McGill University
School of Computer Science

On the Limits of Program Parallelism and its Smoothability

ACAPS Technical Memo 40
June 26, 1992

Kevin B. Theobald
Guang R. Gao
Laurie J. Hendren

Advanced Compilers, Architectures and Parallel Systems Group

Some recent studies of the so-called "limits on instruction parallelism" in application programs have reported limits that are surprisingly low (3-7 instructions) for well-known benchmark programs. In this paper, we report results of a new study of instruction-level parallelism and the smoothability of this parallelism. In addition to showing a strikingly high limit of parallelism for an oracle machine model, we also study the following new aspects of parallelism and smoothability.

Parallelism Limits: In addition to confirming some results recently reported (i.e. by Wilson and Lam [LW92]), our work also provides answers to the following important questions for architects and compiler writters which were left open:

- What are the most important characteristics of the oracle machine models?
- What happens if we allow each test program to run to full completion instead of stopping after a limited number of instructions?
- Do these results apply to other real programs (run to completion) in addition to the selected benchmark programs?
- How do various restrictions on the use and reuse of memory impact the potential parallelism?

Smoothability: In our study, smoothability is measured quantitatively and compared for a number of programs. Our results indicate that the parallelism obtained has a relatively smooth temporal profile, although different programs may have different smoothabilities. In particular, our results indicate that machine models with speculative multithreading exhibit impressive levels of instruction-level parallelism with high smoothability.

Keywords: Computer architecture modeling, Instruction-level parallelism, Parallel processing, Parallelism measurements, Smoothability.

Contents

1 Introduction **4**

2 Methodology and Models **5**

2.1 Our Trace Simulation Methodology 5

2.1.1 Memory Disambiguation 7

2.1.2 Register and Memory Renaming 7

2.1.3 Control Barrier Elimination 8

2.1.4 Finite Windows 9

2.1.5 Unrolling 10

2.2 Experimental Models Used 10

3 Smoothability **11**

4 Experimental Results **13**

4.1 Effects of Branch and Jump Prediction 15

4.2 Finite Scheduling Windows 16

4.3 Frugal Oracles 17

4.4 Unrolling Results 18

4.5 Smoothability Results 19

4.6 Compiler Optimization 20

5 Previous Studies **21**

6 Conclusions **22**

A Experimental Testbed **25**

List of Figures

1 Packing parallelizable instructions: An example 7
2 Standard smoothability profile 12
3 Standard processor utilization profile 13

List of Tables

1 Oracle and Machine Models 10
2 Benchmarks and test cases used 14
3 A summary of results. 15
4 The effects of finite window sizes 17
5 The effect of frugal use of memory 18
6 The effects of unrolling 19
7 The effect of finite processors 20

1 Introduction

At the current rate of progress in VLSI technology, it will be possible by the turn of the century to produce a single chip with tens to hundreds of millions of transistors. This level of integration can be used to implement a high-density, high-performance processor on a single chip, perhaps one based on advanced architectural ideas such as multithreaded architectures [A+90, ALKK90, KS88, NA89, HF88, DCC+87, ND91, WG89]. Such processors can then become the building blocks of an advanced, massively-parallel computer.

Those contemplating the design of these highly-parallel machines must ask two fundamental questions:

Question 1 (Limits of Parallelism): How much parallelism exists in the target applications?

Question 2 (Smoothness of Parallelism): Is the available parallelism smooth enough to be effectively exploited?

In order to justify having machines supporting extensive parallelism, there must be enough useful work to do at any point in time to keep most of the processors usefully busy. This means that not only must there be enough parallelism in the application to be exploited by these processors, but the parallelism must also be smoothable, to be fairly evenly distributed over time. Otherwise, Amdahl's Law will take effect and prevent the computer from achieving a good speed-up.

There have been many experiments, involving a wide variety of machine models and target applications, to measure the limits of parallelism that may be exploited in a program [JW89, LW92, BYP+91, Wal91]. Some have reported limits that are surprisingly low (3-7 instructions) for well-known benchmark programs even under their best machine model [Wal91]. In [LW92], it has been demonstrated that such limits can become substantially higher if more powerful machine models are considered which can exploit parallelism by eliminating constraints due to sequential control flow.

Most previous studies are interested in Question 1 above. In this paper, we are interested in answers to both Questions 1 and 2, and we report results of our study which are unique in the following aspects:

Limits of Parallelism: In answering Question 1, our results have shown a strikingly high limit of parallelism for oracle machine models. In addition to SPEC benchmark programs, we are also interested in a wider variety of real programs, and to that end we have added an interesting program for large-vocabulary speech recognition which we received from industry. For our experiments we allow each test program to run to its completion, in some cases the traces have lengths of over 3 billion instructions! This may turn out to be critical in providing measurement of smoothness over the entire execution of a program (See Section 4.6). We also study the impact of various restrictions on the use of memory to the potential parallelism. Thus, our results provide answers to some unanswered questions in previous studies.

Smoothability of Parallelism: We introduce *smoothability* as a measure to characterize instruction-level parallelism. Smoothability is measured quantitatively and compared for a number of programs. Our results indicate that the parallelism obtained has a relatively smooth temporal profile. In particular, under speculative multithreading, our collection of both scientific and non-scientific programs not only exhibit impressive instruction-level parallelism, but also exhibit a high degree of uniformity in the parallelism.

To search for the limits of parallelism and measure their smoothability under different architectural ideas, we have developed an experimental testbed. This testbed is based on a trace simulation method similar to previous experiments [AS92, LW92, Wal91]. The tool allows us to analyze the execution of real code and calculates, under various architecture models, how much parallelism could potentially be attained for a given model. Using this tool, we begin with a true "Omniscient Oracle" machine model and measure the idealized limit of parallelism. We then examine the degradation in performance as we selectively remove key features from the Omniscient Oracle and move it towards more realistic machine models at different levels. A total of 13 models are used in the study, including the oracle. Parallelism smoothness was measured for some models in order to study the "smoothability" aspect described later in the paper.

Our oracle machine models ignore control dependencies entirely (in this sense similar to the *oracle* proposed in [NF84]). While the results obtained from the oracle are an upper bound, and may be unrealistically high, we believe that they give a good indication of the parallelism that may be achieved with suitably-designed parallel algorithms.

It is not the goal of this research to design a machine which can execute the benchmarks in a manner corresponding to the analysis in our experiments. Rather, we are interested in studying the upper bounds of potential parallelism inherent in the application. Our objective in this paper is not to provide a recipe for the design of specific architectures and measure the possible parallelism achievable. Rather, we study what effects of the limitations of each architecture model make it *impossible* to achieve the parallelism limits.

In the next section, we describe how trace analysis works and develop the models used in our study. Smoothability is defined and discussed in Section 3. The experimental results and their ramifications are presented in 4.6. In Section 5 we compare our work to previous studies. The final section, Section 6, summarizes the paper, discusses the conclusions that may be drawn from this study, and describes how we plan to extend this analysis tool in the future. We briefly describe implementation of the tool in an appendix.

2 Methodology and Models

This section describes the trace simulation techniques and machine models used in our study. First we show conceptually how our trace analysis reveals parallelism in a program, and how various features added to a model affect potential parallelism. These effects are illustrated using a simple loop. Next, we present the 13 basic machine models used in this study. and discuss what the use of these models to measure parallelism can tell us about exploiting parallelism in real computers.

2.1 Our Trace Simulation Methodology

Trace analysis begins with a trace of the execution of a program. This trace consists of a stream of operations representing the actual order of instructions executed (*not* the static object code). For each executed operation, the trace gives the opcode, PC address, memory-access address (if any), and the destination of a branch or jump.

The analyzer reads the operations from the stream and packs them into *parallel instructions* (PI). As the analyzer reads each operation in the trace, it inserts the operation into the earliest PI possible,

while simultaneously respecting the dependencies between that operation and all previous operations. The following types of dependency between operations may exist:

- *Data dependency:* if an operation S_2 reads a storage location[1] which was most recently written by operation S_1, and if S_1 was packed into PI p, then S_2 can be scheduled no earlier than PI $p+1$.

- *Anti-dependency:* if an operation S_2 writes a storage location which was most recently read by operation S_1, and if S_1 was packed into PI p, then S_2 can be scheduled no earlier than PI p. (It is assumed that the write and read can occur simultaneously, and S_1 will read the proper value.)

- *Output dependency:* if an operation S_2 writes a storage location which was most recently written by operation S_1, and if S_1 was packed into PI p, then S_2 can be scheduled no earlier than PI $p+1$.

- *Control dependency:* if operation S_1 is the most recent conditional branch prior to a given operation S_2 in the trace, and if S_1 was packed into PI p, then S_2 can be scheduled no earlier than PI $p+1$.

In this study, we assume that all operations (including memory references) take a single cycle to execute. Memory is global with unlimited access, and there are no caches. In our 13 basic models, infinitely many functional units are available to execute all operations which can be packed into one PI (we limit the number of processors later when we study smoothability, discussed in the next section). In most cases, we also allow operations arbitrarily far apart in the dynamic trace to be packed into the same PI, so long as all other dependency constraints are obeyed. (Real machines may not be so generous, but this study aims to determine how much parallelism is truly inherent in a program, independent of any particular hardware limitations.)

To illustrate the basic principle, consider this simple loop fragment for multiplying two vectors componentwise:

```
S1:  ld [%2+%1],%f0
S2:  ld [%3+%1],%f1
S3:  fmuls %f0,%f1,%f2
S4:  st %f2,[%4+%1]
S5:  subcc %1,4,%1
S6:  bg S1
S7:  nop
```

A dynamic trace would have the sequence S1-S7 repeated many times. The left half of Figure 1 shows how our trace analyzer would pack these operations into PI's if all four dependency types listed above were obeyed. Arcs are drawn to show the data dependencies between operations. The arc with a small tick mark drawn from S4 to S5 represents an anti-dependency. The dashed line indicates a barrier caused by the conditional branch at S6, which prevents all future operations from being scheduled before that barrier. One iteration of the loop can be initiated every 4 cycles, so the parallelism is 1.5.

Other models, described later, relax the dependency constraints, or tighten the resource limitations. The following subsections describe how these constraints may be modified, through such means as register renaming and branch prediction, and the effects such changes will have on overall parallelism.

[1] A "storage location" means either a register or a memory cell.

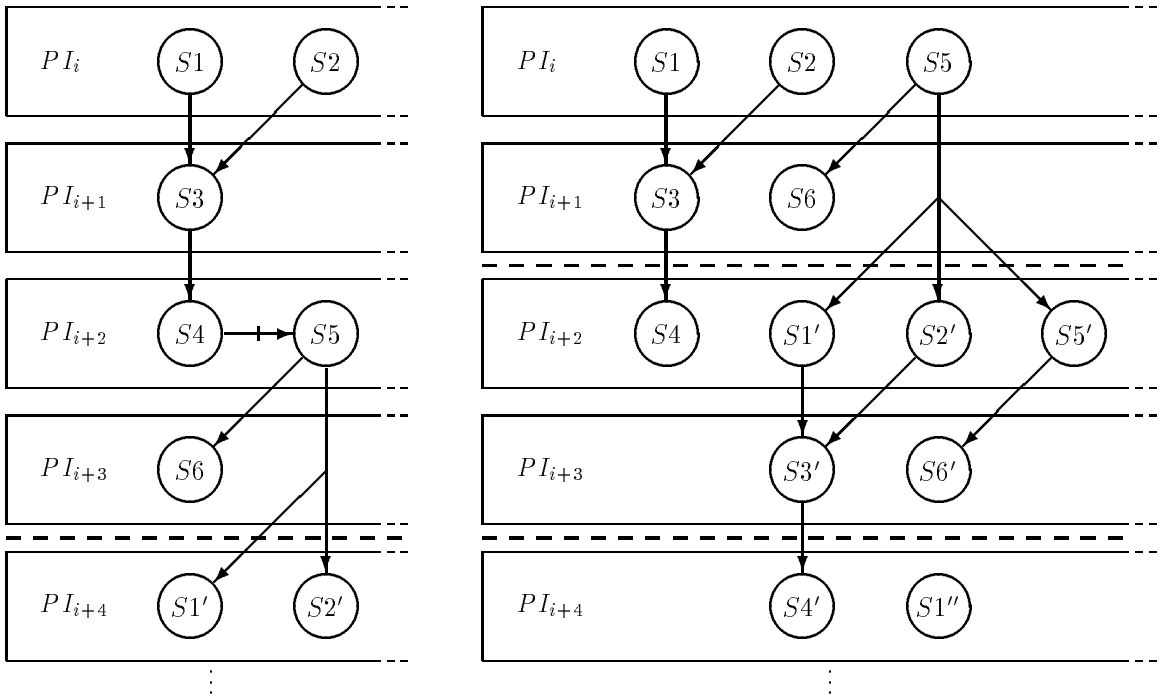

Figure 1: Packing parallelizable instructions: An example

2.1.1 Memory Disambiguation

Memory references can't always be determined at compile-time, due to indirect addressing. A conservative analysis would assume that any two memory references *could* refer to the same memory location, so that a dependency would exist between them. Thus, a conservative scheduler would have to assume that an anti-dependency might exist between S2 and S4, making it impossible to overlap separate iterations of the loop.

However, since the actual addresses of memory accesses appear in the trace, the analyzer can determine at run-time whether two memory references really conflict. This would model the potential effects of perfect compiler alias-analysis and/or special hardware to check for memory conflicts at run-time. It can also model an architecture with a more disciplined use of memory that makes such conflicts more explicit (hence easier to detect). Our analyzer supports this feature (run-time memory disambiguation) as an option.

2.1.2 Register and Memory Renaming

A false dependency (anti or output) exists between two operations when one must follow the other, *not* because the latter requires data produced by the former, but merely because the latter needs to reuse a storage location (register or memory cell) used by the former. An example of this in the code sample is the use of register %1 for the loop index, which cannot be updated (by S5) before being used to construct a memory address (by S4). Thus, overlapping the iterations of the loop body, as in software

pipelining, is impossible.

False dependencies also exist in references to main memory. This can limit parallelism in two ways:

1. Many sequential programs reuse data structures (such as arrays) to optimize the use of memory. Such optimization may be a good idea in programs with a single thread of control. In parallel machines, however, reusing memory in this way means that available processors cannot start operating on an updated version of a data structure while other processors continue to read the old version, because the update must wait until all processors are finished with the old copy.

2. Under the conventional stack architecture, procedure calls at the same level in a program reuse the same portion of the stack, leading to contention for that part of the stack. For instance, when executing object code corresponding to

   ```
   x = sin(y) + cos(z);
   ```

   the processor cannot push z onto the stack and call `cos` until it has popped the return value of `sin(y)` off the stack, even though these two function calls are independent and could run in parallel.

The inhibiting effects of false dependencies can be eliminated by ensuring that each register or memory location is written only once. In a real processor, false dependencies can be reduced by creating a sufficiently large register file, coupled with other techniques such as *register renaming*. The architecture provides additional registers that are not programmer-visible, and dynamically allocates them to store each new value generated. This will enforce a *single-assignment* rule and hence resolves the anti- and output-dependences.[2]

False dependencies in the heap can, in principle, be eliminated by adhering to the single-assignment rule, either explicitly in the programming language or by using *memory renaming*. Conventional architectures do not presently perform memory renaming, although dynamic dataflow machines with "colored tokens" employ a form of renaming.

False dependencies in the stack can be caused by the linear stack model, the standard runtime memory organization for supporting nested procedure invocations on sequential machines. Such false dependencies can be eliminated, for example, by organizing the memory frames for procedure invocation in a tree-like structure, as proposed in several multithreaded architecture models [CSS+91, NA89].

Infinite renaming, in which a register or memory location can be renamed any number of times, is equivalent to ignoring all false dependencies between objects of a particular type. For instance, if register renaming is applied to the dependence graph in Figure 1, then S5 can be executed in parallel with S1 and S2 of the same iteration, as the anti-dependency with S4 no longer exists. This moves operation S6 up as well, so the conditional-branch barrier has moved up by 2 PI's. If perfect alias analysis is also used, then the iteration issue rate has increased to once every other cycle, raising parallelism to 3, as the right half of Figure 1 shows.[3]

2.1.3 Control Barrier Elimination

[2] For example, The IBM RISC 6000 superscalar machine implements a form of register renaming in its floating point unit (FPU).

[3] This assumes the input and output arrays do not overlap. If memory disambiguation is not included in the model, then S1 and S2 cannot execute until S4 of the previous iteration has finished, because there *might* be a data dependence between them.

Previous parallelism experiments in which conditional branches were barriers to upward motion of operations generally had low parallelism, especially in non-numerical programs. This is because conditional branches occur quite frequently in most programs, which limits the search for parallelism to small "basic blocks" in the code. To get better parallelism results, one must consider models representing possible architectures which reduce the deleterious effects of conditional branches.

The *oracle*, introduced in [NF84], makes the most optimistic assumptions by ignoring control dependencies entirely. Our oracles, like most others, also ignore output and anti-dependencies, so that parallelism is only limited by true data dependencies in the program.

It is possible to use more conservative models between the extremes of having an oracle and forcing barriers at every conditional branch. One method is *branch prediction*, in which the processor tries to decide which branch is most likely to be taken, and takes that branch. When a predicted branch is taken, and it later turns out that that branch should not have been taken, then all instructions executed after the incorrect branch must be aborted and their side-effects reversed. There can be more branches in between the time when a mis-predicted branch is taken and the time at which this error is discovered, which means that at any time there may be multiple active machine states, only one of which is valid.

Branch prediction is modeled by the trace analyzer by ignoring barriers created by correctly-predicted branches, and keeping barriers created by incorrectly-predicted branches. Wall describes a method of dynamic branch prediction in which the prediction of which way to branch comes from a 2-bit count which is based on the history of previous executions of that branch. Jump prediction is done simply by predicting that the processor will jump to the previous jump destination. Branch prediction counts and jump prediction addresses are stored in tables hashed by the lower bits of the PC address of the branch or jump. In our experiments, the hash tables are large enough to give each PC address a unique table entry.

A more conservative assumption, which does not involve much speculation, is to assume a form of coarse-grain parallelism in which the barriers produced by branches in one procedure do not affect the scheduling of operations within other procedures. We call this feature *procedure separation*. Procedure separation allows procedures with no data dependencies to execute in parallel, which they probably would be unable to do otherwise if no branch prediction were used.

(Some previous studies [Kum88, LW92] extract more parallelism by analyzing the code to look for instructions that are really not control-dependent on one another, even though they are separated by conditional branches. This would allow, for instance, two consecutive but independent DO-loops to run concurrently. We have not implemented this feature at present.)

2.1.4 Finite Windows

So far, we have assumed that an operation from anywhere in the trace can be placed in the earliest PI possible, subject to data and control dependencies. In theory, the last operation in the sequential trace could be packed into the first PI. However, we may place limitations on how far apart two operations can be packed.

One possible limitation is to assume that the parallel machine can only look so far ahead of the program counter when looking for operations that are ready to be executed. With a limited window size, the analyzer conceptually keeps future operations from the trace stream in the window and packs them into PI's from the window. When the trace stream fills the window, the analyzer must "issue" the lowest-numbered unissued PI, thereby making it unavailable for further packing, and remove the operations in that PI from the window. This limitation was adopted by Wall, who used 2K-operation windows. This models how a real superscalar could schedule instructions at run-time, though in practice,

Model Name	Rename Regs.	Rename Memory	Branch Predict	Jump Predict	Proc. Sep.	Window Length	Unroll
Omniscient Oracle	yes	all	perfect	perfect	yes	∞	no
Unrolling Omni. Or.	yes	all	perfect	perfect	yes	∞	yes
Myopic Oracle	yes	all	perfect	perfect	yes	2048 pre	no
Short Oracle	yes	all	perfect	perfect	yes	2048 post	no
Tree Oracle	yes	stack	perfect	perfect	yes	∞	no
Linear Oracle	yes	heap	perfect	perfect	yes	∞	no
Frugal Oracle	yes	disambig.	perfect	perfect	yes	∞	no
Speculative Multithreaded	yes	all	table	table	yes	∞	no
Unrolling Spec. Mult.	yes	all	table	table	yes	∞	yes
Multithreaded	yes	all	none	none	yes	∞	no
Unrolling Multithreaded	yes	all	none	none	yes	∞	yes
Smart Superscalar	yes	all	none	none	no	∞	no
Stupid Superscalar	no	none	none	none	no	∞	no

Table 1: Oracle and Machine Models

the window size of current superscalars is very small.

Another limitation is to place a finite window *after* the scheduling. In this case, the analyzer keeps a finite queue of PI's available for packing. An arbitrary number of operations from the trace can be packed into these PI's, but if a new PI is initiated and this causes the PI's to exceed the window length, the analyzer must "issue" the lowest-numbered unissued PI, again making it unavailable for further packing.

2.1.5 Unrolling

Even when false dependencies and control dependencies are ignored, parallelism in loops may be restricted because there are true data dependencies between successive versions of the loop index variables. In other words, in a FOR loop, only one iteration of the loop can be initiated every cycle, since the index variable can only be incremented once per cycle.

It is conceivable that a processor could initiate all iterations of a loop simultaneously by generating the entire set of indexes simultaneously, provided they were related in a simple way, such as a constant difference. To measure how such a feature could affect parallelism, we provide an option which models this feature by eliminating the normal 1-cycle execution time of any operation that increments or decrements a register by a constant.

2.2 Experimental Models Used

We used 13 models in our study. We chose the models by deciding which architectural features we believe are important for a high-performance parallel architecture to have, and created models in which these features were adjusted, in order to measure how the given feature (or lack thereof) changed the amount of available parallelism. The models are summarized in Table 1.

In general, we chose a top-down approach. From the previous discussion, it is clear that the most

ideal oracle model, which we call the *Omniscient Oracle*, should have the following features:

- Infinite processors
- No barriers due to control dependences
- Perfect memory disambiguation
- Infinite register renaming
- Infinite memory renaming
- No window limitations

We then created models in which one or more features are restricted, to measure their importance.

Thus, five additional oracles, which retain the first four features in the list above, but are limited in some other way, were added. The *Myopic Oracle* and *Short Oracle* measure the effect of allowing only local parallelism to be used, by imposing a size limit of 2048 on one of the two scheduling windows (see Section 2.1.4). The Myopic Oracle has a 2K limit on the *pre-scheduling* window and the Short Oracle has a 2K limit on the *post-scheduling* window.

The Omniscient, Myopic, and Short Oracles have unlimited memory renaming. The other limited oracles measure what happens when memory renaming is selectively removed. The *Tree Oracle* allows renaming of stack variables, to measure the limits of parallelism exploitable by a machine using a tree of stacks or some equivalent implementation, but does not allow renaming in the heap. The *Linear Oracle* retains the linear stack model, by not allowing stack elements to be renamed, but allows renaming in the heap. The *Frugal Oracle* has no memory renaming.

The other machines do not have perfect branch prediction. The *Multithreaded Machine* has procedure separation, but no speculation. This machine is so named because its behavior models what might occur on an idealized multithreaded machine (with infinite renaming) executing a sequential benchmark without the algorithm modifications that would be necessary to achieve the oracles' levels of performance. The *Speculative Multithreaded Machine* has procedure separation, and also has branch and jump prediction capabilities, using a separate table for each, as described in Section 2.1.3. The *Smart Superscalar* has neither, but still has unlimited memory/register renaming and disambiguation. The *Stupid Superscalar* has none of these features.[4] The two superscalar machines can only find parallelism within basic blocks.

Finally, we have unrolling versions (see Section 2.1.5) of the Omniscient Oracle, Speculative Multithreaded Machine, and Multithreaded Machine.

3 Smoothability

As highlighted in the introduction, an important question to ask about programs with high degrees of parallelism is how evenly can this parallelism be distributed. If a program's parallelism is concentrated in short bursts of massive parallelism separated by long sequential sections, then Amdahl's Law will take over and the machine will need many more processors than the average to achieve the theoretical

[4] This model is similar to Wall's "Stupid" model [Wal91], except that Wall's model was limited to 64 processors, and could only schedule operations from within a window of 2K instructions. These limitations, however, have almost no impact on the Stupid model, since parallelism is limited to that within basic blocks.

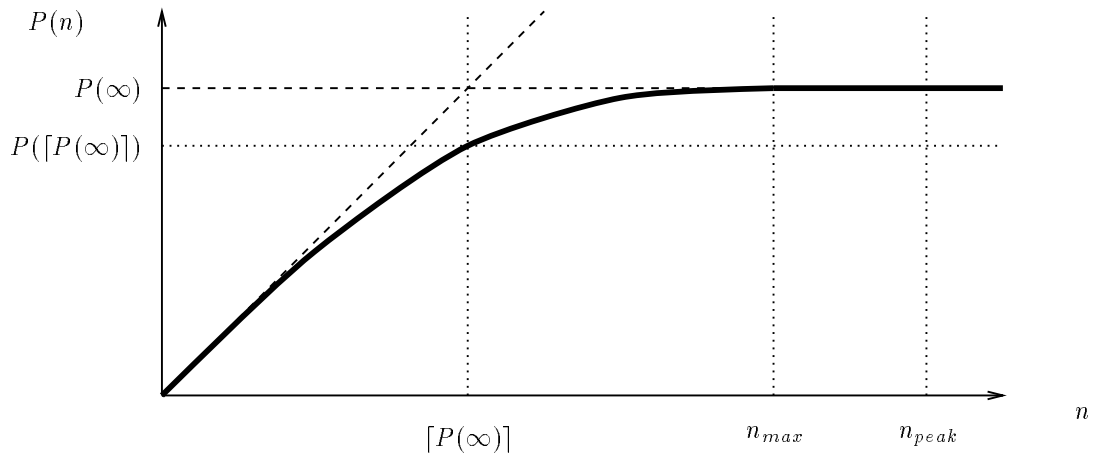

Figure 2: Standard smoothability profile

parallelism limits. Furthermore, the utilization of these processors is poor during the sequential sections of the execution. If the number of processors is limited, then some operations will be delayed, but this delay will add to the total execution time only if the delayed operations are in critical paths. In this section, we describe how our analyzer can measure this effect, and we define a value called *smoothability* which quantifies this property.

Our analyzer has the option of limiting the number of operations which can be packed into one PI. This limitation puts an upper bound on the maximum attainable parallelism, and may cause some operations to be delayed (scheduled into later PI's) because the PI's in which they first could be packed are already full. Because the analyzer's scheduler processes operations in the order in which they appear in the dynamic trace, it is always the later instructions which will be delayed.

For a given architectural model, and a given benchmark program, we can define $P(n)$, the "parallelism" function, which gives the average parallelism when the width of each PI is limited to n operations (i.e., there are n processors). We use $P(\infty)$ to denote the parallelism when the number of processors is infinite. A plot of $P(n)$ will look something like Figure 2.

This curve is bounded above by two factors. For small values of n, $P(n)$ is bounded by n, since we can't have greater than linear speedup. This is represented by the diagonal dashed line in Figure 2. Once n reaches $P(\infty)$, that value becomes the new bound, since $P(n)$ can never be greater than $P(\infty)$, as shown by the horizontal dashed line. $P(n)$ is guaranteed to reach $P(\infty)$ if n equals or exceeds n_{peak}, the largest number of operations packed into any single PI, i.e., the peak instantaneous parallelism, because above that point, all operations are executed immediately and there are no delays. Under the more optimistic models, n_{peak} may be quite high, since many operations not dependent on previous operations will be packed into the first PI. Usually, $P(n)$ will reach $P(\infty)$ at some lower value of n, called n_{max}. When n is between n_{max} and n_{peak}, some operations are delayed, but the delays do not affect critical paths, so the total number of PI's doesn't increase.

If a program is ideally (evenly) distributed over time, then the number of operations performed in every cycle is either $\lfloor P(\infty) \rfloor$ or $\lceil P(\infty) \rceil$ (since $P(\infty)$ may be non-integral, and each PI has an integral number of operations). Thus, $\lceil P(\infty) \rceil$ processors are sufficient to guarantee that every operation in the trace is executed as early as possible. A program that is not as smooth, however, will have (with infinite processors) some PI's with more than $\lceil P(\infty) \rceil$ operations. With only $\lceil P(\infty) \rceil$ processors, some of these operations will need to be deferred until a later PI in which not so many operations are packed. If the deferred operation is not on a critical path, and its results are not immediately needed, then this delay won't increase the total number of PI's needed to execute the program. However, if the operation is in

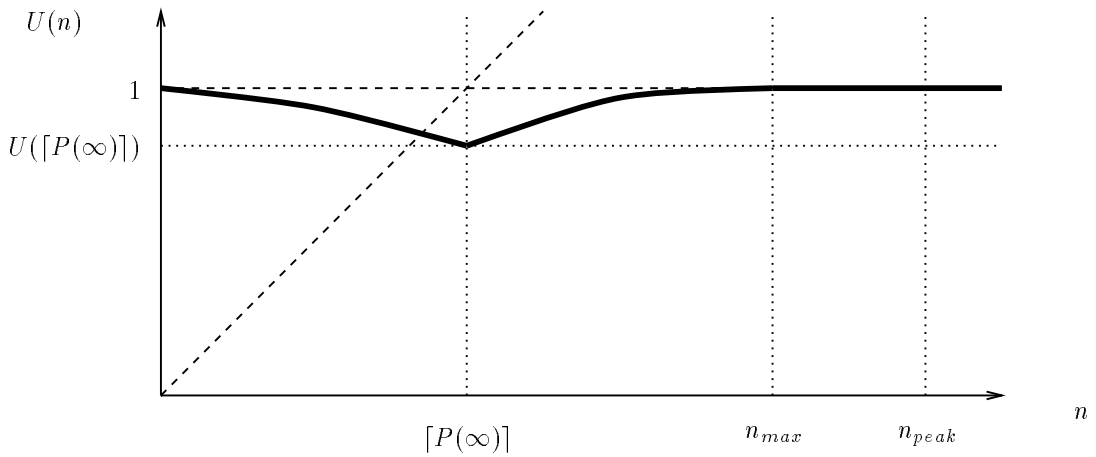

Figure 3: Standard processor utilization profile

a critical path, and it is delayed by k cycles, it is possible that every future operation dependent on that operation will be delayed by k cycles, increasing the total number of PI's by k.

Since $P(n)$ is bounded above in the manner previously described, we can define another function to represent how well the processors are utilized compared to the ideal case:

$$U(n) = \begin{cases} \frac{P(n)}{n} & \text{if } n \leq P(\infty) \\ \frac{P(n)}{P(\infty)} & \text{if } n > P(\infty) \end{cases}$$

A utilization curve corresponding to the curve in Figure 2 is shown in Figure 3.

We define *smoothability* to be the processor utilization at the critical value $n = \lceil P(\infty) \rceil$, i.e.,

$$S = U(\lceil P(\infty) \rceil) = \frac{P(\lceil P(\infty) \rceil)}{P(\infty)}$$

Our analyzer can keep a record of how many operations are in each PI, and can delay operations whenever a PI becomes full. To measure the smoothability for a given model, we first compute $P(\infty)$ for that model, round it to the next integer, and rerun the program with the resulting value as the PI width limit. We compute smoothability for both the Omniscient Oracle and the Speculative Multithreaded Machine. We also take parallelism profiles for both, and record the average number of cycles each operation must be deferred. This gives us an idea whether the smoothability is mostly due to a genuinely even distribution of operations over time, or to the ability to defer operations without excessively slowing down the execution of critical paths.

4 Experimental Results

To measure the parallelism available under the various models, a testbed was developed to implement the trace analyzer. Since programs can show significant differences in behavior when run on short inputs and on larger inputs, we required our testbed to handle traces for realistic input sizes. Also, we stipulated that all programs tested should run to completion, giving the analyzer enough opportunity to extract parallelism from all parts of the code. Implementation details are provided in Appendix A.

We decided to base the analysis on traces of programs executed on Sun SparcStations. The Sparc architecture was chosen for the testbed because it is representative of present-day RISC processors. Also, the Sparc processor has a special feature facilitating procedure calls, the partitioning of registers into sets which are selectively visible to the processor via "register windows." The mundane task of saving and restoring local registers and passing parameters is much more efficient. This avoids inflating the parallelism numbers with the pushes and pops which the analyzer would schedule in parallel. Finally, some of the recent trace-driven experiments have used MIPS workstations, and we decided to confirm the generality of the results by using a different RISC processor.

We ran our analysis tool on 9 benchmarks comprising a total of 20 test cases, under each of the 13 models. We chose problems that we felt are representative of the types of computations that are likely to be performed on future high-performance architectures. We looked for programs large enough to give figures suggestive of what can be achieved with "grand challenge" problems, without being so large as to overwhelm our analyzer.

The benchmarks consist of three regular FP-intensive scientific programs written in FORTRAN, one irregular FP-intensive FORTRAN program, and five symbolic applications written in C. All are from standard benchmark test suites, except for a speech-recognition program taken from an actual industrial application. The tests are summarized in Table 2.

Source	Program	Description	Test Case	Useful Operations	Call Dep.	% of ops. FP	Ld	St
DLX suite	`tex`	Text formatting	`man.tex (1 p.)`	15,184,459	19	.01	16	18
			`draft.tex (11)`	108,543,512	23	.04	15	8.0
Indust.	`speech`	Speech recognit.		551,267,528	12	4.5	14	2.8
SPEC test suite	`li`	Lisp interpreter	`(queens 7)`	204,921,097	75103	0	22	8.6
	`espresso`	Boolean minimization	`bca.in`	468,808,718	33	<.01	23	2.5
			`cps.in`	624,184,555	41	<.01	21	3.9
			`ti.in`	729,302,047	43	<.01	20	4.6
			`tial.in`	1,190,056,688	30	<.01	22	4.5
	`eqntott`	Truth-table gen.	`int_pri_3.eqn`	1,769,805,904	18	0	33	0.7
DLX suite	`spice`	Analog circuit simulation	`small`	26,026,482	17	10	30	9.9
			`large`	1,032,185,404	17	12	32	10
SPEC test suite	`tomcatv`	Mesh generation	N=33	46,850,826	17	16	46	13
			N=65	187,962,609	17	17	47	13
			N=129	751,099,835	17	17	47	13
			N=257	3,018,222,124	17	17	48	13
	`doduc`	Hydrocode simulation	`tiny`	103,371,553	14	14	36	10
			`small`	522,997,889	14	14	36	10
			`ref`	3,018,328,348	14	14	36	10
	`fpppp`	Quantum chemistry	NATOMS=4	276,896,598	14	19	43	10
			NATOMS=6	1,257,591,073	14	19	43	10

Table 2: Benchmarks and test cases used

In cases where benchmarks had more than one input case, we ran all available cases to see how parallelism might vary for different sizes and types of inputs. We used the standard input cases listed, except that we reduced the size of the Queens problem run on the Lisp interpreter from 9 to 7 queens, and the number of atoms in fpppp from 8 to 6.

One of the goals of our study was to analyze complete program executions, rather than stopping at some point as in some previous studies. Our experiments confirmed the wisdom of this choice, as

some of the programs showed long-term changes in parallelism over time. By programming our analyzer to print intermediate results at regular intervals, we found, for instance, that parallelism of `eqntott` under the Multithreaded model was over 6 after 100 million operations, but dropped below 2 after half a billion operations.

Each of the next three subsections discusses the observed effects of one type of architectural limitation on parallelism. This is followed by a presentation of the smoothability results, and a discussion of the role of compiler optimization.

4.1 Effects of Branch and Jump Prediction

Benchmark	*Omniscient Oracle*	*Speculative Multithreaded*	*Multithreaded*	*Smart Superscalar*	*Stupid Superscalar*
`tex` (man)	99.1	22.2	5.28	2.04	1.74
`tex` (draft)	192	12.2	3.04	2.08	1.66
`speech`	8,104	129	6.30	1.80	1.62
`li` (queens 7)	1,544	41.3	3.88	1.67	1.43
`espresso` (bca)	1,224	9.65	1.78	1.47	1.37
`espresso` (cps)	541	10.8	2.18	1.58	1.43
`espresso` (ti)	403	11.8	2.31	1.64	1.47
`espresso` (tial)	371	10.3	2.12	1.59	1.45
`eqntott`	43,298	300	1.66	1.46	1.43
`spice` (small)	76.5	27.2	8.18	3.53	2.00
`spice` (large)	62.8	22.4	6.11	3.90	2.04
`tomcatv` (N=33)	1,058	51.4	19.4	14.1	2.69
`tomcatv` (N=65)	2,108	52.6	20.1	18.3	2.76
`tomcatv` (N=129)	4,197	53.6	19.2	18.7	2.74
`tomcatv` (N=257)	8,418	54.2	19.2	19.1	2.74
`doduc` (tiny)	562	158	31.0	4.88	2.33
`doduc` (small)	617	174	32.8	4.92	2.33
`doduc` (ref)	621	202	33.2	4.92	2.33
`fpppp` (NATOMS=4)	4,977	297	52.7	30.1	2.88
`fpppp` (NATOMS=6)	11,296	311	54.1	30.9	2.88

Table 3: A summary of results.

A summary of the parallelism results for 5 base models is given in Table 3. Examining this table, we note that the results vary from surprisingly high parallelism results for the Omniscient Oracle to the expectedly poor parallelism results for the stupid superscalar model. In the case of the Omniscient Oracle, we find impressive parallelism for `speech` (8,104), `eqntott` (43,298), `tomcatv` (1,058 – 8,418), and `fpppp` (4,977 – 11,296). In addition, all other programs but `spice` and `tex` show parallelism results greater than 300.

A comparison of the first four models shows the importance of branch and jump prediction. The Speculative Multithreaded machine is just like the Omniscient Oracle, except it does not know the outcomes of branches and indirect jumps. Therefore, it must predict them, and sometimes makes mistakes. The Multithreaded model is a further restriction that has no jump or branch prediction. It does, however, allow instructions separated by a conditional branch to execute concurrently if they are

in different procedure instances. The Smart Superscalar doesn't even make this allowance.

From Table 3 we can make the following observations:

- For some programs like `li`, `espresso`, `eqntott`, and `tomcatv`, the power to predict jumps and branches perfectly is extremely important. As expected, for scientific programs that have regular, data-independent behavior like `doduc`, branch predication is much less important.

- There is a substantial parallelism loss between the Speculative Multithreaded model and the Multithreaded model. This result indicates that for programs with irregular control flow, speculative execution may be very important.

4.2 Finite Scheduling Windows

In order to study what characteristics of the Omniscient Oracle give such good parallelism results, we performed a series of experiments where we constrained other selected features of the oracle. In Table 4, we compare the parallelism results of the Omniscient Oracle and Myopic Oracle. These two oracles are identical, except the Myopic Oracle schedules operations from a window whose maximum length is 2048. The comparison is quite startling, and leads to the following observations:

- For all programs but `spice`, the Myopic Oracle performs significantly worse. In many cases, part of the loss is due to the fact that the size of the window (2,048) is an upper bound on the maximum parallelism. However, in no case did the parallelism on the Myopic Oracle even come *close* to this bound.

- For symbolic programs like `eqntott` and `speech`, and numeric programs like `fpppp`, the Myopic Oracle gives less than 2% of the Omniscient Oracle's parallelism. Thus, we can conclude that parallel operations in these programs may be separated by large distances in the trace.

- For most applications, the Myopic Oracle's performance on a given program declined as the problem size increased. As the problem size increases, many loops become longer, pushing parallel operations further apart. This effect is most noticable with `tomcatv` and `fpppp`.

We tried another variation of a fixed-window oracle, by placing the 2K-window *after* the operation scheduler. The Short Oracle can see operations across an infinite distance, but can only reach a short distance when scheduling them. This removes the 2K upper bound on parallelism, and the Short Oracle will always perform at least as well as the Myopic Oracle. However, this model still shows the extent to which concurrent operations are separated in the instruction trace. We found:

- For all numerical programs except `fpppp`, and for `spice`, the finite window limit has little or no impact on parallelism. This suggests that most of the parallelism comes from loops, which can't be parallelized so effectively if the window comes *before* the scheduler.

- For symbolic programs like `speech`, `li` and `eqntott`, the Short Oracle is almost as bad as the Myopic Oracle, suggesting that most parallelism comes from widely-separated locations in the program, and not from within loops.

Benchmark	Omniscient Oracle	Myopic Oracle	% of Omni	Short Oracle	% of Omni
`tex` (man)	99.1	34.0	34.3	39.4	39.8
`tex` (draft)	192	62.1	32.3	123	63.8
`speech`	8,104	90.5	1.12	146	1.80
`li` (queens 7)	1,544	64.0	4.14	92.6	6.00
`espresso` (bca)	1,224	35.1	2.86	103	8.42
`espresso` (cps)	541	59.6	11.0	312	57.7
`espresso` (ti)	403	66.7	16.5	317	78.6
`espresso` (tial)	371	55.3	14.9	256	68.9
`eqntott`	43,298	141	0.32	1,314	3.03
`spice` (small)	76.5	40.1	52.4	74.0	96.8
`spice` (large)	62.8	28.8	45.8	61.1	97.3
`tomcatv` (N=33)	1,058	159	15.0	1,058	100
`tomcatv` (N=65)	2,108	125	5.93	2,108	100
`tomcatv` (N=129)	4,197	101	2.40	4,197	100
`tomcatv` (N=257)	8,418	86.4	1.03	8,418	100
`doduc` (tiny)	562	103	18.3	550	98.0
`doduc` (small)	617	104	16.9	614	99.6
`doduc` (ref)	621	104	16.7	621	99.9
`fpppp` (NATOMS=4)	4,977	75.6	1.52	3,455	69.4
`fpppp` (NATOMS=6)	11,296	75.8	0.67	4,976	44.0

Table 4: The effects of finite window sizes

4.3 Frugal Oracles

Having established that window limitations are usually bad, we next experimented with frugal oracles. The results of these experiments are given in Table 5. In this table we give the parallelism measurements for the Tree Oracle (stack renaming, but no heap renaming), the Linear Oracle (heap renaming, but no stack renaming), and the truly Frugal Oracle (no stack or heap renaming). For each result, we give the percentage of parallelism relative to the Omniscient Oracle. Studying these results, we can make the following observations:

- Stack renaming is very important for numeric programs, and for symbolic programs like `speech` and `eqntott`. This is probably due to overlapping of successive iterations that call the same procedure in numeric programs, and concurrent execution of separate procedure calls in symbolic programs.

- Heap renaming is extremely important for `li`, `speech`, and `eqntott`, irrelevant to `fpppp`, and almost irrelevant to `doduc`. In the case of the symbolic programs, this reflects the fact that the programs are computing with large dynamically-allocated data structures. `espresso` and `tomcatv` show variations in performance with problem size, but in opposite directions.

- Every program shows marked loss of parallelism with the Frugal Oracle. Thus, it appears that renaming of both stack-allocated data structures and heap-allocated data structures is an important characteristic of the Omniscient Oracle. Usually, one or the other is the limiting case for a given application. In all cases, the Frugal Oracle performs as well or almost as well as the worst of the Tree Oracle and Linear Oracle.

Benchmark	Omniscient Oracle	Tree Oracle	% of Omni	Linear Oracle	% of Omni	Frugal Oracle	% of Omni
`tex` (man)	99.1	32.1	32.4	79.2	79.9	32.0	32.3
`tex` (draft)	192	75.8	39.4	167	86.9	71.5	37.2
`speech`	8,104	136	1.68	57.1	0.70	53.6	0.66
`li` (queens 7)	1,544	58.5	3.79	336	21.8	58.3	3.78
`espresso` (bca)	1,224	126	10.3	906	74.0	124	10.1
`espresso` (cps)	541	86.9	16.1	523	96.6	86.9	16.0
`espresso` (ti)	403	84.5	21.0	403	100	84.5	21.0
`espresso` (tial)	371	138	37.1	370	99.8	138	37.1
`eqntott`	43,298	1,742	4.02	1,314	3.03	1,314	3.03
`spice` (small)	76.5	58.8	76.9	27.6	36.1	25.6	33.5
`spice` (large)	62.8	36.0	57.3	20.1	32.0	19.7	31.4
`tomcatv` (N=33)	1,058	423	40.0	161	15.2	161	15.2
`tomcatv` (N=65)	2,108	443	21.0	157	7.45	157	7.45
`tomcatv` (N=129)	4,197	451	10.7	155	3.69	155	3.69
`tomcatv` (N=257)	8,418	457	5.43	155	1.84	155	1.84
`doduc` (tiny)	562	554	98.6	25.8	4.59	25.8	4.58
`doduc` (small)	617	615	99.7	25.0	4.06	25.0	4.06
`doduc` (ref)	621	621	99.9	25.0	4.02	25.0	4.02
`fpppp` (NATOMS=4)	4,977	4,977	100	71.0	1.43	70.9	1.43
`fpppp` (NATOMS=6)	11,296	11,296	100	72.8	0.65	72.8	0.64

Table 5: The effect of frugal use of memory

Our studies show, however, that for irregular code, memory renaming by itself produces little benefit. Our experiments showed little difference between the Smart Superscalar and the Stupid Superscalar for such applications (refer to Table 3). Thus, to achieve further improvements, we must go to more aggressive models, like the Multithreading and Speculative Multithreading models.

4.4 Unrolling Results

We performed a final set of experiments to see how more parallelism could be extracted from programs through static unrolling, as described in Section 2.1.5. We reran the programs on the Omniscient Oracle, Speculative Multithreading Machine, and Multithreading Machine, allowing the instructions that add constants to registers to execute instantaneously rather than taking the usual one-cycle delay. The results are in Table 6. From this we can see

- With the exception of `speech`, unrolling had little impact on parallelism under the Omniscient Oracle model. While statically unrolling a loop may allow all iterations to commence simultaneously, this apparently doesn't decrease the lengths of critical paths very much.
- Unrolling wasn't a much greater benefit to the Multithreaded Machine. Only `tomcatv` had more than a 7% improvement.
- Unrolling was a significant benefit to the Speculative Multithreading Machine. Many programs saw improvements of one-third, and `tomcatv` programs nearly doubled in speed. This is because the elimination of the one-cycle delay between successive loop iterations shortens the total time

needed to execute an entire loop. Since the last iteration of a loop is typically followed by a mispredicted branch, which creates a barrier for all succeeding operations inside the procedure containing the loop, packing the loop in fewer PI's reduces the total number of PI's needed for the whole procedure.

Benchmark	*Omniscient Oracle*			*Speculative Multithr.*			*Multithreaded*		
	Orig.	Unr.	% Inc.	Orig.	Unr.	% Inc.	Orig.	Unr.	% Inc.
`tex` (man)	99.1	101	1.47	22.2	24.2	9.19	5.28	5.55	5.21
`tex` (draft)	192	200	3.90	12.2	12.6	3.36	3.04	3.10	1.91
`speech`	8,104	12,225	50.8	129	130	1.34	6.30	6.48	2.73
`li` (queens 7)	1,544	1,566	1.43	41.3	46.3	12.0	3.88	3.97	2.24
`espresso` (bca)	1,224	1,224	0.01	9.65	11.5	21.9	1.78	1.85	3.82
`espresso` (cps)	541	542	0.08	10.8	13.8	28.0	2.18	2.32	6.70
`espresso` (ti)	403	403	<0.00	11.8	15.8	33.9	2.31	2.48	7.09
`espresso` (tial)	371	371	0.04	10.3	11.7	13.7	2.12	2.20	3.44
`eqntott`	43,298	43,298	<0.01	300	388	29.4	1.66	1.66	0.18
`spice` (small)	76.5	78.7	2.90	27.2	28.1	3.62	8.18	8.46	3.39
`spice` (large)	62.8	64.0	1.86	22.4	23.1	3.07	6.11	6.23	1.91
`tomcatv` (N=33)	1,058	1,059	0.07	51.4	96.1	86.9	19.4	23.7	22.7
`tomcatv` (N=65)	2,108	2,109	0.07	52.6	104	98.1	20.1	24.5	21.9
`tomcatv` (N=129)	4,197	4,200	0.07	53.6	111	107	19.2	23.6	23.0
`tomcatv` (N=257)	8,418	8,424	0.07	54.2	114	111	19.2	23.7	23.0
`doduc` (tiny)	562	563	0.28	158	168	6.28	31.0	32.0	3.30
`doduc` (small)	617	619	0.32	174	186	6.65	32.8	33.9	3.38
`doduc` (ref)	621	623	0.33	202	216	7.26	33.2	34.3	3.37
`fpppp` (NATOMS=4)	4,977	4,996	0.37	297	315	5.89	52.7	54.4	3.22
`fpppp` (NATOMS=6)	11,296	11,317	0.18	311	331	6.34	54.1	55.8	3.05

Table 6: The effects of unrolling

4.5 Smoothability Results

In all of the models that we have presented so far, we have assumed an infinite number of processors. It is therefore interesting to study the effect of limiting the number of processors. That is, if we really had a limited number of processors, is the parallelism profile smooth enough to be effectively mapped to those processors?

In Table 7 we list the parallelism results for the Omniscient Oracle and the Speculative Multithreaded Machine. For each model, we present the smoothability (defined in Section 3). In all cases under both models, the smoothability is better than 50%.

We also list the average operation delay, i.e., the average number of PI's by which each operation is delayed before it can be executed. (The average includes instructions which are executed as soon as they are ready, which are counted as 0.) Low numbers, as with `li` and `fpppp`, indicate that the application is already fairly evenly distributed in time, so not much "smoothing" is needed. High numbers, as with `doduc` (Speculative Multithreaded Machine) suggest that many operations are being delayed a long time before being executed. The high smoothability numbers, however, suggest that these delays do not significantly increase the lengths of critical paths.

Benchmark	Omni. Oracle	% Smooth-ability	Average Delay	Speculative Multithread.	% Smooth-ability	Average Delay
`tex` (man)	99.1	93.6	12.4	22.2	75.7	264
`tex` (draft)	192	98.3	23.3	12.2	75.4	75.2
`speech`	8,104	65.3	9.70	129	61.0	440
`li` (queens 7)	1,544	94.9	0.24	41.3	72.2	5.24
`espresso` (bca)	1,224	92.6	4.66	9.65	71.2	63.4
`espresso` (cps)	541	79.7	7.88	10.8	75.5	78.4
`espresso` (ti)	403	93.0	6.74	11.8	75.5	57.2
`espresso` (tial)	371	92.3	30.6	10.3	79.3	258
`eqntott`	43,298	62.6	73.5	300	64.3	26.4
`spice` (small)	76.5	92.2	2.40	27.2	75.8	16.6
`spice` (large)	62.8	97.4	19.4	22.4	78.0	78.9
`tomcatv` (N=33)	1,058	87.1	6.38	51.4	71.8	1.54
`tomcatv` (N=65)	2,108	85.5	13.8	52.6	71.9	2.38
`tomcatv` (N=129)	4,197	84.7	29.0	53.6	73.5	4.07
`tomcatv` (N=257)	8,418	84.4	59.1	54.2	73.9	7.53
`doduc` (tiny)	562	99.2	1.84	158	76.8	501
`doduc` (small)	617	99.4	1.82	174	77.2	544
`doduc` (ref)	621	99.4	1.79	202	80.8	518
`fpppp` (NATOMS=4)	4,977	98.2	6.00	297	59.1	0.97
`fpppp` (NATOMS=6)	11,296	98.7	3.93	311	59.0	0.912

Table 7: The effect of finite processors

4.6 Compiler Optimization

We wish to make one final point concerning optimization. All programs were compiled at maximum optimization, except for three of the FORTRAN programs. In these cases, optimization level had a significant impact on program parallelism. Optimization level –O2 was used on `tomcatv` because that yielded the best results for executing the sequential program, in terms of total instruction count and the numbers of memory accesses. However, what is more significant is that the Omniscient Oracle parallelism dropped from 1,058 to 38 when optimization was increased to -O3! This is surprising, since the Omniscient Oracle only accounts for true data dependencies, and we would not expect automatic compiler optimizations to change the overall pattern of data dependences in a program. We presently have no explanation for this anomaly.

Two other FORTRAN programs were compiled at sub-maximum levels because the increase in parallelism was far greater than the sequential instruction count. When –O2 was used for `spice`, the total number of PI's decreased by 16% even though there were 9% more sequential instructions. Using –O1 for `doduc` reduced total PI's by more than a factor of 8, even though there were 52% more sequential instructions. In these two cases, one may wish to multiply the parallelism figures reported by the reciprocal factors 91% (`spice`) and 66% (`doduc`) for a fair comparison with the optimized sequential program.

5 Previous Studies

Most previous studies on the limits of instruction-level parallelism have taken one of two approaches. One approach is to analyze the selected benchmark at either the source-code or object-code level, usually with a special interpreter that is based on a certain parallel machine model. The other approach, used in our study, is to schedule machine instructions from a trace generated by an actual execution of the object code. The following presents some of the previous work along both paths.

An early experimental tool designed by Kuck, et al [KMC72], compiled small Fortran programs to run on an abstract parallel machine. By recognizing parallelism at the source code level (e.g., parallel iterations of DO loops) and by aggressively reordering complex expressions, they were able to speed up most programs by a factor of 2-7, and speed up some by as much as 25. Tjaden and Flynn built a simulator based on the superscalar model [TF70]. Their simulator could execute ordinary IBM 7094 machine code, but attempted to reschedule operations dynamically using a look-ahead window of between 2 and 10 operations. They obtained speedups of up to 3 on a suite of 31 library routines. Both studies got modest results because they only looked for parallelism within small blocks, and could not execute independent instructions from separate blocks. These blocks are typically separated by conditional branches, and if the decision whether to execute one block is controlled by the outcome of a conditional branch in another block, the first block can't begin execution until the branch has been resolved.

Nicolau and Fisher wrote an interpreter for intermediate code generated by a compiler front-end, and a tool to analyze the instruction trace generated by the interpreter [NF84]. They addressed the shortcomings of the previous studies by allowing operations from many parts of the program to execute concurrently. They introduced the oracle model, which they added to their analysis tool. The analyzer also ignored any dependency which was not a "true" data dependency. They were able to obtain respectable speedups from small scientific routines.

The previous work discussed so far only studied small programs, primarily small numerical routines. To give a better idea of how *large* programs will fare on parallel machines, there have been several recent studies on this subjects [Kum88, BYP+91, Wal91, LW92, AS92]. Below we briefly survey work which are most relevant to ours.

Kumar [Kum88] developed a tool that automatically added statistics-gathering statements to FORTRAN source-code. By running the modified FORTRAN programs, he was able to measure parallelism figures for scientific applications. The tool also performed control-dependence analysis, so that independent FORTRAN statements separated by barriers could still run concurrently. Kunar found the potential for high levels of parallelism in regular numerical code.

Wall [Wal91] analyzed benchmarks (SPEC programs plus some real programs and toy benchmarks) in which the number of executed operations ranged from 1 million to 2 billion. He developed a series of models representing various levels of optimism about what the hardware and compiler could do. He then ran the benchmarks on a MIPS R3000 processor, and analyzed the instruction traces in a manner similar to Tjaden and Flynn, and Nicolau and Fisher. Even under his "perfect" model, with oracle-like branch prediction, parallelism rarely exceeded 40. This is because his schedule used a finite (2K) input window from which to fill parallel instructions, and because he did not allow for memory renaming.

Recently, Wilson and Lam have reported their study of the way control flow limits parallelism. They have demonstrated that substantially higher parallelism can be achieved by relaxing the constraints imposed by control flow on parallelism using control-dependence analysis, executing multiple flows of control simultaneously, and speculative execution [LW92]. They used six SPEC programs and 4 other programs as the test suite, and each program was simulated for 100 million instructions.

6 Conclusions

In this paper, we have presented a study of the instruction level parallelism and provided some answers to questions regarding the limits of parallelism and its smoothability.

We have reported on the construction of a testbed that was used to work with a wide variety of architecture models. This testbed allowed us to experiment with both standard benchmark programs and a real speech application. The testbed was implemented in such a way as to allow us to perform experiments on complete traces for reasonably large input data sets.

By examining an Omniscient Oracle and specializations of this oracle, we were able to study the key features that are important in finding large amounts of fine-grain parallelism. These results showed that restricting the oracle by making it myopic (finite instruction window) or frugal (no memory renaming) resulted in reducing the parallelism dramatically. As part of the Frugal Oracle test we looked at the effect of allowing only stack renaming, and showed that this is a useful feature for many programs. However, for other programs stack renaming is not enough, and heap renaming is also required. This indicates that further work at both the compiler and architecture level must be undertaken to remove the memory reuse bottleneck.

We also showed that the power of branch and jump prediction is very important, especially for programs with irregular control flow. This confirms the results of previous experiments. In particular, we found disappointingly low amounts of parallelism (¡9) for the irregular programs under the Multithreaded model. Other studies have seen the same results, even when they performed control-dependence analysis (see Section 2.1.3) to increase parallelism within a procedure. However, some applications, like `tex`, intuitively seem to have much more potential parallelism than found by our Multithreaded model (3.04) or Lam and Wilson's model with control-dependence analysis (6.18).

This has important implications for future directions in architecture. It is sometimes believed that to exploit maximum parallelism, we only need to rewrite existing algorithms in a better language, such as a functional language, so that an efficient compiler can extract all the available parallelism without being burdened by such sequential artifacts as memory reuse. But the Multithreaded model gets poor results, even though it eliminates false data dependencies, and represents what could be achieved by an efficient dataflow machine running well-compiled programs.

This would suggest that in many cases, large-scale parallelism will not be achieved merely by rewriting existing imperative-language programs in different languages. They must be redesigned completely, starting at the algorithmic level. Whether this will require the programmer to write programs with an explicit parallel machine model in mind, or to use a more abstract programming paradigm such as logic programming, is an interesting, but entirely different area of research.

If such changes are impractical, then one should consider some sort of speculation for programs with irregular control flow. We showed that this can significantly improve program performance. We also showed that if such speculation is used, simultaneous initiation of multiple loop iterations usually can boost program execution speed even more.

Smoothability has been demonstrated to be an interesting measure of the program parallelism. We reported the observation that all programs under test demonstrated smoothability better than 50%. Furthermore, different programs may exhibit different smoothabilities. Thus, there is good hope for mapping the available parallelism onto finite resources.

Our analyzer has proven to be a valuable experimental tool, and many useful extensions are possible. For future work, we plan to extend our existing testbed in the following directions:

- Add control-dependence analysis (see Section 2.1.3 to extract more parallelism from the non-oracle models.
- Add a memory latency model to the various experimental architectures.
- Connect the testbed with our compiler so that we can add measurements relative to the real alias analyses that can be performed by a parallelizing compiler.
- Add more programs to the test suite.
- Experiment with various multithreading scheduling strategies.

References

[A+90] Robert Alverson et al. The Tera computer system. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 1–6, Seattle, Washington, May 28–31, 1990. IEEE Computer Society and ACM SIGARCH. Also in *Computer Architecture News*, 18(2), June 1990.

[ALKK90] A. Agarwal, B. H. Lim, D. Kranz, and J. Kubiatowicz. APRIL: A processor architecture for multiprocessing. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 104–114, Seattle, Washington, May 28–31, 1990. IEEE Computer Society and ACM SIGARCH. Also in *Computer Architecture News*, 18(2), June 1990.

[AS92] T. M. Austin and G. S. Sohi. Dynamic dependence analysis of ordinary programs. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 342–351, Gold Coast, Australia, May 19–21, 1992. ACM SIGARCH and IEEE Computer Society. Also in *Computer Architecture News*, 20(2), May 1992.

[BYP+91] M. Butler, T-Y. Yeh, Y. Patt, M. Alsup, H. Scales, and M. Shebanow. Single instruction stream parallelism is greater than two. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 276–286, Toronto, Ontario, May 28–30, 1991. ACM SIGARCH and IEEE Computer Society. Also in *Computer Architecture News*, 19(3), May 1991.

[CSS+91] David E. Culler, Anurag Sah, Klaus Eric Schauser, Thorsten von Eiken, and John Wawrzynek. Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 164–175, Santa Clara, California, April 8–11, 1991. ACM SIGARCH, SIGPLAN, SIGOPS, and the IEEE Computer Society. Also in *Computer Architecture News*, 19(2), April 1991; *Operating Systems Review*, 25, April 1991; *SIGPLAN Notices*, 26(4), April 1991.

[DCC+87] W. J. Dally, L. Chao, A. Chien, S. Hassoun, W. Horwat, J. Kaplan, P. Song, B. Totty, and W. Wills. Architecture of a message-driven processor. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pages 189–195, Pittsburgh, Pennsylvania, June 2–5, 1987. IEEE Computer Society and ACM SIGARCH. Also in *Computer Architecture News*, 15(2), June 1987.

[HF88] R. H. Halstead Jr and T. Fujita. MASA: A multithreaded processor architecture for parallel symbolic computing. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 443–451, Honolulu, Hawaii, May 30–June 2, 1988. IEEE Computer Society and ACM SIGARCH. Also in *Computer Architecture News*, 16(2), May 1988.

[JW89] N. P. Jouppi and D. W. Wall. Available instruction-level parallelism for superscalar and superpipelined machines. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 272–282, Boston, Massachusetts, April 3–6, 1989. ACM SIGARCH, SIGPLAN, SIGOPS, and the IEEE Computer Society. Also in *Computer Architecture News*, 17(2), April 1989; *Operating Systems Review*, 23, April 1989; *SIGPLAN Notices*, 24, May 1989.

[KMC72] D. J. Kuck, Y. Muraoka, and S. C. Chen. On the number of operations simultaneously executable in FORTRAN-like programs and their resulting speed-up. *IEEE Transactions on Computers*, C-21(12):1293–1310, December 1972.

[KS88] J. T. Kuehn and B. J. Smith. The horizon supercomputing system: Architecture and software. In *Proceedings of Supercomputing '88*, pages 28–34, Orlando, Florida, November 1988. IEEE Computer Society and ACM SIGARCH.

[Kum88] M. Kumar. Measuring parallelism in computation-intensive scientific/engineering applications. *IEEE Transactions on Computers*, C-37(9):1088–1098, September 1988.

[LW92] M. S. Lam and R. P. Wilson. Limits of control flow on parallelism. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 46–57, Gold Coast, Australia, May 19–21, 1992. ACM SIGARCH and IEEE Computer Society. Also in *Computer Architecture News*, 20(2), May 1992.

[NA89] R. Nikhil and Arvind. Can dataflow subsume von Neumann computing? In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 262–272, Jerusalem, Israel, May 28–June 1, 1989. IEEE Computer Society and ACM SIGARCH. Also in *Computer Architecture News*, 17(3), June 1989.

[ND91] P. R. Nuth and W. J. Dally. A mechanism for efficient context switching. *Proceedings of the IEEE International Conference on Computer Design*, 1991.

[NF84] Alexandru Nicolau and J. A. Fisher. Measuring the parallelism available for very long instruction word architectures. *IEEE Transactions on Computers*, C-33(11):968–976, November 1984.

[TF70] G. S. Tjaden and M. J. Flynn. Detection and parallel execution of independent instructions. *IEEE Transactions on Computers*, C-19(10):889–895, October 1970.

[Wal91] D. W. Wall. Limits of instruction-level parallelism. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 176–188, Santa Clara, California, April 8–11, 1991. ACM SIGARCH, SIGPLAN, SIGOPS, and the IEEE Computer Society. Also in *Computer Architecture News*, 19(2), April 1991; *Operating Systems Review*, 25, April 1991; *SIGPLAN Notices*, 26(4), April 1991.

[WG89] W. D. Weber and A. Gupta. Exploring the benefits of multiple contexts in a multiprocessor architecture: Preliminary results. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 273–280, Jerusalem, Israel, May 28–June 1, 1989. IEEE Computer Society and ACM SIGARCH. Also in *Computer Architecture News*, 17(3), June 1989.

A Experimental Testbed

The analyzer consists of two parts. The first part, the decoder, takes the stream output from an existing trace generator, and decodes the opcode and other information, performing basic block flow analysis, and deriving the relevant dependence data. The second part, the scheduler, packs the operations into PI's based on this data. The scheduler can run multiple models simultaneously, provided there is enough memory.

Section 2.1 talks about building a dependence graph for the entire program. Such an undertaking would be out of the question for benchmarks involving billions of operations. Fortunately, since the trace is generated by a sequential thread of control, it is only necessary to store the dependence information relative to the last access of each storage location. Thus, the amount of memory needed to record and compute dependences is proportional to the total number of locations used, *not* the total number of *times* they are used.

A table records the earliest PI in which a given storage location can be read and written. The following shows what the table would look like after the execution of the branch at the end of the first iteration the left half of Figure 1. Note that integer registers 2–4 are defined prior to entering this loop.

Register	Read	Write
Int. register 1	$i+3$	$i+3$
Int. register 2	$*$	i
Int. register 3	$*$	i
Int. register 4	$*$	$i+2$
$\vdots$	$\vdots$	$\vdots$
Int. cond. codes	$i+3$	$i+3$
FP register 0	$i+1$	$i+1$
FP register 1	$i+1$	$i+1$
FP register 2	$i+2$	$i+2$
$\vdots$	$\vdots$	$\vdots$

When the second occurrence of S1 is encountered in the trace, the decoder will see that it is dependent on writing `%f0`, and reading `%1`, `%2`, and the memory location read by the load. The maximum of all the corresponding entries in the table is $i+3$. If the branch were predicted correctly, then the operation could be packed into PI $i+3$. Without branch prediction, the operation would have to go into PI $i+4$, because the scheduler also keeps track of the most recent barrier. Once the operation has been packed, the read and write entries for `%f0` are updated with either $i+4$ (branch predicted) or $i+5$ (branch not predicted).

If there is no memory disambiguation, then a single entry in the table represents all memory accesses. Otherwise, there must be a separate entry for each unit of memory. This is done using a hash table of linked lists of tables representing individual pages of memory. Renaming, for both register and memory use, is done by ignoring the write component of the table entries. For models with partial memory renaming (e.g., the Tree Oracle), the decoder uses the memory address to determine if it is in the stack or the heap.

The scheduler keeps track of the most recent barrier encountered, unless an oracle is used. If non-oracle branch prediction is used, the branching hash table is used to derive the *predicted* branch outcome, which is compared to the *actual* outcome revealed in the trace. If a misprediction results, the barrier is updated. If procedure separation is used, then barriers are kept on a stack. When a procedure

is called, a new barrier is allocated on the stack, and starts with the barrier of the calling procedure. When a procedure is exited, its barrier is discarded and the calling procedure's barrier again becomes the current barrier.

Finite processor and window length limits require keeping a table of how many operations are packed into each PI. Because this table can become quite large, it was necessary to perform optimizations such as freeing up space whenever a long contiguous series of PI's is filled to the limits. If a finite window length is used, as in the Myopic Oracle, then an additional barrier is needed to mark the first unissued PI. The analyzer frees up storage as this barrier moves forward.