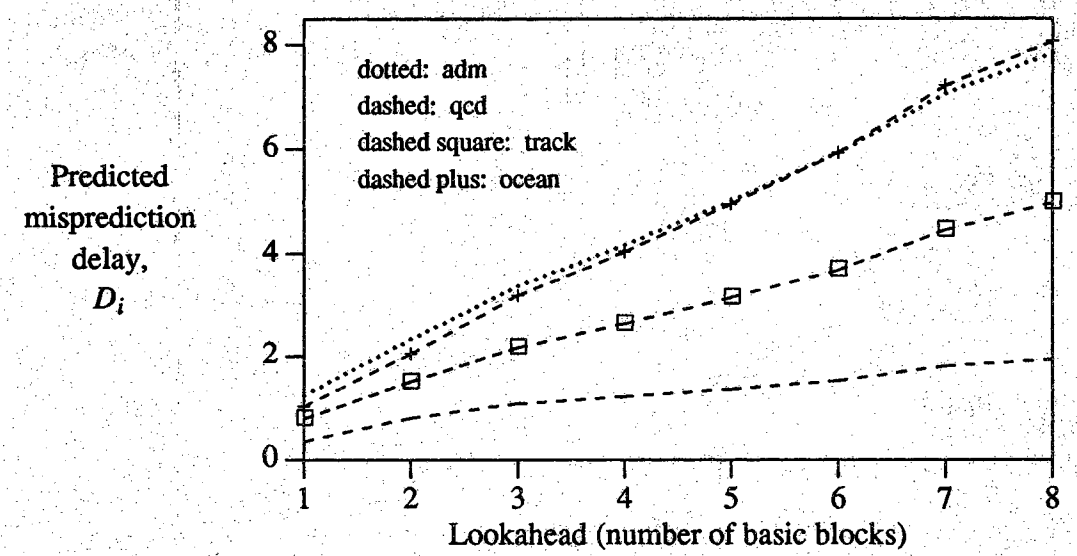


Figure B.8 Predicted misprediction delay based on the empirically collected p_ω distribution as a function of the amount of dynamic lookahead, in terms of number of basic blocks for adm, qcd, track, and ocean benchmarks.

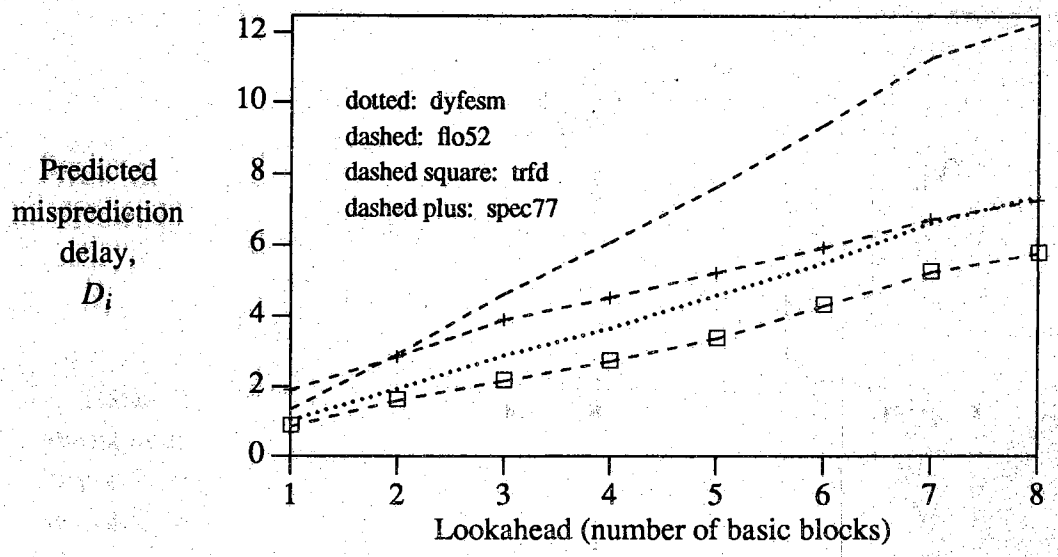


Figure B.9 Predicted misprediction delay based on the empirically collected p_ω distribution as a function of the amount of dynamic lookahead, in terms of number of basic blocks for dyfesm, flo52, trfd, and spec77 benchmarks.

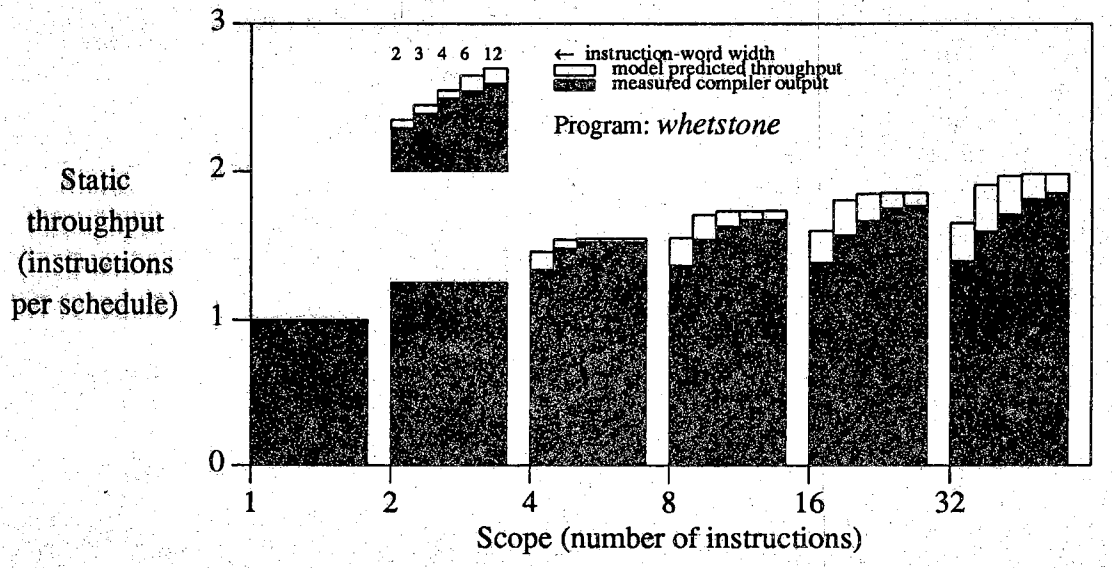


Figure B.10 Throughput under resource and scope constraints for the *whetstone* benchmark; resources varied with instruction word width = 2,3,4,6, and 12. The compiler output was influenced by resource constraints that are not part of the model.

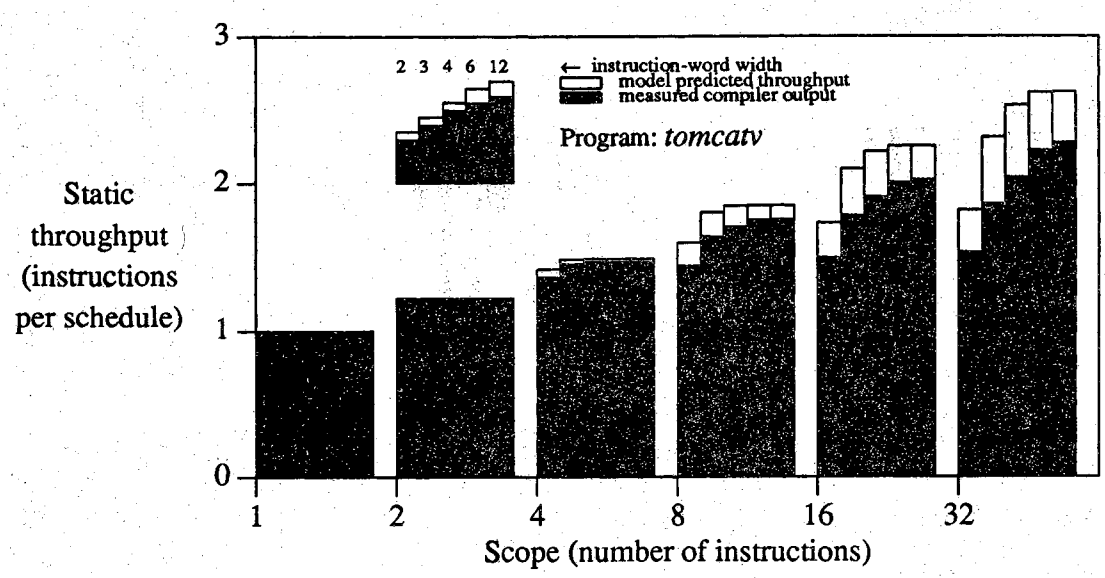


Figure B.11 Throughput under resource and scope constraints for the *tomcatv* benchmark; resources varied with instruction word width = 2,3,4,6, and 12. The compiler output was influenced by resource constraints that are not part of the model.

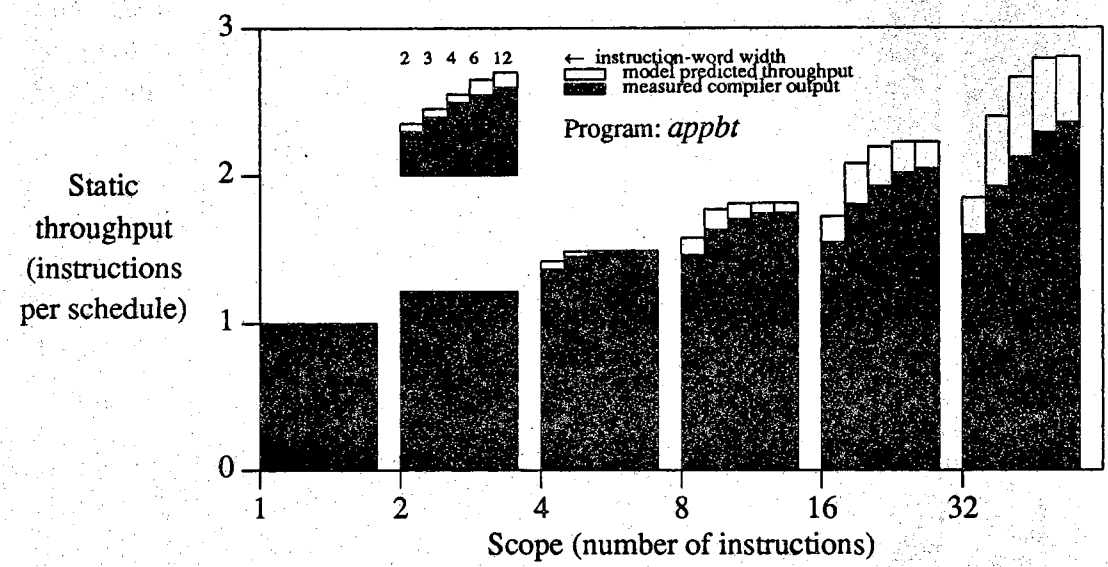


Figure B.12 Throughput under resource and scope constraints for the *appbt* benchmark; resources varied with instruction word width = 2,3,4,6, and 12. The compiler output was influenced by resource constraints that are not part of the model.

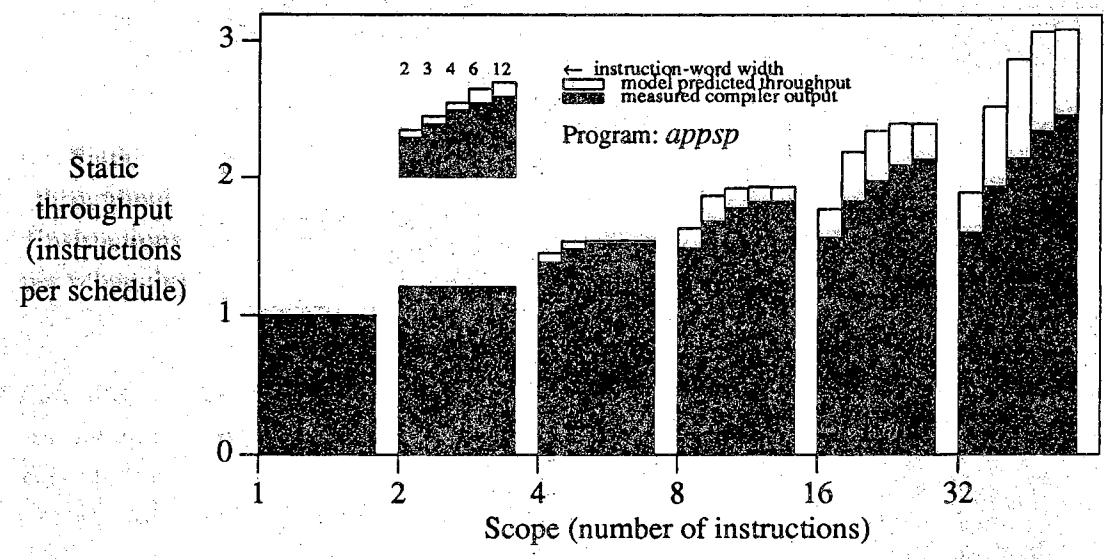


Figure B.13 Throughput under resource and scope constraints for the *appsp* benchmark; resources varied with instruction word width = 2,3,4,6, and 12. The compiler output was influenced by resource constraints that are not part of the model.

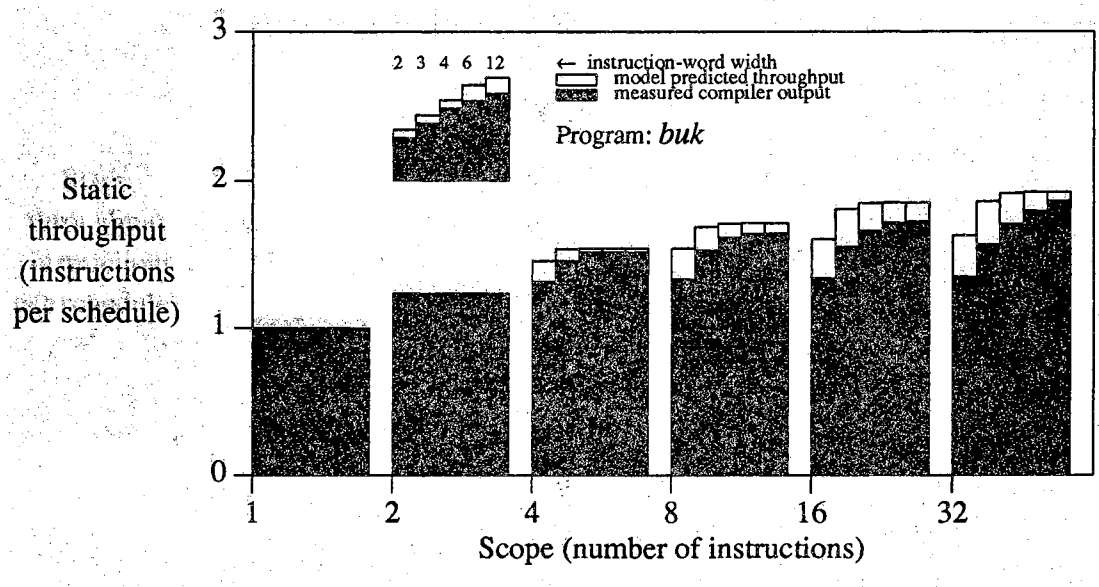


Figure B.14 Throughput under resource and scope constraints for the *buk* benchmark; resources varied with instruction word width = 2,3,4,6, and 12. The compiler output was influenced by resource constraints that are not part of the model.

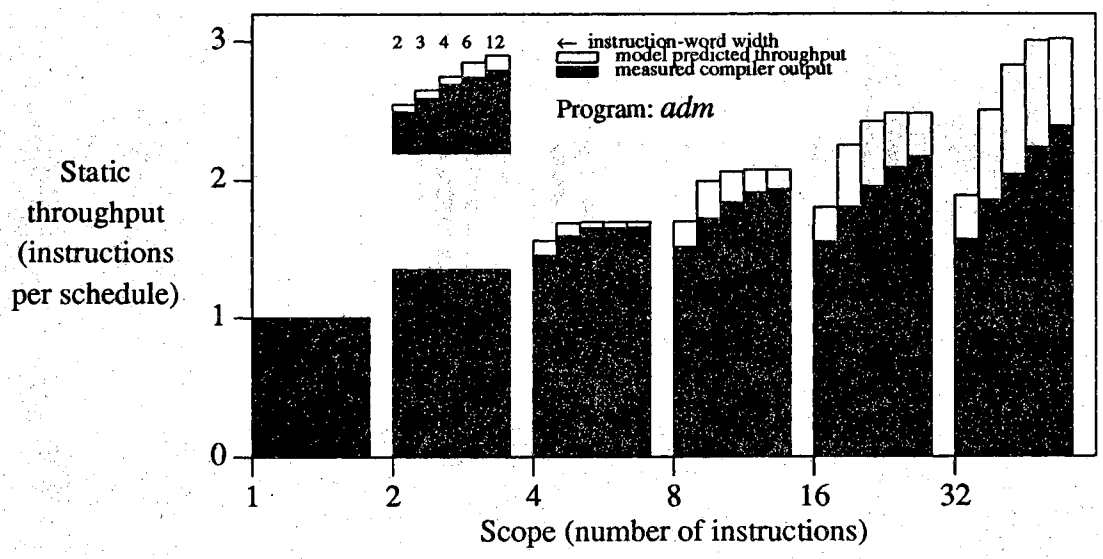


Figure B.15 Throughput under resource and scope constraints for the *adm* benchmark; resources varied with instruction word width = 2,3,4,6, and 12. The compiler output was influenced by resource constraints that are not part of the model.

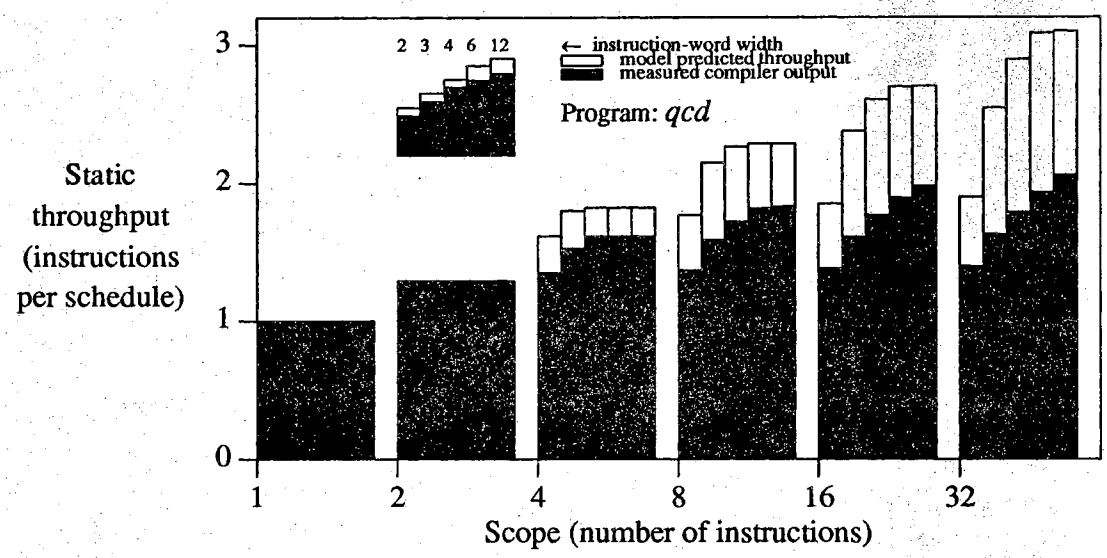


Figure B.16 Throughput under resource and scope constraints for the *qcd* benchmark; resources varied with instruction word width = 2,3,4,6, and 12. The compiler output was influenced by resource constraints that are not part of the model.

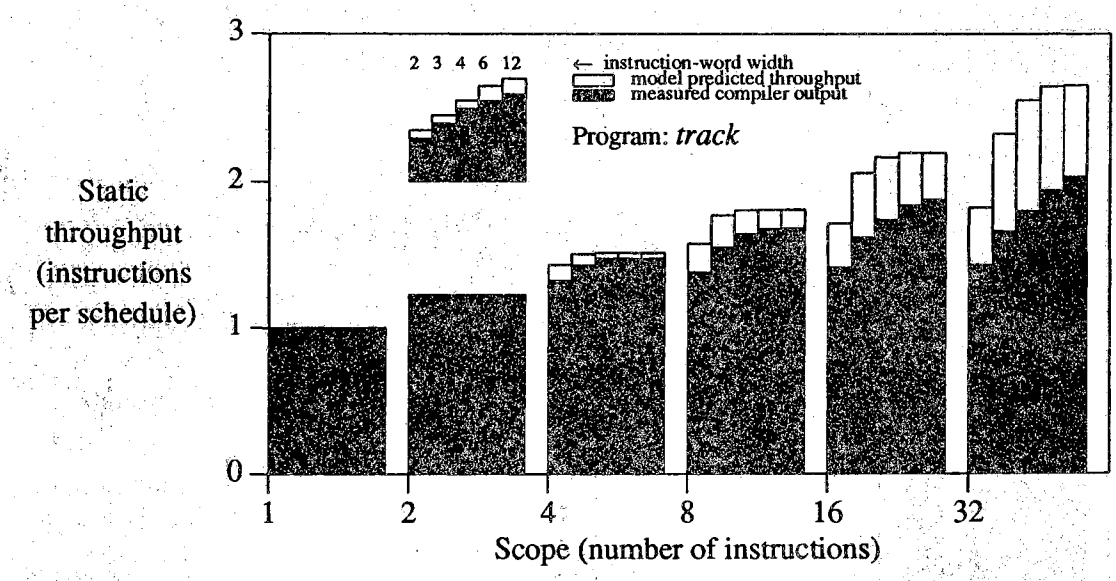


Figure B.17 Throughput under resource and scope constraints for the *track* benchmark; resources varied with instruction word width = 2,3,4,6, and 12. The compiler output was influenced by resource constraints that are not part of the model.

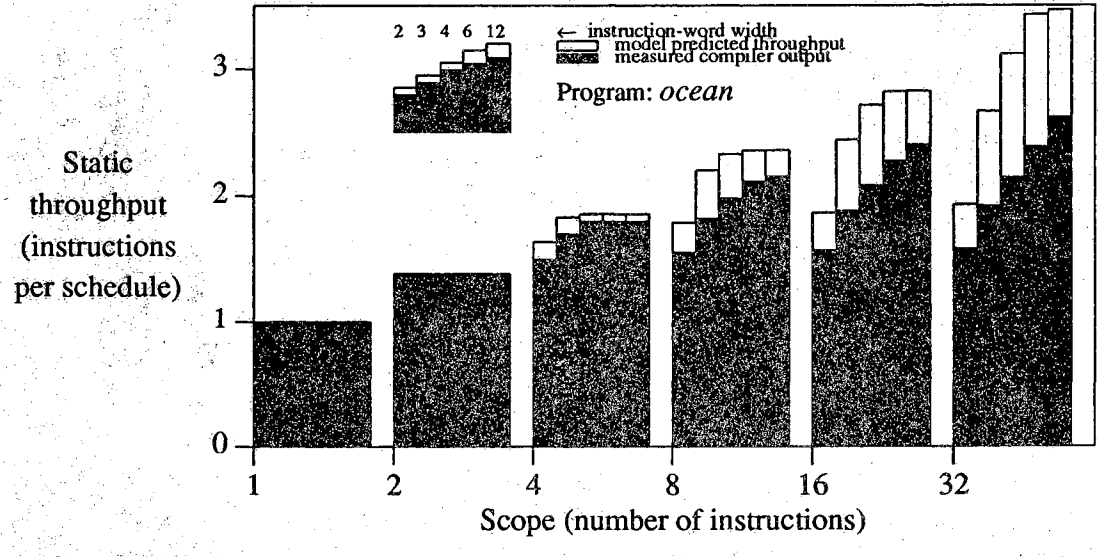


Figure B.18 Throughput under resource and scope constraints for the *ocean* benchmark; resources varied with instruction word width = 2,3,4,6, and 12. The compiler output was influenced by resource constraints that are not part of the model.

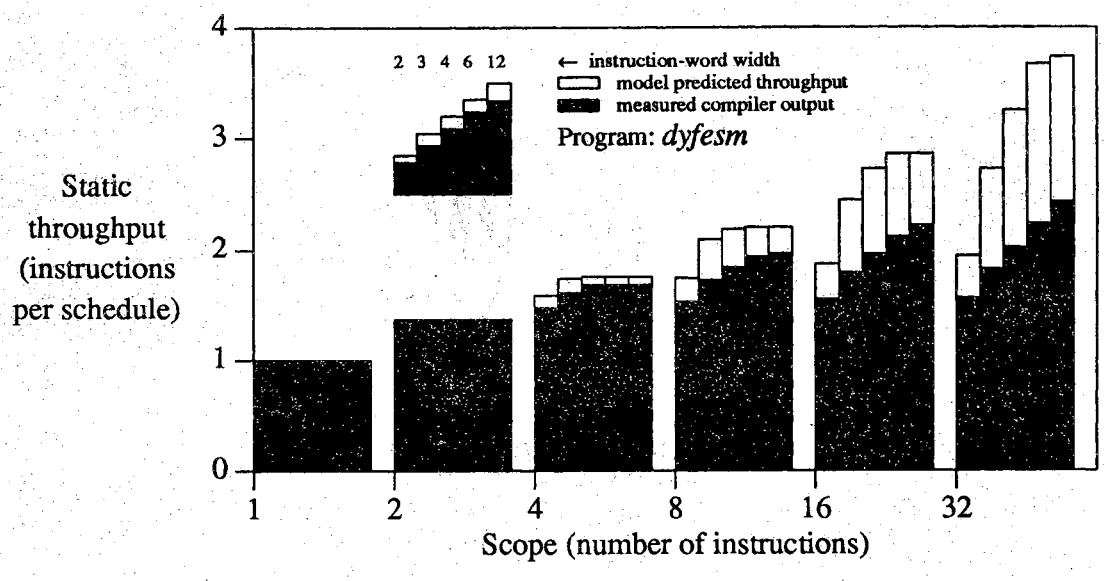


Figure B.19 Throughput under resource and scope constraints for the *dyfesm* benchmark; resources varied with instruction word width = 2,3,4,6, and 12. The compiler output was influenced by resource constraints that are not part of the model.

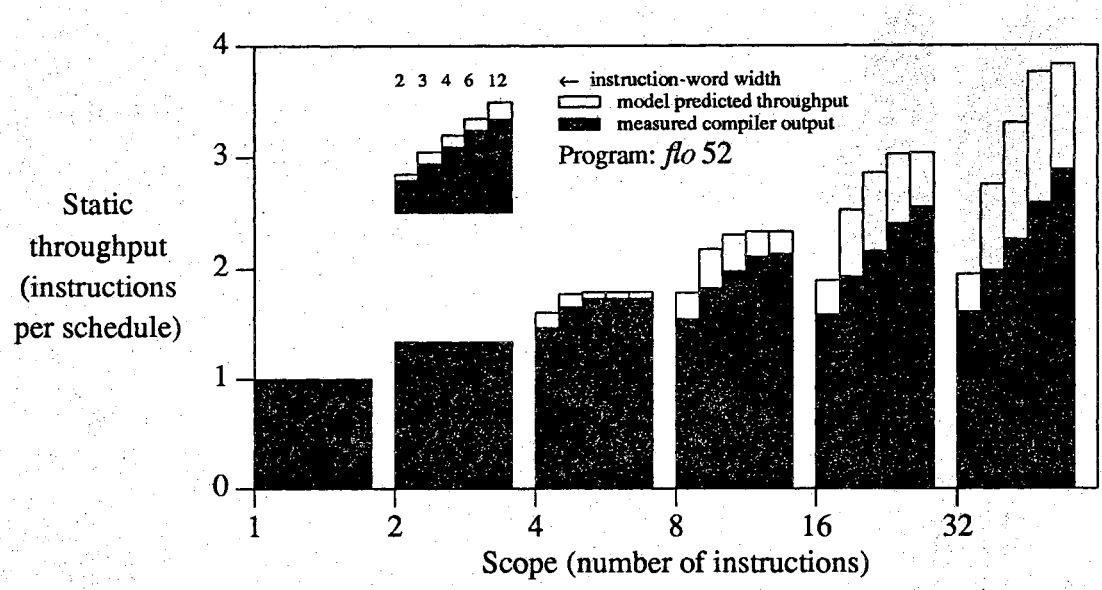


Figure B.20 Throughput under resource and scope constraints for the *flo52* benchmark; resources varied with instruction word width = 2,3,4,6, and 12. The compiler output was influenced by resource constraints that are not part of the model.

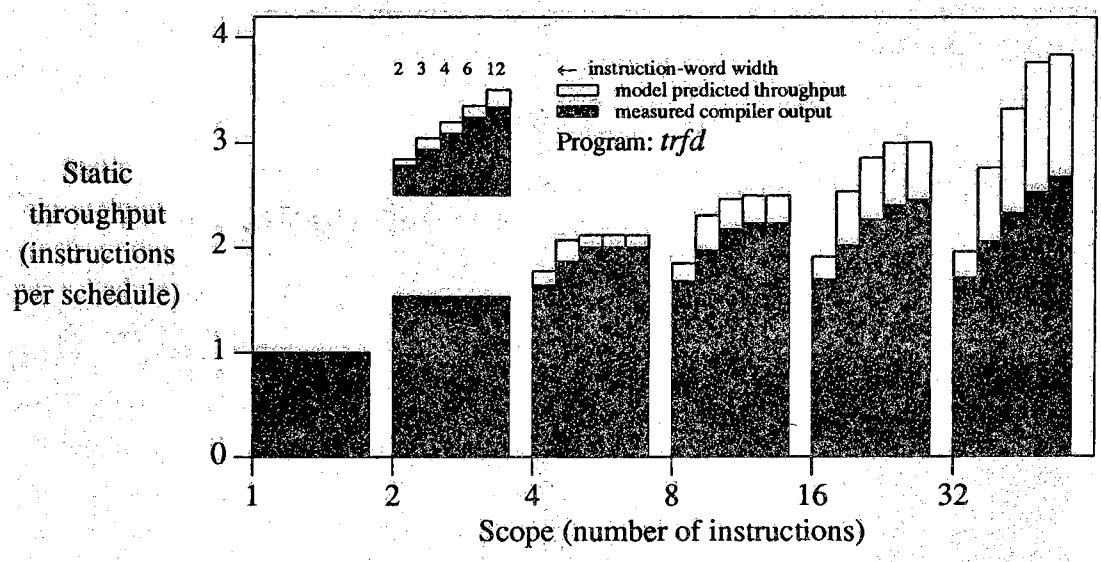


Figure B.21 Throughput under resource and scope constraints for the *trfd* benchmark; resources varied with instruction word width = 2,3,4,6, and 12. The compiler output was influenced by resource constraints that are not part of the model.

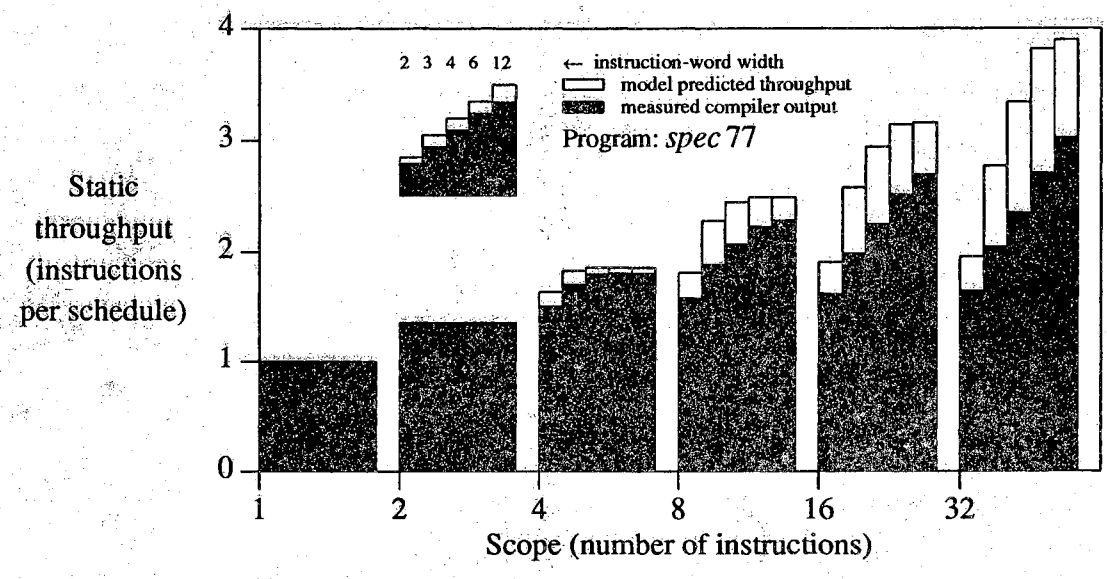


Figure B.22 Throughput under resource and scope constraints for the *spec77* benchmark; resources varied with instruction word width = 2,3,4,6, and 12. The compiler output was influenced by resource constraints that are not part of the model.