McGill University
School of Computer Science
ACAPS Laboratory

Advanced Compilers, Architectures
and Parallel Systems

The Effects of Resource Limitations on Program Parallelism

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

ACAPS Technical Memo 52
January 29, 1993

Presented at the Second Workshop on Data-Flow Computing,
Hamilton Island, Australia, May 24–26, 1992

ACAPS • School of Computer Science • 3480 University St. • Montréal • Canada • H3A 2A7

Abstract

Many of the studies of the "limits of instruction parallelism" in application programs made use of machine models with unlimited or ideal resources. In this study, we impose certain limitations on machine resources and measure the effects of those limitations. Beginning with an ideal "oracle" model, we selectively remove the ideal branch-prediction capability, and the ability to rename variables in the heap and stack, to measure the impact of control dependences and memory reuse on potential parallelism.

Next, we study what happens when the number of processors is limited, which forces some operations to be deferred to a later time. We introduce and define *smoothability*, which quantifies how much the parallelism and processor utilization are affected by such deferment. Finally, a memory latency model is added to measure the effects of non-local memory accesses. Our experiments show that programs with a large amount of intrinsic parallelism are quite "smoothable," even when memory latency is included. This suggests that real multiprocessor designs should be able to run these programs with a high level of processor utilization.

Keywords: Computer architecture modeling, dataflow computers, latency tolerance, parallelism, smoothability.

Contents

List of Figures

List of Tables

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 dataflow or multithreaded architectures [1, 4, 5, 7, 9, 14, 16]. Such processors can then become the building blocks of an advanced, massively-parallel computer.

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

Limits of Parallelism: How much parallelism exists in the target applications?

Resource Limits: How is the available parallelism affected by limitations in machine resources? In particular, is it "smooth" enough to be effectively exploited when the number of processors is limited, and when non-local memory accesses have high latency?

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; it must be possible to distribute the operations evenly 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 [2, 3, 10, 11, 12, 13, 21, 22]. Most studies measured the limits of parallelism under ideal conditions (e.g., where parallelism is only limited by true flow dependences), and then measured the drop in parallelism when specific architectural limitations were introduced into the model. Our experimental methods are similar, but we study different kinds of limitations:

1. Our "idealized" model includes the ability to rename memory objects in the stack and the heap, which eliminates false data dependences. By selectively removing this ability from the heap and the stack, we can observe the impact on parallelism of the reuse of memory.

2. We consider what happens when the number of processors is limited. For instance, we measure the speedup (average parallelism) attainable from an unlimited number

of processors, then limit the number of processors to that speedup value. In such an experiment, an operation occurring during a period of peak parallelism may have to be deferred to a later time when fewer operations are being performed. We introduce *smoothability* as a measure to characterize program parallelism. Smoothability is measured quantitatively and compared for our test programs. Our results indicate that the parallelism obtained has a relatively smooth temporal profile.

3. Finally, we consider the effects of non-uniform memory access latencies, to model what would be expected on a large-scale multiprocessor with either distributed memory or cached shared memory. We find that while memory latency can reduce the upper limits of parallelism, the parallelism is still quite smoothable, and it is possible for processors to tolerate non-local memory latencies by performing other operations while waiting for the accesses to complete.

To search for the limits of parallelism and measure its smoothability under different architectural ideas, we have developed an experimental testbed called SITA (Sequential Instruction Trace Analyzer) [18, 19, 20]. This testbed is based on a trace simulation method similar to previous experiments [2, 3, 12, 22]. The tool allows us to analyze the execution of real code and calculates, under a chosen architecture model, how much parallelism could potentially be attained for that model. Using this tool, we began with a true "Omniscient Oracle" machine model and measured the idealized limit of parallelism. We then examined the degradation in performance as we selectively added constraints to the resources of the Omniscient Oracle and moved it towards more realistic machine models at different levels. We ran a total of 405 experiments, whose results are presented in this paper.

Our oracle machine models, like oracles in other studies [13], ignore control dependences entirely. While the results obtained from the oracle are an upper bound, and may be unrealistically high, we believe that suitably-designed programs with appropriate parallel algorithms can get parallelism results which are much closer to the oracle. (One shortcoming common to most of the cited parallelism studies is that they look for parallelism in code that had been originally written for uniprocessors.)

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. Instead, 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 then measure the achievable parallelism, but rather to study which 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. Section 3 presents the results from the basic studies, in which all memory accesses are fast and there are infinitely many processors. Smoothability is defined and discussed in Section 4. The latency model is added in Section 5. In Section 6, we compare our work to previous studies. The final section summarizes the paper, discusses the conclusions that may be drawn from this study, and describes how we plan to extend SITA in the future. Appendix A discusses the implementation of SITA, and experimental results are presented in tabular form in Appendix B.

2 Methodology

This section describes the trace simulation techniques 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 describe the tool used to analyze the traces, and the benchmarks used in the experiments.

2.1 Trace Simulation

Figure 1 shows the basic methodology used in our study. We would like to determine how a benchmark program might be executed on an idealized machine with a given set of architectural characteristics, as shown in the left fork in the flowchart. Lacking such a machine, we model it using trace simulation, as shown in the right fork.

First, a benchmark program is compiled into an executable file using a conventional sequential compiler. Then the executable code is run through a trace generator, which produces a program trace. This trace consists of a stream of operations representing the actual sequence of instructions executed (*not* the static object code). For each executed operation, the trace gives the opcode, PC address, and load or store address (if any).

The trace scheduler reads the operations from the stream and packs them into *parallel instructions* (PI). As the scheduler reads each operation in the trace, it inserts the operation into the earliest PI possible, while simultaneously respecting the dependences between that operation and all previous operations. The following types of dependences between operation S_1, inserted into PI_i, and a later operation S_2 may exist:

- *Flow dependence:* if S_2 reads a storage location (register or memory cell) which was most recently written by S_1, then S_2 can be scheduled no earlier than PI_{i+1}.
- *Anti dependence:* if S_2 writes a storage location which was most recently read by S_1, then S_2 can be scheduled no earlier than PI_i. (It is assumed that the write and read can occur simultaneously, and S_1 will read the proper value.)

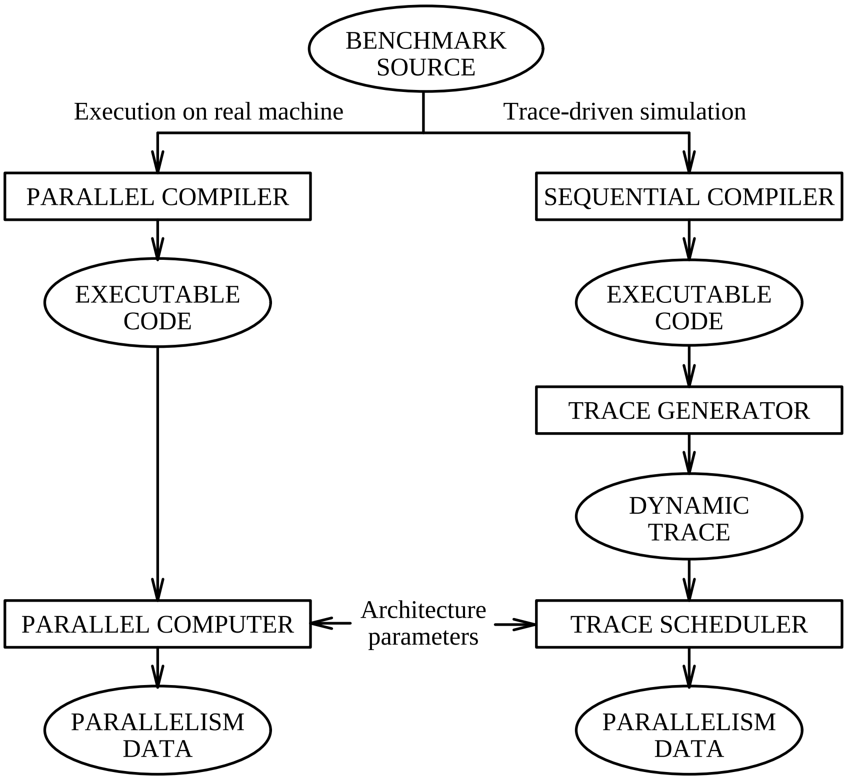

Figure 1: Trace simulation methodology

- *Output dependence:* if S_2 writes a storage location which was most recently written by S_1, then S_2 can be scheduled no earlier than PI_{i+1}.
- *Control dependence:* if S_1 is the most recent conditional branch in the trace whose outcome must be decided before it is known whether or not S_2 will be executed, then S_2 can be scheduled no earlier than PI_{i+1}.

For the initial experiments, 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. Infinitely many functional units are available to execute all operations which can be packed into one PI. (In later sections, we limit the number of processors and add latency to the memory accesses.) Operations arbitrarily far apart in the dynamic trace can be packed into the same PI, so long as all other dependence 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:

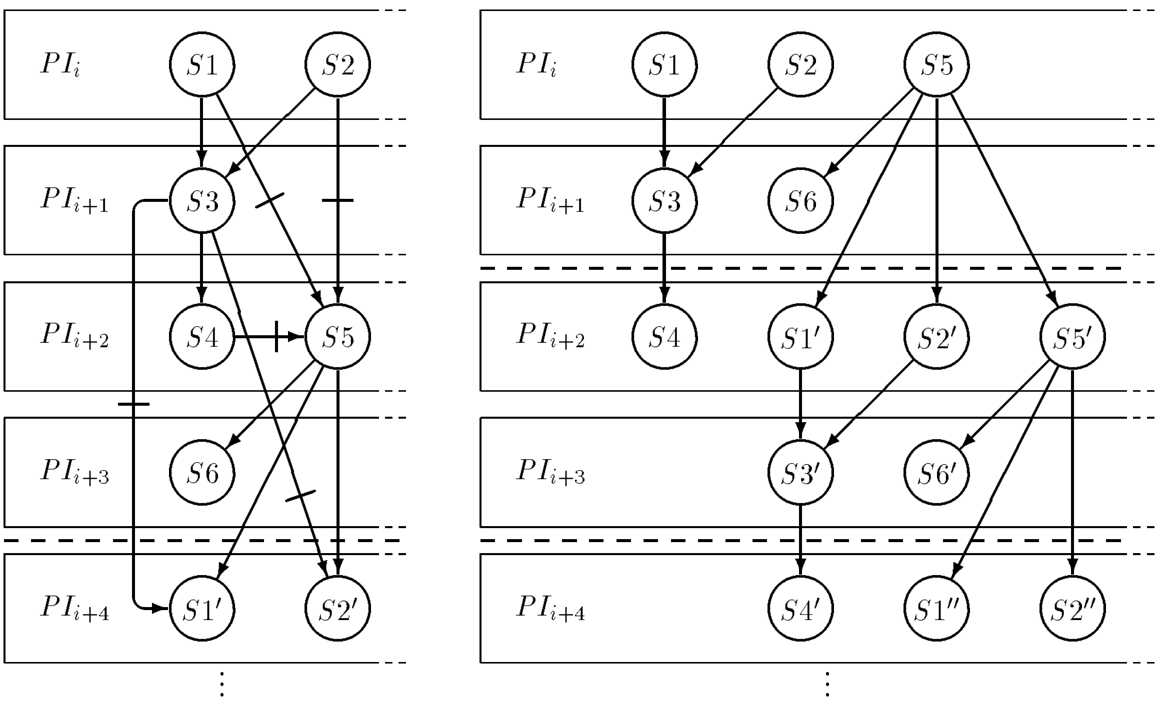

Figure 2: Packing parallelizable instructions: An example

```
S1:  ld [%o2+%o1],%f0
S2:  ld [%o3+%o1],%f1
S3:  fmuls %f0,%f1,%f2
S4:  st %f2,[%o4+%o1]
S5:  subcc %o1,4,%o1
S6:  bg S1
S7:  nop
```

In a dynamic trace, the sequence S1-S7 may be repeated many times. The left half of Figure 2 shows how our trace analyzer would pack these operations into PI's if all four dependence types listed above were obeyed. Plain arcs are drawn to show the flow dependences between operations. The arcs with small marks (e.g., from S4 to S5) indicate anti dependences. The dashed line represents a barrier caused by the conditional branch at S6, which may prevent future operations from being scheduled before that barrier.[1] One iteration of the loop can be initiated every 4 cycles, so the parallelism is 1.5. (Parallelism is defined as the total number of sequential operations divided by the total number of PI's required by the scheduler.) The right half of Figure 2 shows how the operations would be packed if anti and output dependences were ignored.

[1]For operations S1–S3 and S5, there are also output dependences between corresponding operations in different iterations; these have been left out of Figure 2 for clarity.

Other models, described later, relax the dependence constraints; some also tighten the resource limitations. The next three 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.

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, in which case a dependence would exist between them. Thus, for instance, a conservative scheduler would have to assume that a flow dependence might exist between S4 in iteration j and S1 in iteration $j+1$, making it harder to overlap separate iterations of the loop.

However, since the actual addresses of memory accesses appear in the trace, a trace 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. Our tool allows the user to select run-time memory disambiguation.

2.1.2 Register and Memory Renaming

A false dependence (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 %o1 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 dependences may also exist in references to main memory. This can reduce parallelism in two ways:

1. Many sequential programs reuse data structures (such as arrays) to conserve 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 dependences can be eliminated by ensuring that each register or memory location is written only once. In a real processor, false dependences 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 resolve the anti and output dependences.[2]

False dependences in the heap can, in principle, be eliminated by adhering to the single-assignment rule, either explicitly in the programming language (e.g., Sisal [6]), or by using *memory renaming*. Conventional architectures do not presently perform memory renaming, although dynamic dataflow machines with "colored tokens" [1] employ a form of renaming.

False dependences in the stack can be caused by the linear stack model, the standard run-time memory organization for supporting nested procedure invocations on sequential machines. Such false dependences can be eliminated, for example, by organizing the memory frames for procedure invocation in a tree-like structure, as proposed in several dataflow and multithreaded architecture models [4, 14, 16].

Infinite renaming, in which a register or memory location can be renamed any number of times, is equivalent to ignoring all false dependences between objects of a particular type. For instance, if register renaming is applied to the dependence graph in Figure 2, then S5 can be executed in parallel with S1 and S2 of the same iteration, as the anti dependence 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 disambiguation is also used, then the iteration issue rate increases to once every other cycle, raising parallelism to 3, as the right half of Figure 2 shows.[3]

2.1.3 Control Barrier Elimination

[2]For instance, the IBM RISC 6000 superscalar machine implements a form of register renaming in its floating point unit [15].

[3]This assumes the input and output arrays do not overlap in such a way as to create flow dependences.

Previous parallelism experiments in which conditional branches were barriers to upward motion of all operations generally had low parallelism, especially in non-numerical programs. This is because conditional branches occur quite frequently in most programs, which confines 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 [13], makes the most optimistic assumptions by ignoring control dependences entirely. Our oracles, like most others, also ignore output and anti dependences, so that parallelism is only limited by true flow dependences 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 way to reduce the serializing effects of conditional branches is to analyze the code to determine which parts of the code are really control-dependent on a given branch. For instance, while the then-clause and else-clause of an if-statement are control-dependent on the if-expression, the statements *following* the if-statement often are not. Similarly, two consecutive but independent for-loops should be able to run concurrently. Also, the barriers produced by branches in one procedure usually do not affect the scheduling of operations within other procedures.

These examples show why we have adopted the definition of control dependence in Section 2.1 rather than making a branch block the upward motion of *all* succeeding instructions in the trace. We assume that a compiler can analyze the source code to determine the true control dependences, so that operations on a parallel architecture can be scheduled accordingly. For some scientific code with regular control structures, this analysis is sufficient to yield high levels of parallelism close to an oracle [12, 20].

If this is not enough, then *speculative execution* can be used to get more parallelism. 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 mispredicted 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 all relevant barriers created by correctly-predicted branches, and keeping barriers created by incorrectly-predicted branches (see Appendix A). We used 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 [17, 20, 22]. Jump prediction is done simply by predicting that the processor will jump to the previous jump destination.

2.2 Experimental Tool and Benchmarks

To measure the parallelism available under the various models, the SITA (Sequential Instruction Trace Analyzer) testbed was developed. Since programs can show significant differences in behavior when run on short inputs and on larger inputs, we designed SITA 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.

In this study, we ran our analysis tool on five benchmarks under nine basic models (described in the next section), with additional tests done after limiting the processor resources and adding memory latency. We chose problems that we feel 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 a regular FP-intensive scientific program, written in Fortran, and four symbolic applications written in C. All are from standard benchmark test suites, except for a speech-recognition program taken from an actual industrial application. We used standard input cases, except that we reduced the size of the queens problem run on the Lisp interpreter (`li`) from nine queens to seven. Also, we tried two different sizes for the scientific benchmark. The tests are summarized in Table 1.

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 SITA to print intermediate results at regular intervals, we found, for instance, that total parallelism of `eqntott` under one of the models dropped by half between the checkpoints at 100 million operations and half a billion operations.

Source	*Program*	*Description*	*Test Case*	*Useful Ops* $\times 10^6$	*Call Depth*	*% of ops.* *FP*	*Ld.*	*St.*
DLX suite	`tex`	Text formatting	`draft (11 p.)`	109	23	.04	15	8.0
Industry	`speech`	Speech recognit.	recognize “he”	551	12	4.5	14	2.8
SPEC89	`li`	Lisp interpreter	`(queens 7)`	205	75103	0	22	8.6
test	`eqntott`	Truth-table gen.	`int_pri_3.eqn`	1,770	18	0	33	0.7
suite	`fpppp`	Quantum	NATOMS=4	277	14	19	43	10
		chemistry	NATOMS=6	1,258	14	19	43	10

Table 1: Benchmarks and test cases used

3 Basic Experiments

This section presents the results of our “basic” experiments, involving models with infinitely many processors and uniform memory latency. These experiments were intended to find the upper bounds of parallelism in a program, and to measure how these limits are affected by control dependences and the reuse of memory. In the two sections that follow this one, we limit the number of processors and add memory latency.

In general, we chose a top-down approach. From the discussion in the previous section, it is clear that the most ideal oracle model, which we call the *Omniscient Oracle*, should have the following features:

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

We then created models in which one or more features are restricted, to measure their importance. We ran a total of 54 experiments. All the basic models used are summarized in Table 2 and described in the individual subsections that follow.

3.1 Effects of Branch and Jump Prediction

Model Name	*Memory Disamb.*	*Rename Regs.*	*Rename Memory*	*Control Barriers*
Omniscient Oracle	yes	yes	all	none
Speculative Dataflow	yes	yes	all	all relevant; prediction
Fine Dataflow	yes	yes	all	all relevant
Coarse Dataflow	yes	yes	all	all in same procedure
Smart Superscalar	yes	yes	all	all
Tree Oracle	yes	yes	stack	none
Linear Oracle	yes	yes	heap	none
Frugal Oracle	yes	yes	none	none
Stupid Superscalar	no	no	none	all

Table 2: Basic machine models

The first five models in our study are the *Omniscient Oracle, Speculative Dataflow, Fine Dataflow, Coarse Dataflow,* and *Smart Superscalar* models. All of these models have perfect memory disambiguation (see Section 2.1.1) and infinite memory/register renaming (see Section 2.1.2). The only difference among them is how much they are affected by control dependences.

The Omniscient Oracle has perfect foreknowledge of every jump and branch outcome, while the other four do not. Therefore, the latter are constrained by control dependences. These four models reflect different levels of optimism about how much can be done to determine true control dependences and predict branches. In the most pessimistic model, the Smart Superscalar, if a conditional branch is scheduled in PI_i, then *all* instructions that follow it in the trace must be scheduled no earlier than PI_{i+1}, whether or not they are control-dependent on that branch (as defined in Section 2.1). In the Coarse Dataflow model, a conditional branch only impacts the scheduling of a future instructions if both are within the same procedure invocation (we call this rule *procedure separation*). The Fine Dataflow machine is even more lenient; any two instructions may be scheduled concurrently if they are not directly or indirectly control-dependent (as defined in Section 2.1) even if there are conditional branches between them in the trace. This model's name comes from the fact that its behavior models what might occur on an idealized dataflow machine (with infinite renaming) executing a sequential benchmark without the algorithm modifications that would be necessary to approach the oracles' levels of performance. The Speculative Dataflow model attempts to predict branches so that control-dependent instructions may be speculatively executed before the branch outcomes are known, but unlike an oracle, it sometimes makes mistakes.

Examining Table 3, we note that the results vary from surprisingly high parallelism results for the Omniscient Oracle to generally poor parallelism results for the Smart Superscalar model. In the case of the Omniscient Oracle, we find impressive parallelism for `speech` (8,105), `eqntott` (43,298), and `fpppp` (4,978 – 11,296). Reasonable parallelism was measured for `li` (1,570), and moderate parallelism was measured for `tex` (192).

Benchmark	*Omniscient Oracle*	*Speculative Dataflow*	*Fine Dataflow*	*Coarse Dataflow*	*Smart Superscalar*
`tex`	192	36.3	5.31	3.04	2.08
`speech`	8,105	1,192	45.4	6.30	1.80
`li`	1,570	59.4	9.71	3.91	1.69
`eqntott`	43,298	675	2.24	1.66	1.46
`fpppp` (4)	4,978	4,166	1,179	52.7	30.1
`fpppp` (6)	11,296	10,720	3,023	54.1	32.6

Table 3: Results on five basic models

From Table 3 we can make the following observations:

- For some programs like `li` and `eqntott`, the power to predict jumps and branches *perfectly* is extremely important. As expected, for scientific programs that have regular, data-independent behavior like `fpppp`, branch prediction is much less of an issue, as shown by the relatively high performance of the Fine Dataflow model.
- There is a substantial parallelism loss between the Speculative Dataflow model and the Fine Dataflow model, especially for the symbolic benchmarks. This result indicates that for programs with irregular control flow, speculative execution may be very important.
- In most cases, the performance of the Coarse Dataflow machine is near the geometric mean of the Fine Dataflow and Smart Superscalar machines. This shows that it is important to avoid false control dependences both *between* procedures (by using procedure separation) and *within* procedures (by performing control-dependence analysis). The latter is clearly more important for numerical programs like `fpppp`, since procedures in these programs tend to be larger and less frequent.

3.2 Effects of Memory Reuse

Having seen the effects of control dependences on parallelism, we wanted to measure how the reuse of memory in conventional programs can affect parallelism. Therefore, our next experiments were with "frugal oracles." These oracles are just like the Omniscient Oracle, but they don't have full memory-renaming capability. 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.[4] 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.

[4]In this study, "heap" includes both dynamically-allocated and global (static) variables.

Benchmark	*Omni. Oracle*	*Tree Oracle*	*% of O.O.*	*Linear Oracle*	*% of O.O.*	*Frugal Oracle*	*% of O.O.*	*Smart Super.*	*Stupid Super.*	*% of Smart*
`tex`	192	75.8	39.4	167	86.9	71.5	37.2	2.08	1.66	80.8
`speech`	8,105	136	1.68	57.1	0.70	53.6	0.66	1.80	1.62	89.9
`li`	1,570	59.5	3.79	343	21.8	59.3	3.78	1.69	1.44	85.2
`eqntott`	43,298	1,742	4.02	1,314	3.03	1,314	3.03	1.46	1.43	98.2
`fpppp` (4)	4,978	4,978	100	71.0	1.43	70.9	1.42	30.1	2.88	9.56
`fpppp` (6)	11,296	11,296	100	73.7	0.65	73.6	0.65	32.6	2.91	8.94

Table 4: The effect of frugal use of memory

The results of these experiments are shown in Table 4. In this table, we list the parallelism measurements for the Tree Oracle, the Linear Oracle, and the truly Frugal Oracle. 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`. This reflects the fact that the programs are computing with large dynamically-allocated data structures. Heap renaming is irrelevant to `fpppp`, mostly due to the lack of heap-allocated data structures in Fortran programs (only global variables are affected by this renaming capability).

- 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 every case, the Frugal Oracle performs as well or almost as well as the slower of the Tree Oracle and Linear Oracle.

We ran one final model, the *Stupid Superscalar,* to test the benefits of renaming and disambiguation at the "low" end of the scale. The Stupid Superscalar is like the Smart Superscalar, but has neither memory disambiguation nor register/memory renaming.[5] Results for this model are given in the rightmost two columns of Table 4.

[5]The Stupid Superscalar is similar to Wall's "Stupid" model [22], except that Wall's model was limited to 64 processors, and could only schedule operations from within a window of 2,048 instructions. These limitations, however, had almost no additional impact on the Stupid model, since register and memory reuse, control dependences, and lack of memory disambiguation generally removed almost all available parallelism except that within basic blocks.

Both superscalar models have the property that a conditional branch affects the scheduling of all future instructions. Thus, these machines can generally only find parallelism within basic blocks. Indeed, the rightmost column of Table 4, which gives the performance of the Stupid Superscalar relative to the Smart Superscalar, shows that for irregular code, renaming and disambiguation by themselves produce little benefit. Only `fpppp` shows a big gain when renaming and disambiguation are added, probably due to its large basic blocks and regular code structure. For codes with frequent branches, we must go to more aggressive models, such as the Fine Dataflow and Speculative Dataflow models.

4 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 parallelism limits. Furthermore, the utilization of these processors will be 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.

SITA has the ability to limit 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 SITA'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 3.

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 3. Once n reaches $P(\infty)$, that value becomes the new bound, since $P(n)$ can never be greater than $P(\infty)$; this is shown by a 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

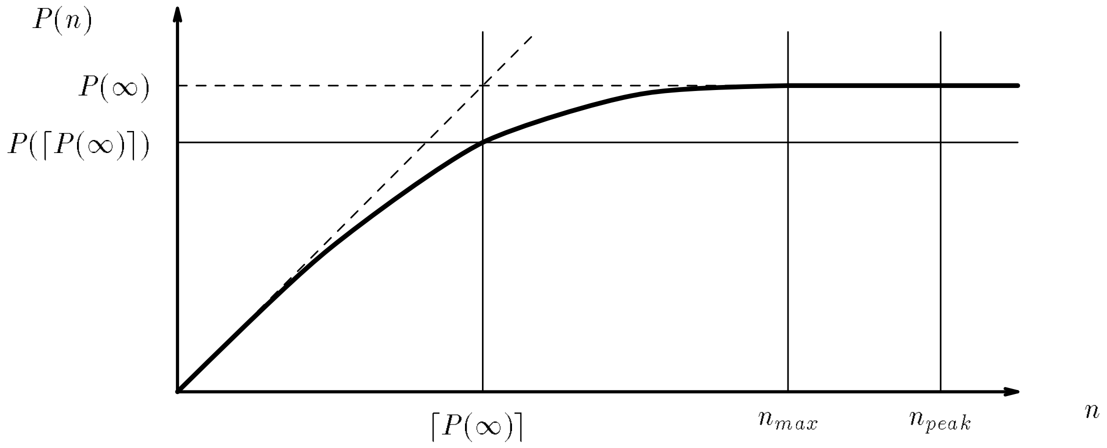

Figure 3: Standard smoothability profile

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 perfectly "smooth" (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 a critical path, and it is delayed by k cycles, it is possible that every future operation which is dependent on it 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 normalize $P(n)$ with respect to that upper bound, to measure how well the processors are utilized compared to the ideal case:

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

A normalized curve corresponding to the curve in Figure 3 is shown in Figure 4.

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

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

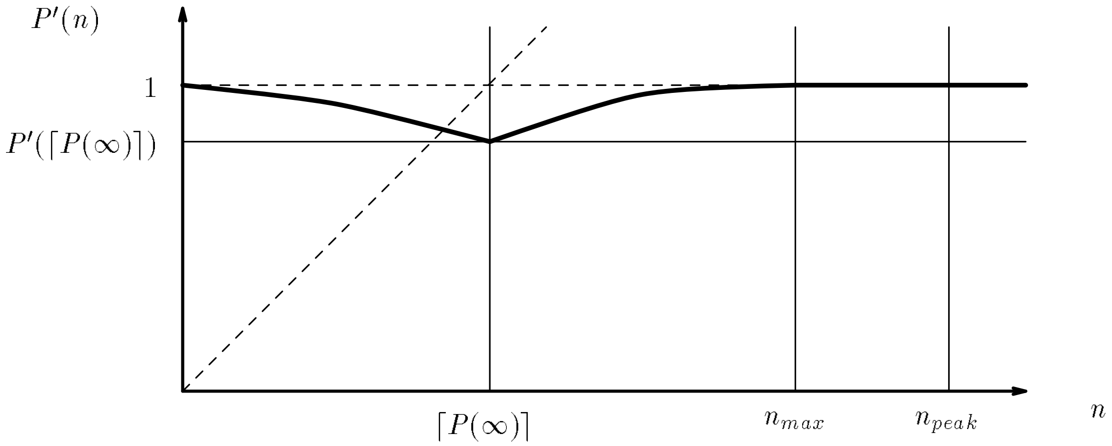

Figure 4: Normalized smoothability profile

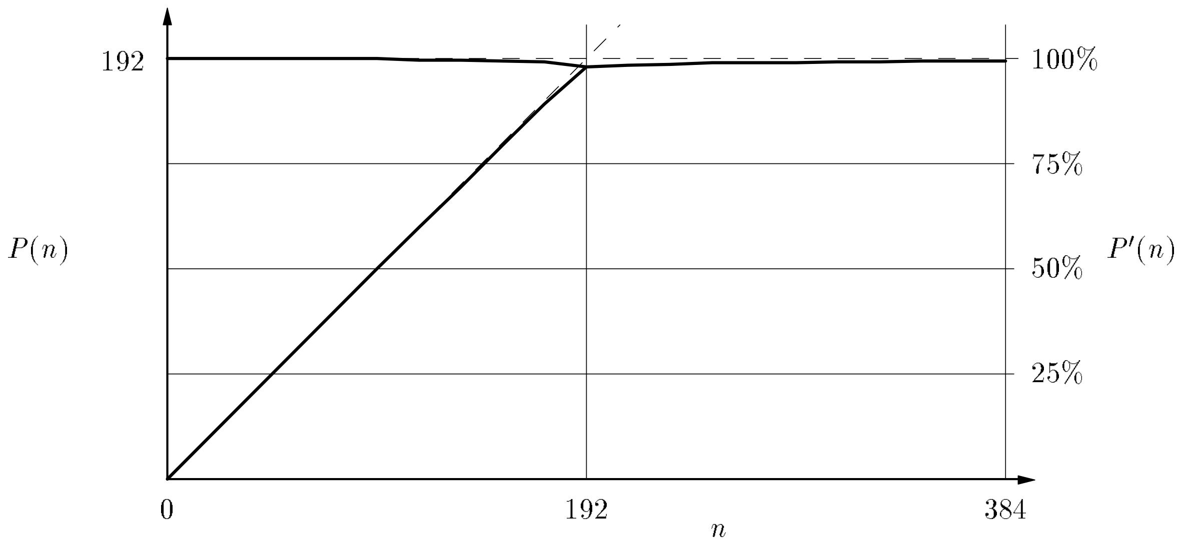

Figure 5: Smoothability profile for `tex` ($S = 98.3\%$)

SITA 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.

The next six figures (5–10) show parallelism curves for the six benchmarks used in our experiments, under the Omniscient Oracle model. To generate each curve, we ran the benchmark 20 times, limiting the number of processors to $kP(\infty)$ (rounded to the nearest integer), where k ranged from 0.10 to 2.00 by increments of 0.10, and $P(\infty)$ came from the Omniscient Oracle column of Table 3. Therefore, each graph shows n (the number of processors) between 0 and $2P(\infty)$. A total of 120 experiments were run.

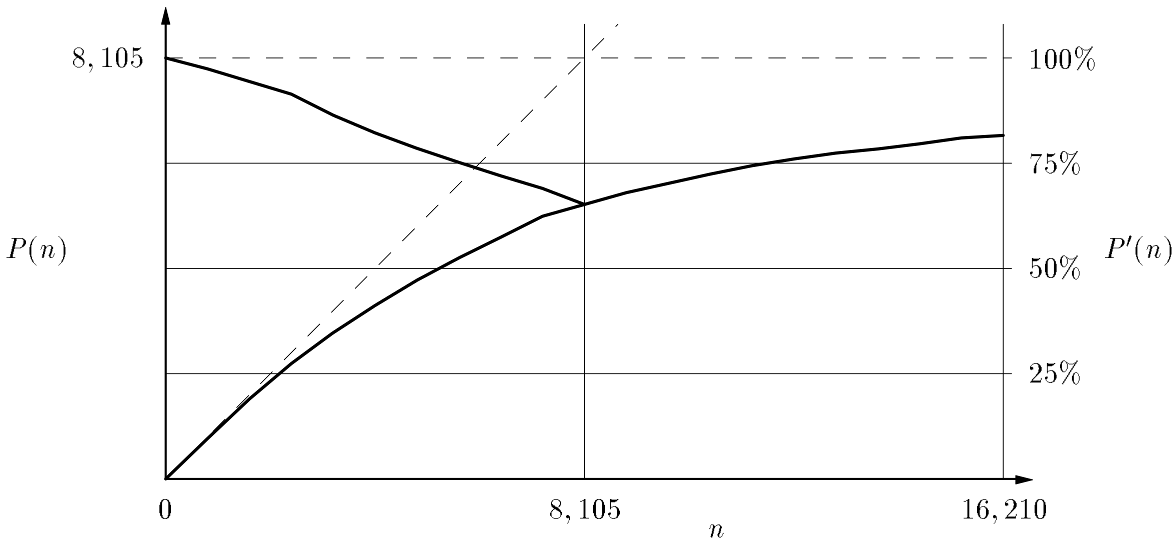

Figure 6: Smoothability profile for `speech` ($S = 65.3\%$)

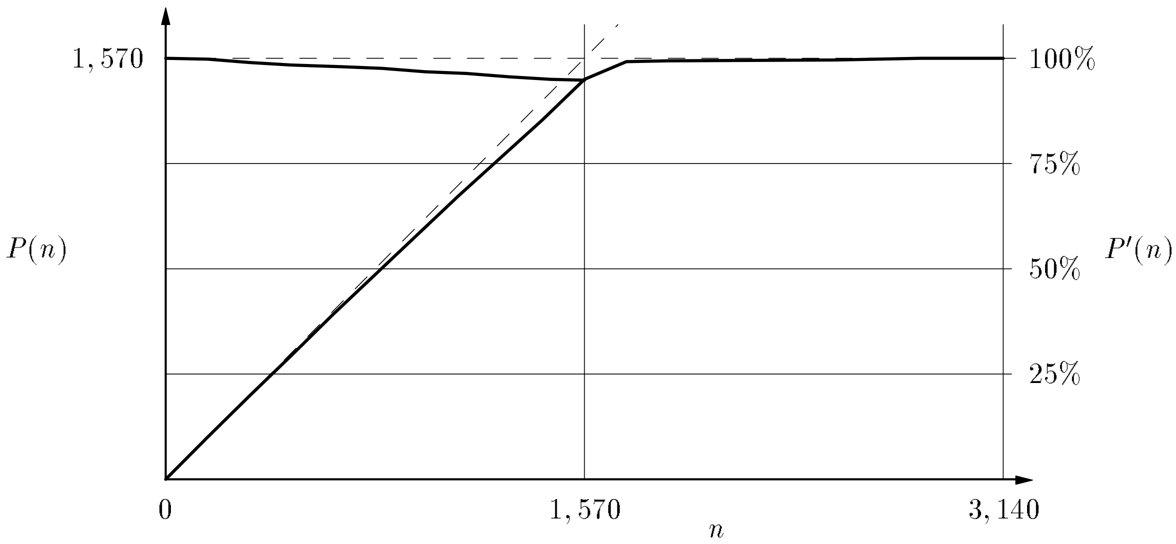

Figure 7: Smoothability profile for `li` ($S = 94.9\%$)

The curve splits into two parts to the left of $n = P(\infty)$. The lower curve represents $P(n)$, corresponding to Figure 3, and measures absolute parallelism, as shown on the scale at the left of each graph. The upper curve represents $P'(n)$, corresponding to Figure 4, and measures the percentage of parallelism achievable compared to the ideal case, as shown on the scale at the right of each graph. With these scales, both curves coincide to the right of $n = P(\infty)$.

From the six curves shown, the following observations can be made:

- The most smoothable application is `fpppp`. This program performs physical simulation,

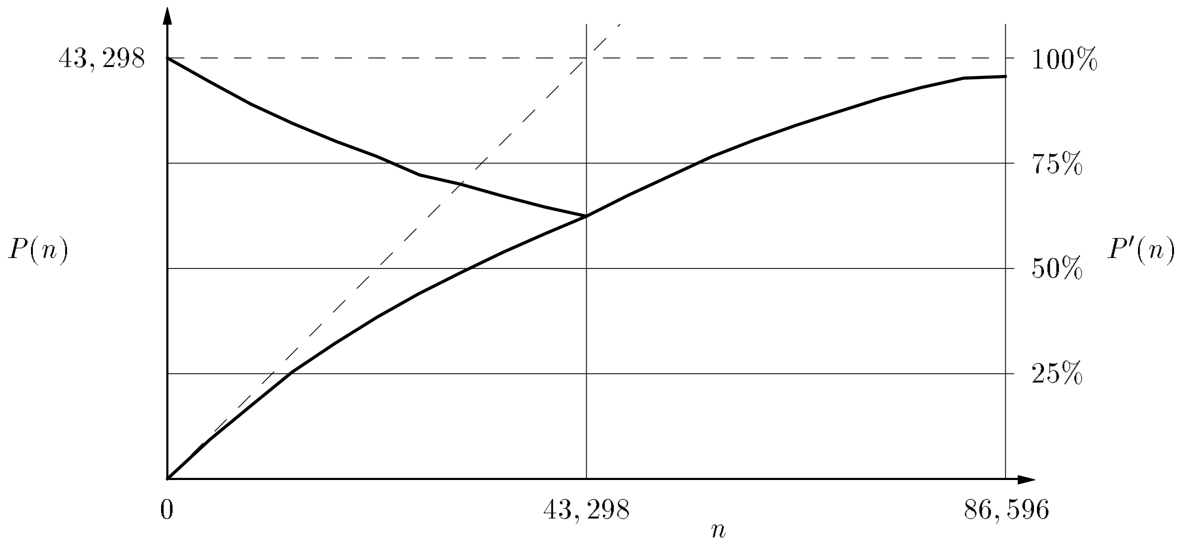

Figure 8: Smoothability profile for `eqntott` ($S = 62.6\%$)

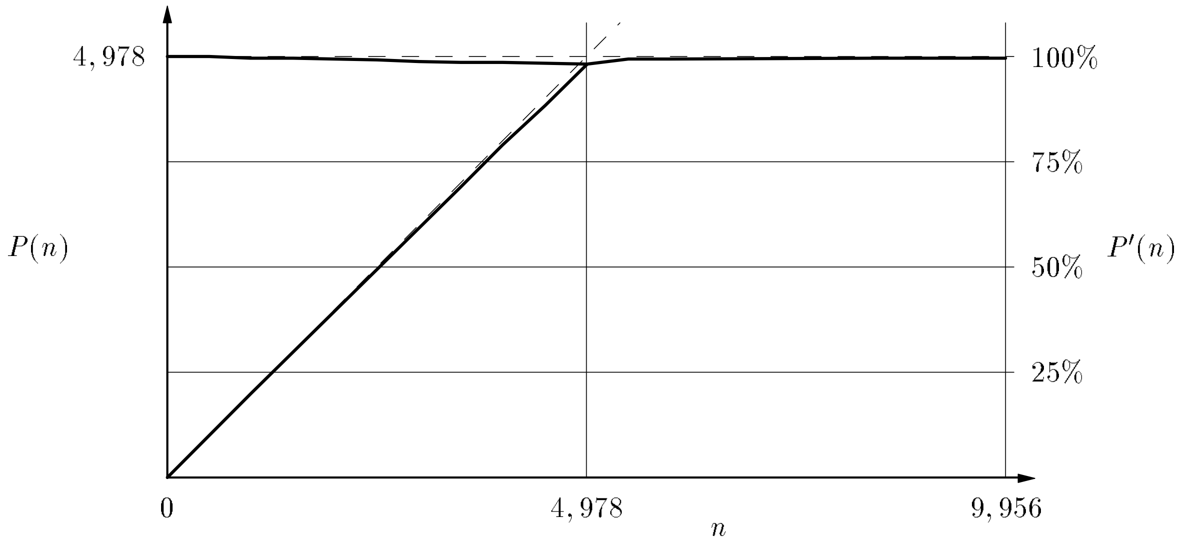

Figure 9: Smoothability profile for `fpppp (4)` ($S = 98.2\%$)

and the computation load is fairly evenly distributed over time. Thus, there is not much peak parallelism, and limiting the processor count only reduces parallelism significantly when the count drops below $P(\infty)$.

- The symbolic applications `tex` and `li` are only slightly less smoothable than `fpppp`.[6] This may be due to their relatively low values of $P(\infty)$. It is easier to achieve maximal parallelism when the maximum is lower.

[6]While the smoothability (S) is slightly higher for `tex` than for `fpppp` (4), note that $P(n)$ rises more slowly for $n > P(\infty)$.

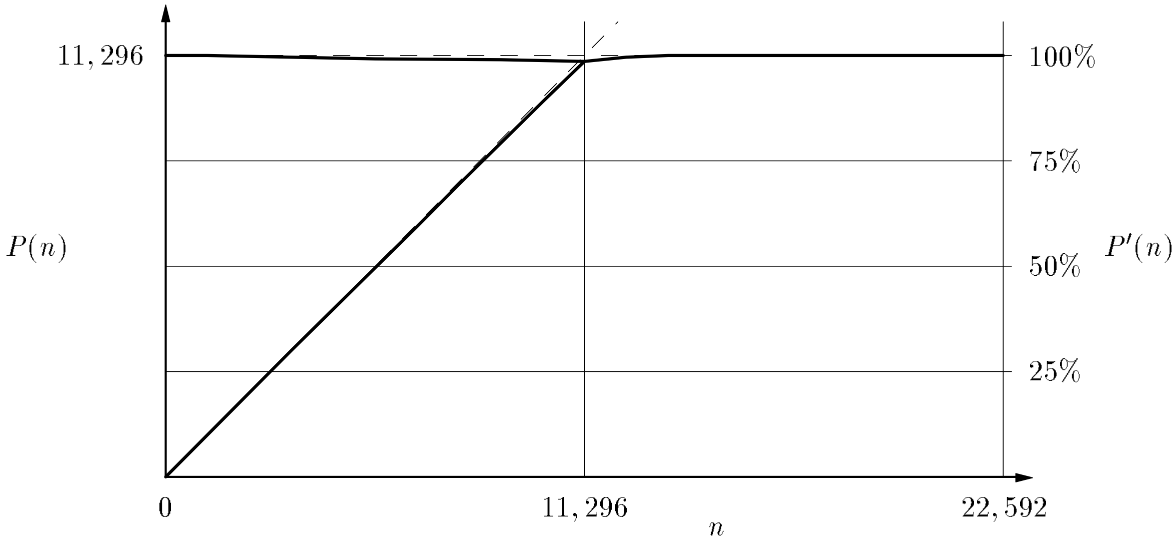

Figure 10: Smoothability profile for `fpppp (6)` ($S = 98.7\%$)

- The other two programs are less smoothable. Even so, in both cases the processor utilization below $n = P(\infty)$ is always at least 60%.

5 Latency

Up to this point, all experiments have assumed that all operations, including memory accesses, take the same amount of time to execute. While it may be acceptable to include local memory operations within this assumption, it won't be true for remote memory operations. Physical constraints will always ensure that operations requiring transmissions over long distances will take longer than operations that can be performed entirely within a processor. Parallel machines will require some kind of communication between processors, and this communication will take longer than a single clock cycle.

Therefore, we wish to measure the effects such latencies will have on parallelism. First, we propose a latency model to be used in our experiments. We then present the results of these experiments involving both latency and finite processors.

5.1 Latency Model

Ordinarily, when a load instruction appears in the instruction trace, SITA schedules the instruction into the earliest possible PI, subject to dependences and constraints, and the register(s) loaded can be read as early as the following PI. Similarly, when a store occurs, the memory location(s) written to can be read as early as the following PI. In our extended model, we compute the latency l for a given load or store operation. If a load or store is scheduled in PI_i, then any operation reading the register or memory location written by the load/store can't be scheduled any earlier than PI_{i+l}.

In a distributed-memory machine, some memory accesses will refer to local items or items in a local cache, and we assume these will only take 1 cycle, like the other instructions. Some fraction of the accesses will refer to items in other processors, and will have a higher latency. Determining which references are local and which are remote is problematic, because it depends on having exact information about how memory objects are distributed among the processors. Since partitioning is beyond the scope of this project, we have chosen a probabilistic model for determining misses. A "miss rate" is chosen prior to running an experiment, and then the analyzer chooses random elements from the instruction stream to be "misses", so that the average miss rate equals the chosen parameter.[7]

When a miss occurs, the analyzer assigns a latency greater than 1 to the memory access. Since both hypercubes and multistage interconnection networks, two popular networks for parallel processors, have average routing distance proportional to $\log n$, we assume in most cases that the latency is $\lceil \log_2 n \rceil$ cycles. For instance, if there are 1,024 processors, the latency for any remote access is 10 cycles. Because the network may not be able to attain this high speed, we assume a higher latency of $10\lceil \log_2 n \rceil$ cycles in some experiments.

5.2 Latency Experiments

For the final 231 experiments, we added both the latency model (described in the previous subsection) and limits on processor count to the Omniscient Oracle model. Each of the six test cases was run with various processor counts, latencies, and miss rates. For each case, five to seven processor counts were chosen; all were powers of 2 in the vicinity of $P(\infty)$ (see the Omniscient Oracle results in Table 3).

For each chosen processor count, seven tests were run. For the first six, the (miss) memory latency was equal to the log of the processor count, and the miss rates were {0%, 10%, 20%, 30%, 60%, 100%}. A seventh test was run in which latency was increased by a factor of 10 (i.e., equal to $10 \log_2 n$) and a miss rate of 10% was assumed. We expected the results to be similar to the results for the shorter-latency network with a 100% miss rate, since the average latency per memory access would be the same.

[7] The term "miss" implies a shared-memory model. It is not entirely an accurate term for a message-passing model, but the choice of memory model is too specific a detail for this study. Therefore, in this section, a "miss" means either a remote memory access or the sending of a value to another processor via a message.

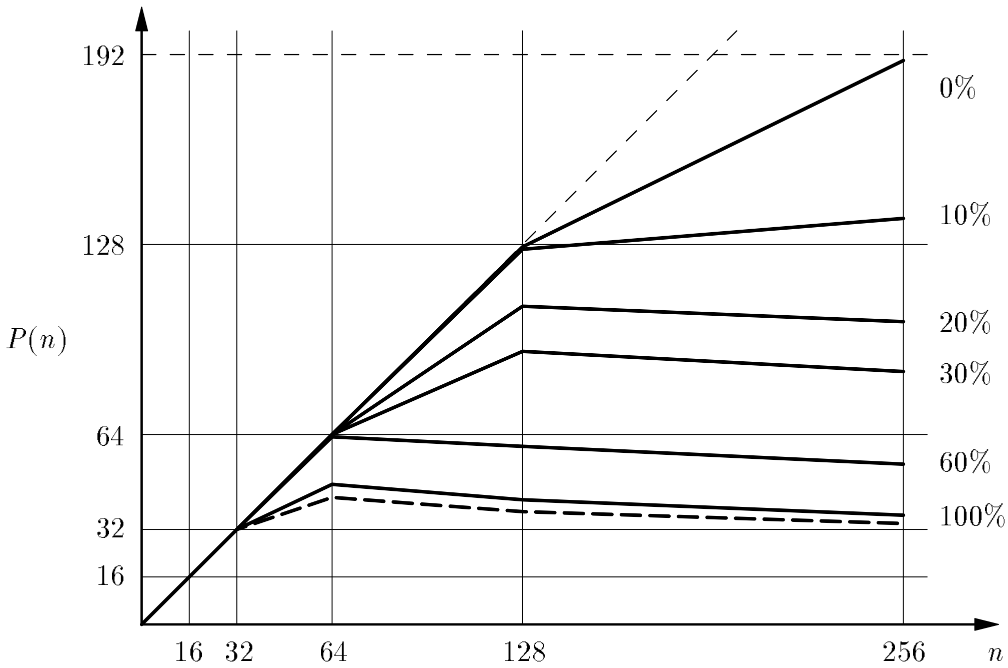

Figure 11: Latency-Smoothability profile for tex (16–256 processors)

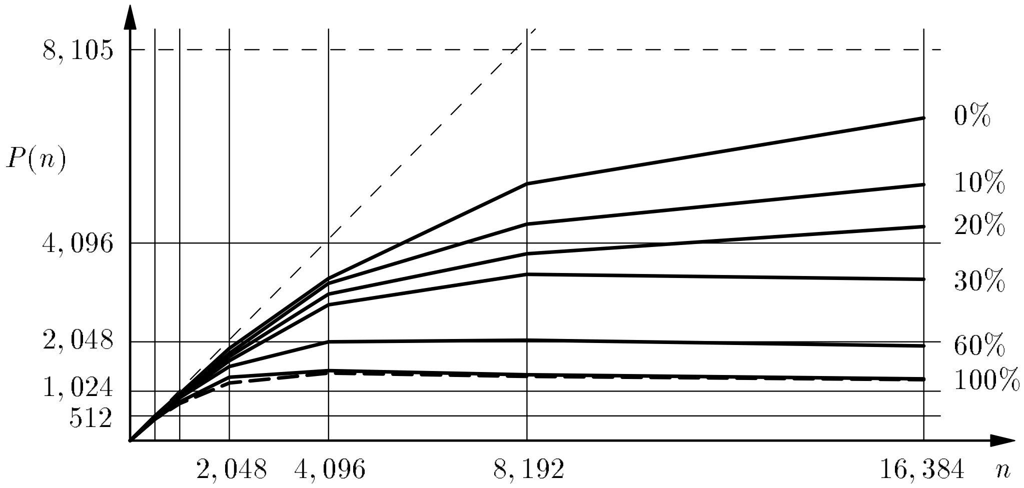

Figure 12: Latency-Smoothability profile for speech (512–16K processors)

Figures 11–16 show the results of the experiments. Each graph shows the parallelism curves for the six miss rates with short latency (labeled with the miss rates) and the single long-latency experiment (thick dashed line). Thin dashed lines show the upper bounds $P(n) = n$ and $P(n) = P(\infty)$.

What can be seen from all the graphs is that for a given miss rate, the parallelism curve looks similar to the parallelism curves shown in Section 4. Parallelism initially increases

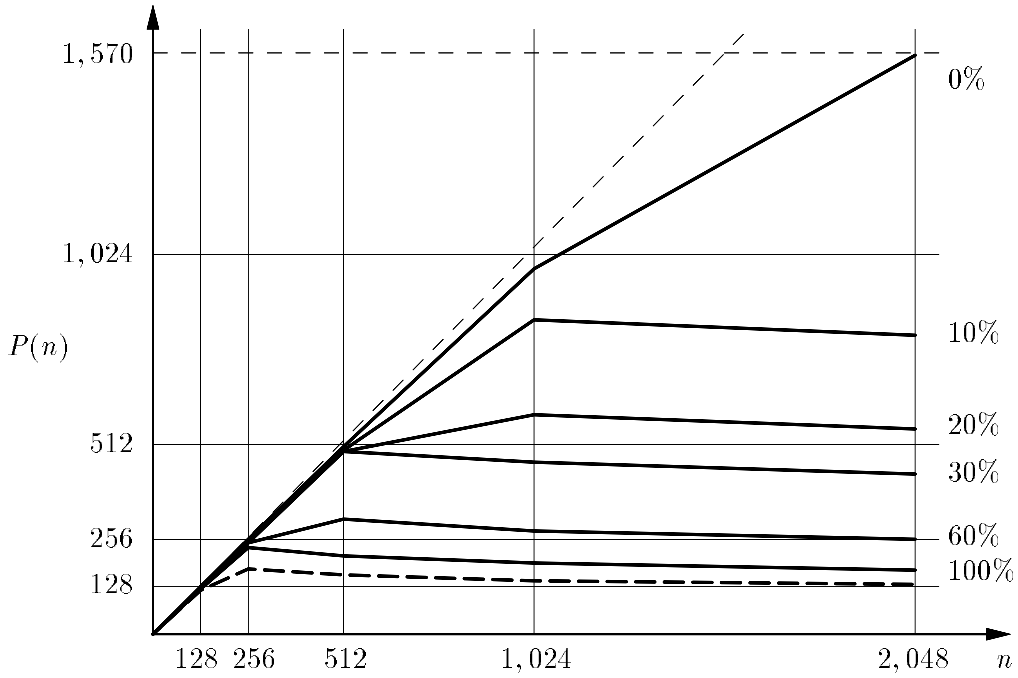

Figure 13: Latency-Smoothability profile for li (128–2K processors)

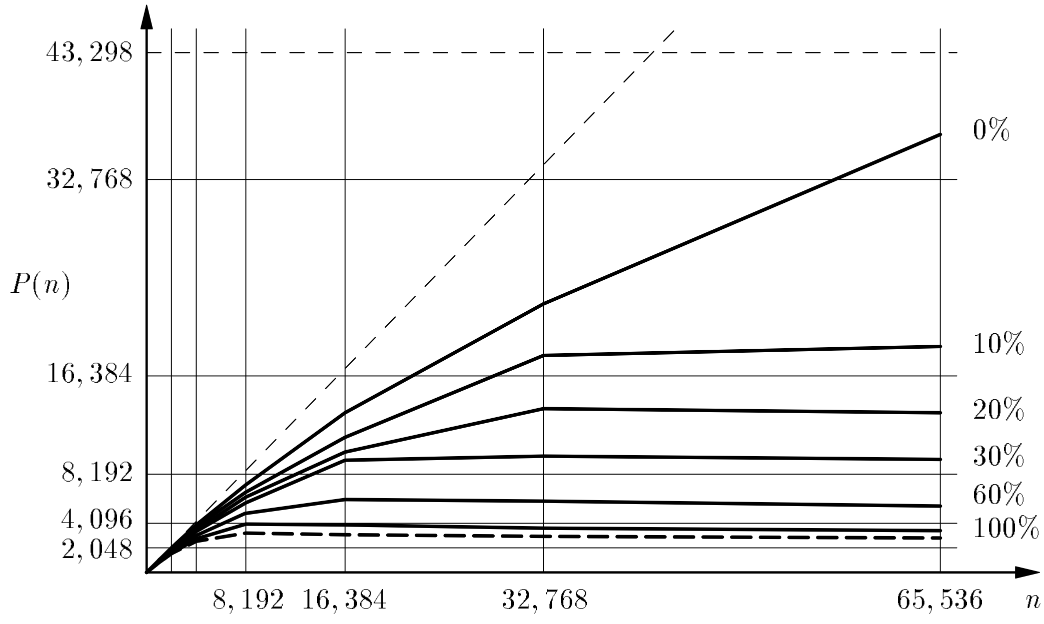

Figure 14: Latency-Smoothability profile for eqntott (1K–64K processors)

linearly, then tapers off as it approaches an asymptotic value. This is the point where adding more processors doesn't decrease execution time significantly, because the limit of program parallelism has been reached. In the experiments with non-zero miss rates, it can be seen that parallelism slowly declines after reaching a maximum. This is an effect of the latency model:

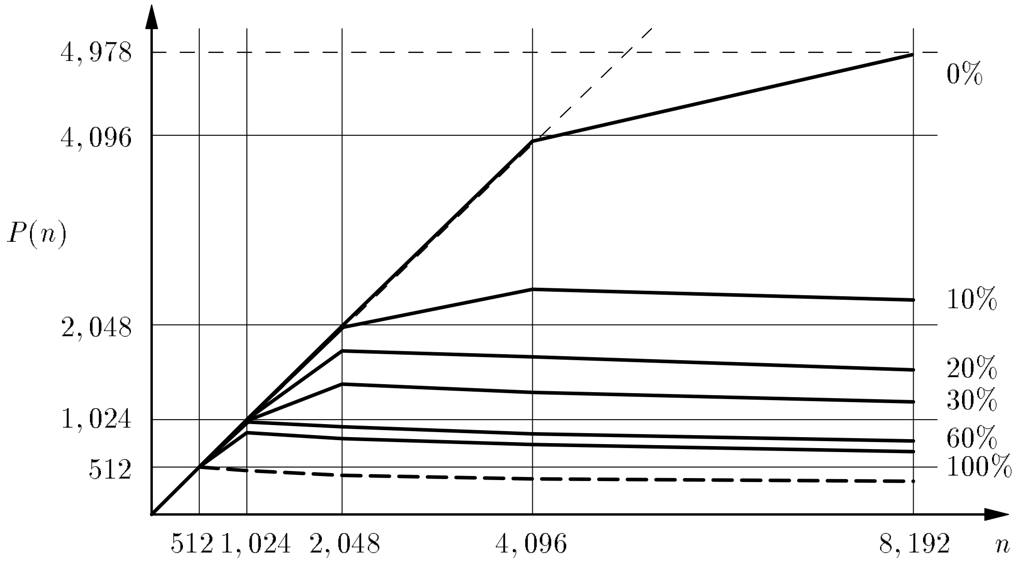

Figure 15: Latency-Smoothability profile for fpppp (4) (512–8K processors)

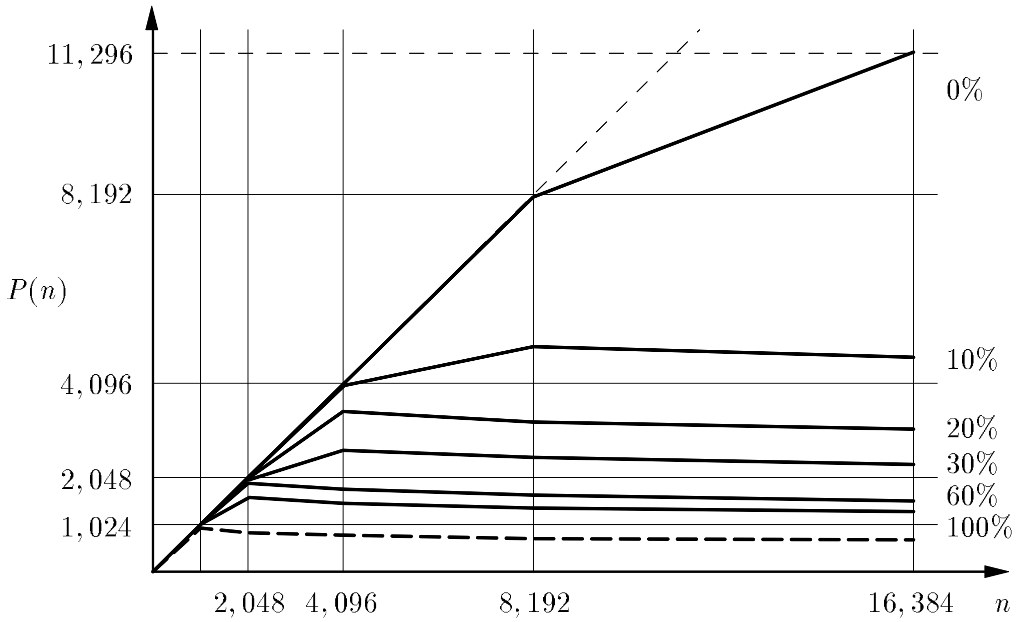

Figure 16: Latency-Smoothability profile for fpppp (6) (1K–16K processors)

miss latency increases by 1 cycle every time the number of processors is doubled.

In all cases, the long-latency model with a 10% miss rate was worse than the short-latency, 100%–miss model, and the relative difference depended on the application. There was almost no difference with `speech`, while `fpppp` showed a significant difference.

The asymptote (maximum parallelism) decreases as the miss rate increases, which shows that increasing memory latency will cause the critical path(s) to lengthen. This is to be expected. When a processor has to wait for a remote access, it can fill in the gap by executing other instructions. However, these other instructions can be executed on a separate processor on a machine without long-latency accesses. Thus, filling in latency gaps by executing other instructions will decrease the overall parallelism. As the widths of the gaps increase, more instructions must be diverted from other processors to fill in the gaps, which decreases available parallelism still further.

What is encouraging, however, is the behavior of each parallelism curve near its "knee" where it first approaches its asymptote. Below the knee, the experiments shows that the processors could achieve good utilization. This shows that the execution of different instructions (i.e., a context switch) to hide memory latencies can be an effective means of improving overall processor utilization.

6 Previous Studies

Most previous studies on the limits of program 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, Muraoka and Chen [10], 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 [21]. 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 [13]. 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 dependence which was not a "true" data dependence. They were able to obtain respectable speedups from small scientific routines.

Kumar [11] 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. Kumar found the potential for high levels of parallelism in regular numerical code.

Wall [22] analyzed benchmarks (SPEC89 programs, system utilities, 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 scheduler used only 64 processors, a finite (2K) input window from which to fill parallel instructions, and no memory renaming (register renaming only).

Two recent studies measured the effects of limiting the size of the scheduling window, which determines how broad the search for parallelism in the sequential trace can be, on the parallelism available in SPEC89 benchmarks. Butler, et al [3], ran each trace for 10 million instructions on an M88000 processor, while Austin and Sohi [2] ran each trace for up to 100 million instructions on a MIPS processor. Both assumed perfect (oracular) branch prediction, but limited how far apart two instructions could be in the sequential trace and still be scheduled in the same parallel instruction. They both concluded that instructions must be drawn from regions far apart in order to achieve significant amounts of parallelism.

Wilson and Lam studied the way control flow limits parallelism [12]. They 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 performing speculative execution. They tested six SPEC89 programs and 4 other programs, running each program on a MIPS R3000 for up to 100 million instructions.

7 Conclusions

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

We have reported on the construction of SITA, a testbed that was used to work with a wide variety of architecture models. SITA was designed to be able to perform experiments on complete traces for reasonably large benchmarks. This allowed us to experiment with both standard benchmark programs and a real speech application.

By examining an Omniscient Oracle and specializations of this oracle, we were able to study key features that are important in finding large amounts of fine-grain parallelism. These results 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 for some of the irregular programs under the Fine Dataflow model. Other studies have seen similar results. However, some applications, like `tex`, intuitively seem to have much more potential parallelism than found by our Fine Dataflow model (5.31) or Lam and Wilson's similar Multithreaded 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 Fine Dataflow model gets poor results, even though it eliminates false data dependences, 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, 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.

Our experiments also showed that restricting the Oracle by making it frugal (no memory renaming) resulted in reducing the parallelism dramatically. As part of this set of tests, 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.

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 60%. We also found that while adding memory latency reduces the upper bound on parallelism, the parallelism obtained is still quite smoothable. Thus, there is good hope for mapping the available parallelism onto finite resources, and that multithreading or dataflow scheduling of operations can be used to tolerate the latencies of remote memory accesses.

Our analyzer has proven to be a valuable experimental tool, and many useful extensions are possible. For future work, we plan to extend SITA in several ways. We plan to add ordering constraints to the scheduler, so that we can measure the effects of various multithreading scheduling strategies. Also, we want to connect the testbed to our McCAT compiler [8] so that we can add parameters relating to the real alias analyses that can be performed by a parallelizing compiler.

8 Acknowledgment

The authors would like to thank Gordon Irlam for making his SPA (Sparc Performance Analyzer) tools publicly available, and for giving technical assistance. The Spy trace generator, part of the SPA package, was an essential link in our experimental testbed.

References

[1] Arvind and Kim P. Gostelow, "The U-Interpreter," *Computer*, 15(2):42–49, February 1982.

[2] Todd M. Austin and Gurindar S. Sohi, "Dynamic Dependence Analysis of Ordinary Programs," in *Proceedings of the 19th Annual International Symposium on Computer Architecture*, Gold Coast, Australia, pp. 342–351, May 19–21, 1992.

[3] Michael Butler, Tse-Yu Yeh, Yale Patt, Mitch Alsup, Hunter Scales, and Michael Shebanow, "Single Instruction Stream Parallelism Is Greater than Two," in *Proceedings of the 18th Annual International Symposium on Computer Architecture*, Toronto, Ontario, pp. 276–286, May 28–30, 1991.

[4] David E. Culler, Anurag Sah, Klaus Erik Schauser, Thorsten von Eicken, 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*, Santa Clara, California, pp. 164–175, April 8–11, 1991. *Computer Architecture News,* 19(2), April 1991.

[5] Jack B. Dennis and Guang R. Gao, "An Efficient Pipelined Dataflow Processor Architecture," in *Proceedings of Supercomputing '88*, Orlando, Florida, pp. 368–373, IEEE Computer Society and ACM SIGARCH, November 14–18, 1988.

[6] John T. Feo, David C. Cann, and Rodney R. Oldehoeft, "A Report on the Sisal Language Project," *Journal of Parallel and Distributed Computing*, 10(4):349–366, December 1990.

[7] Jean-Luc Gaudiot and Lubomir Bic, eds., *Advanced Topics in Data-Flow Computing.* Englewood Cliffs, New Jersey: Prentice-Hall, 1991. Book contains papers presented at the

First Workshop on Data-Flow Computing, held in conjunction with the 16th Annual International Symposium on Computer Architecture in Eilat, Israel, May 1989.

[8] L. Hendren, C. Donawa, M. Emami, G. Gao, Justiani, and B. Sridharan, "Designing the McCAT Compiler Based on a Family of Structured Intermediate Representations," in *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing* (Uptal Banerjee, David Gelernter, Alex Nicolau, and David Padua, eds.), no. 757 in Lecture Notes in Computer Science, New Haven, Connecticut, pp. 406–420, August 3–5, 1992. Berlin: Springer-Verlag. Published in 1993.

[9] Robert A. Iannucci, Guang R. Gao, Robert H. Halstead, Jr., and Burton Smith, eds., *Multithreaded Computer Architecture: A Summary of the State of the Art.* Norwell, Massachusetts: Kluwer Academic Publishers, to appear in 1994. Book contains papers presented at the Workshop on Multithreaded Computers, Albuquerque, New Mexico, November 1991.

[10] David J. Kuck, Yoichi Muraoka, and Shyh-Ching Chen, "On the Number of Operations Simultaneously Executable in Fortran-Like Programs and Their Resulting Speedup," *IEEE Transactions on Computers*, 21(12):1293–1310, December 1972.

[11] Manoj Kumar, "Measuring Parallelism in Computation-Intensive Scientific/Engineering Applications," *IEEE Transactions on Computers*, 37(9):1088–1098, September 1988.

[12] Monica S. Lam and Robert P. Wilson, "Limits of Control Flow on Parallelism," in *Proceedings of the 19th Annual International Symposium on Computer Architecture*, Gold Coast, Australia, pp. 46–57, May 19–21, 1992.

[13] Alexandru Nicolau and J. A. Fisher, "Measuring the Parallelism Available for Very Long Instruction Word Architectures," *IEEE Transactions on Computers*, 33(11):968–976, November 1984.

[14] Rishiyur S. Nikhil and Arvind, "Can Dataflow Subsume Von Neumann Computing?," in *Proceedings of the 16th Annual International Symposium on Computer Architecture*, Jerusalem, Israel, pp. 262–272, May 28–June 1, 1989.

[15] Brett Olsson, Robert Montoye, Peter Markstein, and MyHong NguyunPhu, "RISC System/6000 Floating-Point Unit," in *IBM RISC System/6000 Technology* (Mamata Misra, ed.), pp. 34–42, IBM Corp., first ed., 1990. Order No. SA23-2619.

[16] James Rumbaugh, "A Data Flow Multiprocessor," *IEEE Transactions on Computers*, 26(2):138–146, February 1977.

[17] James E. Smith, "A Study of Branch Prediction Strategies," in *Proceedings of the 8th Annual Symposium on Computer Architecture*, Minneapolis, Minnesota, pp. 135–148, May 12–14, 1981.

[18] Kevin B. Theobald, Guang R. Gao, and Laurie J. Hendren, *On the Limits of Program Parallelism and its Smoothability,* ACAPS Technical Memo 40, School of Computer Science, McGill University, Montréal, Québec, June 1992.

[19] Kevin B. Theobald, Guang R. Gao, and Laurie J. Hendren, "On the Limits of Program Parallelism and its Smoothability," in *Proceedings of the 25th Annual International Sympo-*

sium on Microarchitecture, Portland, Oregon, pp. 10–19, ACM SIGMICRO and IEEE-CS TC-MICRO, December 1–4, 1992.

[20] Kevin B. Theobald, Guang R. Gao, and Laurie J. Hendren, *Speculative Execution and Branch Prediction on Parallel Machines,* ACAPS Technical Memo 57, School of Computer Science, McGill University, Montréal, Québec, December 1992.

[21] Garold S. Tjaden and Michael J. Flynn, "Detection and Parallel Execution of Independent Instructions," *IEEE Transactions on Computers*, 19(10):889–895, October 1970.

[22] David W. Wall, "Limits of Instruction-Level Parallelism," in *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, California, pp. 176–188, April 8–11, 1991. *Computer Architecture News,* 19(2), April 1991.

A Experimental Testbed

The SITA analyzer consists of two parts. The first part, the decoder, takes the stream output generated by the Spy trace generator, decodes the opcode and other information, performs basic block flow analysis, and derives 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. Figure 17 shows what the table would look like after the first iteration of the loop in Section 2.1, assuming it is scheduled as shown in the left half of Figure 2. The Sparc processor has multiple register windows, so SITA has an entry for each *physical* register and performs the mapping at run time. (The number of physical register windows can be selected by the user.) In this example, window registers 9–12 (`%o1`–`%o4`) are mapped to physical registers 73–76. Furthermore, it is assumed in this example that the array base addresses (`%o2`–`%o4`) are created in PI_{i-1}.

When the second occurrence of S1 is encountered in the trace, the decoder will see that it is dependent on writing FP register 0, and reading physical registers 73 and 74 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}. However,

Register	*Read*	*Write*	*Register*	*Read*	*Write*
⋮	⋮	⋮	⋮	⋮	⋮
Int. register 73	$i+3$	$i+3$	Int. cond. codes	$i+3$	$i+3$
Int. register 74	$i-1$	i	FP register 0	$i+1$	$i+1$
Int. register 75	$i-1$	i	FP register 1	$i+1$	$i+1$
Int. register 76	$i-1$	$i+2$	FP register 2	$i+2$	$i+2$
⋮	⋮	⋮	⋮	⋮	⋮

Figure 17: Example state of SITA scheduler

since there is no branch prediction with this example, the operation must 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 columns for FP register 0 are updated with $i+5$, and the write columns for physical registers 73 and 74 are changed to $i+4$.

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 complex dynamic data structure consisting of a hash table of linked lists of tables corresponding to individual pages of memory. The lower bits of the page number are used to hash the entries. 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.

If one of the dataflow models is used, then barriers corresponding to conditional branches are kept on a stack. As conditional branches are scheduled, barriers are added to the stack and remain there until all dependent blocks have been executed, after which they are removed. A stack is sufficient to keep all live barrier information [20]. If branch prediction is used, the 2-bit count for that branch is used to derive the *predicted* branch outcome, which is compared to the *actual* outcome revealed in the trace. If prediction is correct, the barrier is marked as such. When an instruction is to be scheduled, the stack is searched top-down for the first barrier, which corresponds to a mispredicted branch on which it is control-dependent.

A finite processor count requires a table of how many operations are packed into each PI. Two-level paging is used to organize the table. For very long runs, the page table can become huge, so SITA performs optimizations such as freeing up the storage whenever all the PI's on a page are filled to their limits.

B Tables of Smoothability Results

k	n	$P(n)$	$P'(n)$	k	n	$P(n)$	$P'(n)$	k	n	$P(n)$	$P'(n)$
0.1	19	19.000	100.000	0.9	174	172.665	99.233	1.7	328	190.756	99.320
0.2	39	38.999	99.997	1.0	193	188.739	98.270	1.8	347	190.866	99.377
0.3	58	57.993	99.988	1.1	212	189.284	98.554	1.9	367	190.975	99.434
0.4	77	76.985	99.981	1.2	232	189.727	98.784	2.0	386	191.062	99.479
0.5	97	96.972	99.971	1.3	251	190.024	98.938	2.1	405	191.146	99.523
0.6	116	115.796	99.824	1.4	270	190.264	99.064	2.2	425	191.223	99.563
0.7	135	134.496	99.627	1.5	290	190.455	99.163	2.3	444	191.296	99.601
0.8	154	153.136	99.439	1.6	309	190.621	99.250	2.4	463	191.354	99.631
								∞	∞	192.062	100

Table 5: Smoothability experiments for `tex`

k	n	$P(n)$	$P'(n)$	k	n	$P(n)$	$P'(n)$	k	n	$P(n)$	$P'(n)$
0.1	811	789.905	97.399	1.7	13779	6379.380	78.705	10.0	81050	7941.165	97.979
0.2	1621	1536.374	94.779	1.8	14589	6470.583	79.835	15.0	121575	8008.422	98.809
0.3	2432	2221.125	91.329	1.9	15400	6554.207	80.866	20.0	162100	8037.028	99.162
0.4	3242	2811.142	86.710	2.0	16210	6631.234	81.817	25.0	202625	8054.407	99.376
0.5	4053	3344.991	82.531	2.5	20263	6939.073	85.615	30.0	243150	8075.882	99.641
0.6	4863	3829.180	78.741	3.0	24315	7170.869	88.475	35.0	283675	8083.579	99.736
0.7	5674	4272.400	75.298	3.5	28368	7342.502	90.593	40.0	324200	8086.781	99.776
0.8	6484	4677.826	72.144	4.0	32420	7474.614	92.223	50.0	405250	8091.291	99.831
0.9	7295	5039.793	69.086	4.5	36473	7587.472	93.615	60.0	486300	8094.262	99.868
1.0	8105	5292.356	65.298	5.0	40525	7670.987	94.645	70.0	567350	8096.401	99.894
1.1	8916	5517.919	68.081	5.5	44578	7741.220	95.512	80.0	648400	8097.947	99.913
1.2	9726	5714.988	70.512	6.0	48630	7795.407	96.181	90.0	729450	8099.256	99.929
1.3	10537	5889.424	72.664	6.5	52683	7823.951	96.533	100.0	810500	8100.207	99.941
1.4	11347	6045.795	74.594	7.0	56735	7845.440	96.798	200.0	1621000	8104.852	99.999
1.5	12158	6169.550	76.121	8.0	64840	7887.535	97.317	300.0	2431500	8104.971	100
1.6	12968	6279.463	77.477	9.0	72945	7912.782	97.629	∞	∞	8104.971	100

Table 6: Smoothability experiments for `speech`

This appendix contains the raw parallelism data, in tabular form, for both the smoothability experiments of Section 4 and the latency experiments of Section 5. Tables 5–16 contain all the data plotted in Figures 5–16, plus data from additional experiments not included in the figures. In Tables 5–10, $P'(n)$ is listed as a percentage.

Smoothability experiments were run with processor counts greater than $2\lceil P(\infty)\rceil$ on `tex` and the less-smoothable applications `speech` and `eqntott` to see how high the processor count would have to be before getting close to the maximum parallelism. Extra measurements were taken in the vicinity of $n = P(\infty)$ for `fpppp (4)`.

Tables 6 and 8 show an interesting contrast: even though `eqntott` is less smoothable than `speech`, it is possible to get almost all available parallelism (99.99%) from `eqntott` with $6P(\infty)$ processors, while it takes more than $100P(\infty)$ processors to get the same percentage from `speech` (in the latter, n_{max} is between $200P(\infty)$ and $300P(\infty)$). This may be due to the presence of a critical path in `speech` lying near the end of the sequential trace. SITA's scheduler is greedy; it does not look for the most optimal schedule (which would be a difficult

k	n	$P(n)$	$P'(n)$	k	n	$P(n)$	$P'(n)$	k	n	$P(n)$	$P'(n)$
0.1	157	156.438	99.642	0.8	1256	1201.463	95.658	1.5	2355	1567.674	99.886
0.2	314	311.111	99.080	0.9	1413	1343.466	95.079	1.6	2512	1568.478	99.937
0.3	471	464.124	98.540	1.0	1570	1489.310	94.893	1.7	2669	1568.934	99.966
0.4	628	615.473	98.005	1.1	1727	1558.754	99.318	1.8	2826	1569.294	99.989
0.5	785	764.747	97.420	1.2	1884	1561.962	99.522	1.9	2983	1569.354	99.993
0.6	942	912.137	96.830	1.3	2041	1564.311	99.672	2.0	3140	1569.390	99.995
0.7	1099	1057.622	96.235	1.4	2198	1566.092	99.785	∞	∞	1569.463	100

Table 7: Smoothability experiments for `li`

k	n	$P(n)$	$P'(n)$	k	n	$P(n)$	$P'(n)$	k	n	$P(n)$	$P'(n)$
0.1	4330	4086.956	94.387	1.3	56287	33184.399	76.643	2.5	108245	42112.164	97.286
0.2	8660	7724.275	89.195	1.4	60617	34842.839	80.473	2.6	112575	42225.871	97.525
0.3	12989	10980.027	84.533	1.5	64947	36345.761	83.945	2.7	116905	42338.090	97.785
0.4	17319	13918.984	80.368	1.6	69277	37745.315	87.177	2.8	121234	42442.452	98.026
0.5	21649	16582.353	76.596	1.7	73607	39051.917	90.195	2.9	125564	42528.222	98.224
0.6	25979	19014.760	73.193	1.8	77936	40234.527	92.926	3.0	129894	42627.127	98.452
0.7	30309	21246.618	70.100	1.9	82266	41216.118	95.193	3.5	151543	42864.016	98.999
0.8	34638	23302.068	67.273	2.0	86596	41407.345	95.635	4.0	173192	43012.192	99.341
0.9	38968	25203.167	64.677	2.1	90926	41573.977	96.020	4.5	194841	43113.364	99.575
1.0	43298	27087.928	62.563	2.2	95256	41737.866	96.398	5.0	216490	43204.060	99.785
1.1	47628	29216.649	67.479	2.3	99585	41885.549	96.739	6.0	259788	43292.939	99.990
1.2	51958	31134.542	71.909	2.4	103915	41983.552	96.966	∞	∞	43297.338	100

Table 8: Smoothability experiments for `eqntott`

k	n	$P(n)$	$P'(n)$	k	n	$P(n)$	$P'(n)$	k	n	$P(n)$	$P'(n)$
0.1	498	497.181	99.836	0.944	4700	4621.645	98.333	1.105	5500	4956.354	99.576
0.2	996	992.547	99.653	0.964	4800	4717.711	98.286	1.125	5600	4956.708	99.583
0.3	1493	1485.171	99.476	0.984	4900	4814.253	98.250	1.2	5974	4958.042	99.610
0.4	1991	1976.874	99.290	0.994	4950	4862.527	98.233	1.3	6471	4960.085	99.651
0.5	2489	2467.182	99.123	1.000	4977	4888.711	98.217	1.4	6969	4961.596	99.681
0.6	2987	2955.236	98.937	1.0	4978	4889.747	98.238	1.5	7467	4962.663	99.703
0.7	3485	3441.678	98.757	1.000	4979	4890.611	98.255	1.6	7965	4963.998	99.729
0.8	3982	3926.110	98.596	1.005	5000	4910.908	98.662	1.7	8463	4964.709	99.744
0.864	4300	4235.059	98.490	1.025	5100	4954.137	99.542	1.8	8960	4965.600	99.762
0.884	4400	4331.314	98.439	1.045	5200	4954.668	99.547	1.9	9458	4966.401	99.778
0.9	4480	4408.552	98.405	1.065	5300	4954.935	99.547	2.0	9956	4967.114	99.792
0.904	4500	4428.220	98.405	1.085	5400	4955.733	99.563	∞	∞	4977.469	100
0.924	4600	4524.455	98.358	1.1	5476	4956.001	99.569				

Table 9: Smoothability experiments for `fpppp (4)`

task!) but merely schedules each operation as soon as it appears in the trace. Thus, if a critical path appears near the end of the trace, it may not be possible to schedule it as early as it could legally run since the scheduler will have filled the early PI's with other operations already.

Latency experiments were run on some miss rates not included in Figures 11–16. Resource limitations made it impossible to run a complete set of experiments on all data points,

k	n	$P(n)$	$P'(n)$	k	n	$P(n)$	$P'(n)$	k	n	$P(n)$	$P'(n)$
0.1	1130	1128.582	99.875	0.8	9037	8943.125	98.961	1.5	16944	11290.489	99.952
0.2	2260	2254.188	99.743	0.9	10166	10046.183	98.821	1.6	18074	11291.098	99.958
0.3	3389	3375.975	99.616	1.0	11296	11150.735	98.715	1.7	19203	11291.604	99.962
0.4	4518	4494.800	99.487	1.1	12426	11287.753	99.928	1.8	20333	11291.706	99.963
0.5	5648	5611.391	99.352	1.2	13555	11288.665	99.936	1.9	21462	11292.111	99.967
0.6	6778	6725.806	99.230	1.3	14685	11289.375	99.943	2.0	22592	11292.518	99.970
0.7	7907	7835.410	99.095	1.4	15814	11289.982	99.948	∞	∞	11295.863	100

Table 10: Smoothability experiments for `fpppp (6)`

Miss rate	$n = 16$	$n = 32$	$n = 64$	$n = 128$	$n = 256$
0.0	16.000	32.000	63.991	127.615	190.088
0.1	16.000	32.000	63.985	126.969	136.598
0.2	16.000	32.000	63.937	107.848	101.917
0.3	16.000	32.000	63.810	92.171	85.426
0.4	16.000	32.000	63.664	76.972	67.428
0.5	16.000	32.000	63.515	73.105	53.993
0.6	16.000	32.000	63.374	60.060	54.122
0.7	16.000	32.000	61.325	54.876	45.823
0.8	15.999	32.000	55.528	49.292	41.990
0.9	15.999	31.984	50.799	44.821	38.874
1.0	15.999	31.954	47.711	41.932	37.206
(Slow) 0.1	16.000	31.840	42.409	37.798	33.931

Table 11: Latency-smoothability experiments for `tex` ($P(\infty) = 192.062$)

Miss rate	$n = 512$	$n = 1024$	$n = 2048$	$n = 4096$	$n = 8192$	$n = 16384$	$n = 32768$
0.0	503.833	990.164	1913.939	3371.855	5317.475	6669.188	7515.354
0.1	499.090	969.840	1828.557	3262.789	4502.167	5319.903	5682.945
0.2	494.335	950.388	1754.596	3049.013	3883.617	4426.835	4264.103
0.3	488.314	925.034	1663.923	2808.993	3443.917	3346.003	3168.694
0.4		914.545	1624.089	2511.082	2923.133		
0.5		875.581	1491.001	2320.893	2565.862		
0.6	477.715	882.961	1528.654	2062.102	2095.772	1967.023	1847.657
0.7		867.519	1488.115	1867.438	1829.060		
1.0	462.308	824.482	1300.850	1467.211	1358.532	1270.213	1188.266
(Slow) 0.1	447.255	768.399	1194.379	1406.184	1337.771	1251.339	

Table 12: Latency-smoothability experiments for `speech` ($P(\infty) = 8,104.971$)

particularly for the longer benchmarks, but we include the data from all experiments which were run. The additional results are consistent with the observations made in Section 5.

There are a few instances, in which n is small compared to $P(\infty)$, where increasing the miss rate actually increases parallelism slightly. This anomaly is due to the random latency model used by SITA (see Section 5.1). When the miss rate changes, the determination of which accesses are "remote" changes as well. This adjustment can cause accesses on critical paths to change from "remote" to "local," thereby decreasing critical path length. Note, however, that these anomalies only occur with low processor counts and are statistically insignificant.

Miss rate	$n = 128$	$n = 256$	$n = 512$	$n = 1024$	$n = 2048$
0.0	127.674	254.174	503.838	988.307	1564.418
0.1	127.384	252.907	498.124	849.166	805.627
0.2	127.214	252.141	493.816	592.679	552.944
0.3	127.085	251.574	493.228	465.989	431.616
0.6	126.310	247.625	309.852	281.363	257.657
1.0	125.418	234.071	210.436	190.090	173.172
(Slow) 0.1	124.496	174.511	156.903	142.376	130.307

Table 13: Latency-smoothability experiments for `li` ($P(\infty) = 1,569.463$)

Miss rate	$n = 1024$	$n = 2048$	$n = 4096$	$n = 8192$	$n = 16384$	$n = 32768$	$n = 65536$
0.0	1010.593	1993.367	3878.235	7350.620	13309.308	22434.178	36530.359
0.1	1001.282	1952.634	3710.366	6717.064	11278.094	17903.188	18785.305
0.2	993.490	1919.453	3578.908	6269.714	10062.907	13629.079	13345.465
0.3	985.058	1882.254	3435.334	5829.063	9276.318	9635.227	9267.190
0.4	977.515	1852.020	3329.751	5515.308	8287.307	8164.575	7780.012
0.6	961.443	1788.682	3118.033	5002.366	6080.727	5854.408	5524.549
0.7	953.455	1756.945	3018.724	4808.196	5263.728	5024.083	4727.927
1.0	931.783	1678.274	2783.419	4034.194	3935.267	3713.534	3485.637
(Slow) 0.1	914.283	1620.626	2637.455	3278.215	3139.950	2957.760	2782.915

Table 14: Latency-smoothability experiments for `eqntott` ($P(\infty) = 43,297.338$)

Miss rate	$n = 256$	$n = 512$	$n = 1024$	$n = 2048$	$n = 4096$	$n = 8192$
0.0	255.812	511.135	1020.369	2033.179	4037.041	4964.173
0.1	255.783	510.960	1018.984	2027.833	2422.797	2316.448
0.2	255.809	510.600	1018.766	1762.864	1659.087	1564.847
0.3	255.765	510.219	1017.172	1408.061	1315.267	1232.992
0.4	255.720	510.411	1017.258	1212.868	1128.583	1054.551
0.5	255.791	510.249	1015.244	1071.876	994.832	927.487
0.6	255.754	511.148	1014.455	944.292	874.092	813.156
0.7	255.657	510.421	977.111	899.164	831.591	773.132
0.8	255.889	510.539	923.226	848.509	783.990	728.233
0.9	255.877	510.972	900.729	827.399	764.251	709.689
1.0	255.671	510.433	887.540	815.133	752.889	699.050
(Slow) 0.1	253.368	506.338	461.040	422.620	389.701	361.505

Table 15: Latency-smoothability experiments for `fpppp (4)` ($P(\infty) = 4,977.469$)

Miss rate	$n = 1024$	$n = 2048$	$n = 4096$	$n = 8192$	$n = 16384$
0.0	1022.839	2043.249	4076.563	8115.271	11290.084
0.1	1022.477	2041.541	4069.057	4879.888	4658.970
0.2	1022.253	2039.121	3487.409	3281.797	3098.294
0.3	1021.738	2038.390	2663.705	2490.699	2338.338
0.6	1021.244	1926.670	1779.524	1652.933	1542.881
0.7	1021.375	1786.559	1648.206	1529.458	
0.9		1662.291	1532.091		
1.0	1019.696	1639.982	1511.302	1401.097	1305.697
(Slow) 0.1	943.304	863.687	796.100	738.307	688.334

Table 16: Latency-smoothability experiments for `fpppp (6)` ($P(\infty) = 11,295.863$)