In the Proceedings of the 25th Annual International
Symposium on Microarchitecture (MICRO-25),
pp. 10–19, Portland, Oregon, December 1-4, 1992

On the Limits of Program Parallelism and its Smoothability

Kevin B. Theobald, Guang R. Gao, and Laurie J. Hendren
Advanced Computer Architecture and Program Structures Group
McGill University School of Computer Science
3480 University St., Montréal, Québec, Canada H3A 2A7

Abstract

There have been many recent studies of the "limits on instruction parallelism" in application programs. This paper reports a new study of instruction-level parallelism which examines aspects not covered in previous studies, including the effects of various memory reuse policies and long-latency operations, and the results achieved when large benchmarks are allowed to run to completion. We also define and study program smoothability*, which quantifies the extent to which deferring program operations from periods of peak parallelism increases execution time. The results show a high degree of smoothability, suggesting that processor utilization can be quite high when the number of processors is below the parallelism upper bound.*

Keywords: *Computer architecture modeling, Instruction-level parallelism, Parallel processing, Program parallelism, Program smoothability.*

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 based on advanced architectural ideas such as multithreaded architectures [1, 5, 13]. 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 some fundamental questions. First, how much parallelism theoretically exists in the target applications? Second, is the available parallelism sufficiently *smoothable* 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 also that it must be possible, when the number of processors is limited, to spread 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 a number of previous studies, 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, 6, 8, 9, 10, 11, 14, 17, 18]. Below we briefly mention those most relevant to our study; the readers are referred to [16] for a more complete survey. Riseman and Foster [14] modeled a machine with foreknowledge of conditional branch outcomes, which Nicolau and Fisher called the *oracle* model [11]; both papers reported respectable speedups from small programs. Kumar developed a tool that automatically measured parallelism figures from FORTRAN code, and found the potential for high levels of parallelism in scientific numerical programs [9]. Wall analyzed some large benchmarks using a series of models representing various levels of hardware and compiler capabilities [18]. Even under his "perfect" model, with oracle-like knowledge of branch outcomes, parallelism rarely exceeded 40, due to other limitations of the model. Lam and Wilson studied the way control flow limits parallelism [10]. They demonstrated that substantially higher parallelism can be achieved by relaxing the constraints imposed by control flow using control-dependence analysis and speculative execution. They also tested some large programs, running each for 100 million instructions.

Our research addresses some topics not thoroughly covered in the previous research:

- What happens when large programs are run to completion under the more ambitious models?
- What features are most important for exploiting parallelism? In particular, what are the effects of memory reuse in the heap and stack? What are the effects of non-uniform instruction latencies?
- How "smoothable" is the program parallelism?

To search for the limits of parallelism and measure its smoothability with different architectural ideas, we have developed an experimental testbed. This testbed is based on a trace simulation method similar to previous experiments [2, 10, 14, 18]. The tool allows us to analyze the execution of real code and calculates, under a given architectural model, how much parallelism could potentially be attained by 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 drop in performance as we selectively removed key features from the Oracle and moved it towards more realistic machine models at different levels. A total of 21 models were used in the study, ranging from the Omniscient Oracle down to more realistic superscalar machines. Four models were chosen to study the "smoothability" aspect.

Our oracle models, like the oracle in [11], ignore control dependencies entirely. While the results obtained from the oracle are an upper bound, and may be unrealistically high, we believe 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 Section 4. The final section discusses the conclusions that may be drawn from this study, and describes our plans for extending our tool.

2 Methodology and models

This section describes the trace simulation techniques and machine models used in our study. First we show conceptually how our analyzer reveals parallelism in a program, and how the various features of a model can affect potential parallelism. A simple loop is used for illustration. Next, we present the 21 basic models used in the experiments, and discuss what we can learn about exploiting parallelism in real computers from the experimental results.

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 sequence of executed instructions (*not* the static object code). The analyzer reads the operations from the stream and packs them into *parallel instructions* (PI).

As the analyzer reads each operation, it inserts it into the earliest PI possible, respecting the dependencies between that operation and all previous operations. There may be *flow*, *anti*, *output*, and *control* dependencies, depending on the model chosen. If the operation is *flow-dependent* or *output-dependent* [7] on a previous operation packed into PI_p, then the new operation can be scheduled no earlier than PI_{p+1}. If the operation is *anti-dependent* on a previous operation, the new operation can be scheduled in the same PI as the old operation.

In this study, we assume that memory has unlimited access ports, and that each access takes 1 clock cycle. Our *basic* models have infinitely many functional units, though we later limit the number to measure smoothability. In all models, unless otherwise stated, all operations require 1 clock cycle, and we also allow operations arbitrarily far apart in the trace to be packed into the same PI, so long as all other dependency constraints are obeyed. (Real machines may not be so capable, but this study aims at determining how much parallelism is truly inherent in a program, independent of any particular hardware limitations.)

To illustrate, consider this simple loop fragment:

```
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 scheduler would pack these operations into PI's if all dependencies were obeyed. The dashed line indicates a barrier caused by the conditional branch S6, which prevents all future operations from being scheduled before that barrier. One iteration can be initiated every 4 cycles, so the parallelism is 1.5.

The following subsections describe how these constraints may be modified in other models, and how such changes affect overall parallelism.

2.1.1 Memory disambiguation

Memory references can't always be determined at compile-time. For instance, a conservative compiler would have to assume that an anti-dependency *might* exist between S2 and S4, making it impossible to overlap separate iterations of the loop as in software

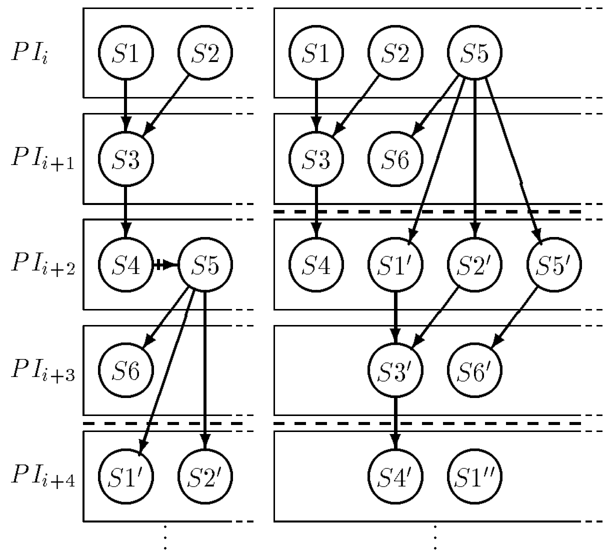

Figure 1: Packing parallel instructions: an example

pipelining, However, since memory addresses appear in the trace, the *analyzer* can optionally *disambiguate* them, i.e., determine at run-time whether two memory references really conflict. This models the potential effects of perfect compiler alias-analysis and/or special hardware to check for memory conflicts at run-time.

2.1.2 Register and memory renaming

A false dependency (anti or output) exists between two operations when one must follow the other merely because they are reusing a register or memory cell. An example of this in the code sample is the use of register %1 (the loop index), which can't be updated (by S5) before being used to construct a memory address (by S4). Thus, loop iterations can't be overlapped.

False dependencies are also created by references to main memory, in both the heap[1] and the stack. False dependencies in both can be caused by the reuse of data structures to minimize memory use. False dependencies also occur in the stack because procedure calls at the same level in a program reuse the same portion of the stack.

False dependencies can be eliminated by ensuring that each register or memory location is written only once. In a real processor, this can be done by creating a sufficiently large register file, coupled with other techniques such as *register renaming.* This will enforce a *single-assignment* rule.[2]

[1] In this study, the heap includes global variables.

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

False dependencies in main memory can, in principle, be eliminated by adhering to the single-assignment rule, either explicitly in the programming or by using *memory renaming.* Conventional architectures don't presently perform memory renaming, although dynamic dataflow machines with "colored tokens" [13] employ a form of renaming. False dependencies from reusing the stack can be eliminated in several ways, such as organizing the memory frames for procedure invocation in a tree-like structure [4, 12].

The analyzer performs unlimited renaming by ignoring all false dependencies between objects. 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 disambiguation 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

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 such programs, limiting 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 branches.

The *oracle*, introduced in [11], makes the most optimistic assumptions by ignoring *all* barriers created by branches and jumps. If an oracle also has unlimited memory and register renaming, then parallelism is only limited by the program's true data dependencies.

It is possible to use more conservative models between the extremes of having an oracle and forcing barriers at every branch. One method is *branch prediction,* in which the processor always jumps to what it thinks is the most likely destination. If a branch is taken, and it later turns out that that branch should *not* have been taken, or vice versa, then all instructions executed after the incorrect transfer 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 the error is discovered, which means that at any time, the machine

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

state may represent many different possible program states, only one of which is valid.

The analyzer models branch/jump prediction by keeping only the barriers created by incorrectly-predicted branches/jumps. We use the same prediction algorithm used in Wall's experiments [18], in which the prediction of which way to branch comes from a 2-bit count which is based on the history of previous outcomes of that branch (see "Strategy 7" in [15]). Jump prediction is done simply by predicting that the processor will jump to the previous destination of that particular jump instruction.

A more conservative approach is to assume a form of coarse-grain parallelism in which the barriers produced by branches in one procedure don't affect the scheduling of operations within other procedures. We call this *procedure separation*. It allows procedures with no data dependencies to execute in parallel.

2.1.4 Other resource limitations

So far, we have placed no limitations on how far ahead the processor may look when seeking operations that are ready to be executed. In principle, the *last* operation in the sequential trace could be packed into the *first* PI. If the window size is limited, then 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 models how a real superscalar could schedule instructions at run-time, though in practice, current superscalars have extremely small windows.

Another limitation is to assume that some operations take more than one clock cycle to execute. Our analyzer allows different classes of operations to be assigned different latency values. It places no limitations on the concurrent issuing of long-latency operations, but requires any instruction dependent on the outcome of a long-latency operation to wait the required number of cycles for the data to become available.

2.2 Experimental models used

We used 21 models in our study (see Table 1). 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.

In general, we chose a top-down approach. From the previous discussion, it is clear that the most ideal model, which we call the *Omniscient Oracle*, should have infinite processors, perfect memory disambiguation, unlimited register and memory renaming, no barriers due to control dependences, and no window limitations. We then created models in which one or more features are restricted, to measure their importance.

Machine Model	Renaming		Br./Jump	Win-	F
	R.	Mem.	Predict	dow	P
Omniscient Oracle	Y	all	perfect	∞	N
Myopic Oracle (2K)	Y	all	perfect	2048	N
Myopic Oracle (64)	Y	all	perfect	64	N
Tree Oracle	Y	stack	perfect	∞	N
Linear Oracle	Y	heap	perfect	∞	N
Frugal Oracle	Y	disam.	perfect	∞	N
Or. w. FP Timing	Y	all	perfect	∞	Y
Speculative Multithr.	Y	all	table/sep.	∞	N
Myopic Spec. (2K)	Y	all	table/sep.	2048	N
Myopic Spec. (64)	Y	all	table/sep.	64	N
Spec. w. FP Timing	Y	all	table/sep.	∞	Y
Stupid/Reg Spec.	Y	none	table/sep.	∞	N
Stu./Reg Spec. (2K)	Y	none	table/sep.	2048	N
Stu./Reg Spec. (64)	Y	none	table/sep.	64	N
Stupid Speculator	N	none	table/sep.	∞	N
Stupid Spec. (2K)	N	none	table/sep.	2048	N
Stupid Spec. (64)	N	none	table/sep.	64	N
Multithreaded	Y	all	proc. sep.	∞	N
Smart Superscalar	Y	all	none	∞	N
Stupid Superscalar	N	none	none	∞	N
FP Uniprocessor	N	none	none	1	Y

Table 1: Basic machine models

Thus, six additional oracles, which retain most of the features above, but are limited in some way, were added. Two *Myopic Oracles* measure the effect of allowing only local parallelism to be used, by imposing a size limit on the scheduling window (see Section 2.1.4). Window sizes of 2048 and 64 were used.[4]

The next three 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, only disambiguation.

The final oracle measures the effect of longer-latency operations on parallelism. It is equivalent to the Omniscient Oracle, except that FP operations

[4] Wall's study [18] also used a 2K window.

FP Operand type	Single	Double	Quad.
Add/subtract	4	4	4
Multiply	7	8	9
Divide, square root	23	36	50
Convert, compare	1	1	1

Table 2: Floating-point timings

have the latencies listed in Table 2. In order to make a fairer comparison with machines with 1-cycle FP operations, we also ran a model representing a uniprocessor with the same FP latencies, which supports out-of-order completion of operands but stalls when it detects a data (RAW) hazard.

The other machines don't 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 *Smart Superscalar* has neither, but still has unlimited memory/register renaming and disambiguation. The *Stupid Superscalar* has none of these features.[5]

The remaining 10 machines have procedure separation, and also has branch and jump prediction capabilities, using a separate table for each, as described in Section 2.1.3. The *Speculative Multithreaded* machine has full memory and register renaming capabilities, the *Stupid/Reg Speculator* has only register renaming, and the *Stupid Speculator* has neither. Each was run with window lengths of 64, 2048, and infinity. Finally, the *Speculative Multithreaded* machine was re-run using the FP latencies in Table 2.

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 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

[5] This model is similar to Wall's "Stupid" model [18].

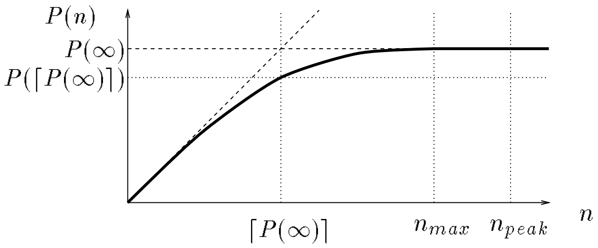

Figure 2: Standard smoothability profile

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 processes operations in the order in which they appear in the 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 speedup can't be more than linear. This is represented by the diagonal dashed line in Figure 2. Once n reaches $P(\infty)$, that value becomes the new bound, as shown by the horizontal dashed line, since $P(n)$ can never be greater than $P(\infty)$. $P(n)$ 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. 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 don't 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). 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)

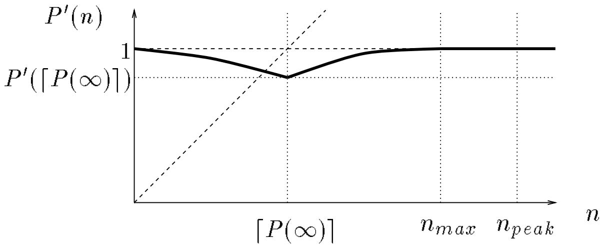

Figure 3: Normalized smoothability profile

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 add to the total number of PI's. However, if the operation is in 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 from above, it can be normalized with respect to that upper bound:

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

Figure 3 shows the normalized form of Figure 2.

If the program is evenly distributed already, or if operations can always be redistributed without increasing total PI count, then $P'(n) = 1$ for all n. Programs that are not ideal will have $P'(n) < 1$ for some intermediate values of n. It can be seen that for a parallelism function with a curve similar to Figure 2, $P'(n)$ will reach a minimum around $n = P(\infty)$. We define *smoothability* to be the normalized parallelism at this critical value, i.e.,

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

To measure the smoothability for a given model, we first compute $P(\infty)$ for that model, round it to the next integer, and run the program again with the resulting value as the PI width limit. We compute smoothability for both the ideal-timing and FP-timing versions of the Omniscient Oracle and the Speculative Multithreaded Machine. We also record the average number of cycles each operation must be deferred. This tells us 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 critical paths.

4 Experimental results

To measure the parallelism available under the various models, a testbed was developed to perform the trace analysis [16]. We chose to analyze traces of programs executed on Sun SparcStations, because the Sparc is representative of present-day RISC processors. Also, the Sparc processor features 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 during procedure calls 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 wanted to confirm the generality of the results by using a different processor.

We ran our analysis tool on 9 benchmarks comprising a total of 20 test cases, under each of the 21 models. We chose problems that are representative of the types of computations 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 overwhelming our analyzer.

The benchmarks (see Table 3) consist of three regular FP-intensive scientific programs, one irregular FP-intensive program, and five symbolic applications. All are from the SPEC benchmark suite, except for `tex` and `spice`, which come from the DLX test suite, and a speech-recognition program taken from a real industrial application. We used the standard input cases, 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. For benchmarks with more than one input case, we ran all available cases to see how parallelism might vary for different sizes and types of inputs.

One of the goals of our study was to analyze complete traces, 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. For instance, the parallelism of `eqntott` under the Multithreaded model was over 6 after 100 million operations, but had fallen below 2 by the half-billion mark.

Table 4 summarizes the complete experimental results. The results of the myopic Stupid Speculators differed from the infinite-window case by no more than 0.01%, and are not listed. Each of the next four subsections discusses the observed effects of one type of architectural limitation on parallelism. This is followed by the smoothability results.

Program	Description	Language
`tex`	Text formatting	C
`li`	LISP interpreter	C
`speech`	Speech recognition	C
`espresso`	Boolean minimization	C
`eqntott`	Truth-table generation	C
`spice`	Analog circuit sim.	FORTRAN
`tomcatv`	Mesh generation	FORTRAN
`doduc`	Hydrocode simulation	FORTRAN
`fpppp`	Quantum chemistry	FORTRAN

Table 3: Benchmarks used

4.1 Branch and jump prediction

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`, `eqntott`, `tomcatv`, and `fpppp`. All programs but `spice` and `tex` exhibit parallelism measurements greater than 300.

A comparison of the Omniscient Oracle with non-oracle machines 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 independent instructions separated by a branch to execute concurrently if they are in different procedure invocations. The Smart Superscalar can't even do this.

From Table 4 we can observe the following:

- 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.

We also tried adding speculative ability to some less-endowed machines, namely, the Stupid Superscalar with and without register renaming. These base machines are closer in ability to present-day superscalars. The results show that both speculation and register renaming are good ways to increase parallelism by modest amounts.

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 Oracles. These oracles are identical, except the Myopic Oracle schedules operations from a window whose maximum length is 2048 or 64. The comparison is quite startling, and leads to the following observations:

- For all programs but `spice`, the Myopic Oracles perform significantly worse. In many cases, part of the loss is due to the fact that the size of the window is an upper bound on the maximum parallelism. However, in no case did the parallelism on either Myopic Oracle even come *close* to the window size.
- For symbolic programs like `eqntott` and `speech`, and numeric programs like `fpppp`, the Myopic Oracles give 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 Oracles' 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 noticeable with `tomcatv` and `fpppp`.
- For all test cases, there was a big drop in performance from the 2K-window Oracle to the 64-window Oracle. The steady loss in parallelism as window size decreases shows that there is an even distribution of the distances between potentially concurrent operations in the trace.

Myopia was less of a detriment to the Speculative Multithreaded machine, presumably because many mispredicted branches are too close together for the analyzer to pack widely-separared operations in the same PI. In most cases, the relative performance loss in going from 2048 to 64 was less for this than for the Oracle, suggesting that the mispredicted branches heavily bias the distance distribution toward short distances. Myopia was almost irrelevant to the lesser speculative models, except for Stupid/Reg in the case of `eqntott`, suggesting that other limitations have a more severe impact on parallelism.

Benchmark	tex		spch.	li	espresso				eqnt.	spice		tomcatv				doduc			fpppp	
Input	man	draft		7 qu.	bca	cps	ti	tial		sm.	lar.	33	65	129	257	tiny	sm.	ref	4	6
Ops (mil.)	15	109	551	202	469	624	729	1190	1704	26	1032	47	188	751	3018	103	523	3018	277	1258
Call Depth	19	23	12	75103	33	41	43	30	18	17	17	17	17	17	17	14	14	14	14	14
FP Ops (%)	.01	.04	4.5	0	<.01	<.01	<.01	<.01	0	10	12	16	17	17	17	14	14	14	19	19
Loads (%)	16	15	15	22	23	21	20	22	33	30	32	46	47	47	48	36	36	36	43	43
Stores (%)	18	8.0	2.8	8.6	2.5	3.9	4.6	4.5	0.7	9.9	10	13	13	13	13	10	10	10	10	10
Oracles:																				
Omniscient	99.1	192	8105	1544	1224	541	403	371	43298	76.5	62.8	1058	2108	4197	8418	562	617	621	4977	11296
Smooth. (%)	93.6	98.3	65.3	94.9	92.6	79.7	93.0	92.3	62.6	92.2	97.4	87.1	85.5	84.7	84.4	99.2	99.4	99.4	98.2	98.7
Delay/op	12.4	23.3	9.70	0.24	4.66	7.88	6.74	30.6	73.5	2.40	19.4	6.38	13.8	29.0	59.1	1.84	1.82	1.79	6.00	3.93
Myopic (2K)	34.0	62.1	90.5	64.0	35.1	59.6	66.7	55.3	141	40.1	28.8	159	125	101	86.4	103	104	104	75.6	75.8
% of Omni.	34.3	32.3	1.12	4.14	2.86	11.0	16.5	14.9	.325	52.4	45.8	15.0	5.93	2.40	1.03	18.3	16.9	16.7	1.52	0.67
Myopic (64)	11.9	12.2	12.4	17.9	12.9	11.9	12.0	12.7	19.6	9.75	9.03	21.9	21.5	22.5	22.6	11.5	11.4	11.4	15.8	16.0
% of Omni.	12.0	6.37	.154	1.16	1.05	2.19	2.99	3.43	.045	12.7	14.4	2.07	1.02	.537	.268	2.05	1.85	1.84	.317	.141
Tree	32.1	75.8	136	58.5	126	86.9	84.5	138	1742	58.8	36.0	423	443	451	457	554	615	621	OO	OO
% of Omni.	32.4	39.4	1.68	3.79	10.3	16.1	21.0	37.1	4.02	76.9	57.3	40.0	21.0	10.7	5.43	98.6	99.7	99.9	100	100
Linear	79.2	167	57.1	336	906	523	OO	370	1314	27.6	20.1	161	157	155	155	25.8	25.0	25.0	71.0	72.8
% of Omni.	79.9	86.9	0.70	21.8	74.0	96.6	100	99.8	3.03	36.1	32.0	15.2	7.45	3.69	1.84	4.59	4.06	4.02	1.43	0.65
Frugal	32.0	71.5	53.6	58.3	124	86.9	TO	138	1314	25.6	19.7	LO	LO	LO	LO	25.8	25.0	25.0	70.9	72.8
% of Omni.	32.3	37.2	0.66	3.78	10.1	16.0	21.0	37.1	3.03	33.5	31.4	15.2	7.45	3.69	1.84	4.58	4.06	4.02	1.43	0.64
FP Timing	OO	OO	OO		OO	OO	OO	OO		15.5	12.9	181	359	712	1425	146	146	146	722	2540
% of Omni.	100	100	100	NR	100	100	100	100	NR	20.2	20.5	17.1	17.0	17.0	16.9	25.9	23.7	23.4	14.5	22.5
Smooth. (%)	93.6	98.3	65.3		OO	OO	OO	92.3		90.0	96.9	80.5	79.2	78.5	78.3	99.9	99.8	99.9	85.6	99.6
Delay/op	12.4	23.3	9.72		4.67	7.87	6.76	30.6		10.4	98.6	42.0	93.7	199	403	4.79	4.16	5.91	40.7	73.3
Speculators:																				
Spec. Multi.	22.2	12.2	129	41.3	9.65	10.8	11.8	10.3	300	27.2	22.4	51.4	52.6	53.6	54.2	158	174	202	297	311
% of Omni.	22.4	6.35	1.59	2.68	.788	1.99	2.93	2.79	.692	35.5	35.7	4.86	2.49	1.28	.643	28.2	28.3	32.5	5.97	2.76
Smooth. (%)	75.7	75.4	61.0	72.2	71.2	75.5	75.5	79.3	64.3	75.8	78.0	71.8	71.9	73.5	74.0	76.8	77.2	80.8	59.1	59.0
Delay/op	265	75.2	440	5.24	63.4	78.4	57.2	258	26.4	16.6	78.9	1.54	2.38	4.07	7.53	501	544	518	0.97	0.91
Myopic (2K)	19.0	10.9	43.9	28.9	7.59	9.16	9.99	8.52	23.8	13.7	11.1	47.0	51.3	53.2	54.1	57.4	60.8	63.9	65.0	65.1
% of S. M.	85.9	89.4	34.1	69.9	78.7	84.9	84.5	82.4	7.93	50.3	49.4	91.4	97.6	99.4	99.9	36.3	34.9	31.7	21.9	20.9
Myopic (64)	9.63	7.43	8.56	10.7	6.33	6.41	6.68	6.09	6.25	7.78	7.54	19.9	20.7	22.2	22.4	10.8	10.8	10.9	15.5	15.7
% of S. M.	43.4	61.0	6.66	25.9	65.6	59.4	56.5	58.9	2.09	28.6	33.6	38.8	39.4	41.4	41.4	6.82	6.21	5.40	5.20	5.04
Stu./Reg Spec.	4.29	3.88	5.19	4.31	4.64	4.18	4.05	3.93	52.2	3.00	2.88	3.09	3.08	3.05	3.05	3.16	3.16	3.16	3.29	3.30
S/R Spec. (2K)	‖	‖	5.19	‖	4.64	4.18	4.05	3.93	18.0	3.00	2.88	‖	‖	‖	‖	‖	‖	‖	‖	‖
S/R Spec. (64)	4.26	3.72	5.06	4.12	4.51	4.10	4.01	3.85	5.87	2.99	2.87	‖	3.07	3.05	3.05	3.16	3.15	3.15	3.29	3.30
Stupid Spec.	2.93	2.71	2.69	2.95	3.02	2.94	2.99	2.76	3.27	2.58	2.53	2.95	2.97	2.95	2.95	3.01	3.01	3.01	2.97	2.98
FP Timing	22.1	12.2	129		9.65	10.8	11.8	10.3		6.40	5.11	SM	SM	SM	SM	57.7	60.6	72.3	102	112
% of S. M.	99.9	99.8	99.9	NR	99.9	99.9	99.9	99.9	NR	23.6	22.8	100	100	100	100	36.4	34.7	35.8	34.5	35.9
Smooth. (%)	75.8	75.5	60.7		71.2	75.5	75.6	79.3		74.3	79.7	71.9	72.7	73.5	74.0	78.6	78.4	84.3	55.9	56.4
Delay/op	265	75.2	371		63.4	78.4	57.1	258		94.4	245	1.54	2.36	4.04	7.42	584	549	518	2.51	2.30
Others:																				
Multithrd.	5.28	3.04	6.30	3.88	1.78	2.18	2.31	2.12	1.66	8.18	6.11	19.4	20.1	19.2	19.2	31.0	32.8	33.2	52.7	54.1
% of Omni.	5.33	1.58	.078	.251	.145	.402	.574	.573	.004	10.7	9.72	1.83	.954	.457	.228	5.52	5.31	5.34	1.06	.479
% of S. M.	23.8	24.9	4.90	9.39	18.4	20.2	19.6	20.5	.553	30.1	27.3	37.6	38.2	35.8	35.5	19.6	18.8	16.5	17.7	17.4
Smart Sup.	2.04	2.08	1.67	1.80	1.47	1.58	1.64	1.59	1.46	3.53	3.90	14.1	18.3	18.7	19.1	4.88	4.92	4.92	30.1	30.9
Stupid Sup.	1.74	1.66	1.43	1.62	1.37	1.43	1.47	1.45	1.43	2.00	2.04	2.69	2.76	2.74	2.74	2.33	2.33	2.33	2.88	2.88
FP Uniproc.	1.00	.999	.866	NR	1.00	1.00	1.00	1.00	NR	.653	.623	.720	.712	.709	.708	.649	.644	.692	.669	.666

NR = Not run because benchmark has no FP operations

Two numbers that look the same actually differ in less-significant digits unless one of the following symbols is used:

LO/OO/TO = Exact same results as Linear/Omniscient/Tree Oracle (same number of PI's)

SM = Exact same results as Speculative Multithreaded (same number of PI's)

‖ = Exact same results as model in row above (same number of PI's)

Table 4: A summary of results

4.3 Frugal oracles

Having established that myopia is usually bad, we next experimented with frugal oracles. In Table 4, 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, renaming of both stack-allocated and heap-allocated data structures seems to be an important characteristic of the Omniscient Oracle. Usually, one or the other is the limiting case for a given program. In all cases, the Frugal Oracle performs as well or almost as well as the worse of the Tree Oracle and Linear Oracle.

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

4.4 Long-latency operations

Finally, we measured the effects of long-latency operations by running the Omniscient Oracle and Speculative Multithreaded machine with FP latencies more consistent with present-day processors. These are listed in Table 4 under "FP Timing." The FP Uniprocessor model at the bottom of Table 4 shows some drop in performance on a uniprocessor (less than one operation per cycle, indication stalls), but in most cases, the drop in performance on the parallel models was much greater, at least for the numerical code. This shows that there are many FP operations on the critical paths. Thus, it may be necessary to devote more chip area to high-speed, low-latency FP units in order to reduce the lengths of these critical paths.

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 interesting to study the effect of limiting the number of processors. If we really had a limited number of processors, is the parallelism profile smooth enough to be effectively mapped to those processors?

In Table 4, we list smoothability measurements (defined in Section 3) for the Omniscient Oracle and the Speculative Multithreaded Machine, with both unit and non-unit FP timings, underneath the infinite-processor measurements. In all cases under all four models, the smoothability is better than 50%. We also list the average number of PI's by which each operation is delayed before it can be executed (delay/op). 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, show that these delays don't significantly increase the lengths of critical paths.

5 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. One important conclusion is that stack renaming can be 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. Another is that the computer must be able to extract parallelism from widely-separated sections of code.

Our study also has important implications for future directions in languages. It is sometimes believed that to get maximum parallelism, we only need to rewrite existing programs in a "better" language, such as a functional language, so that an effective compiler can extract as much parallelism as can be exploited by powerful novel architectures, without being burdened by such sequential artifacts as memory reuse. But some very disappointing results were observed for the Multithreaded model, even though it eliminates false data dependencies, and represents what could be achieved by an efficient dataflow machine running well-compiled programs. Yet some applications, like `tex`, intuitively seem to have much more potential parallelism than found by our Multithreaded model (3.04). This would suggest that in many cases, large-scale parallelism will not be achieved merely by transliterating existing imperative-language programs. Perhaps they should be redesigned completely, starting at the algorithmic level. Short of such changes, one should consider some sort of speculation for programs with irregular control flow. We showed that this can significantly improve parallelism.

Smoothability has been shown to be an interesting measure of the program parallelism. We observed that different programs may exhibit different smoothabilities, but all programs tested demonstrated smoothability better than 50%. 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. We are presently adding control-dependence analysis to extract more parallelism from the non-oracle models, and a memory latency model to investigate the impact on memory hierarchy (e.g. caches, distributed memories). We also plan to extend the tool to compare the effects of multithreading scheduling strategies, and plan to connect the analyzer with our compiler so that we can add measurements relative to the real alias analyses that can be done by a parallelizing compiler. Finally, we are expanding our test suite.

Acknowledgments

This research was supported by Micronet and the Natural Science and Engineering Research Council (NSERC), both funded by the Government of Canada. We also thank the referees who suggested some of the experiments in this paper, and the members of ACAPS at McGill University for their support.

References

[1] A. Agarwal, B. H. Lim, D. Kranz, and J. Kubiatowicz, "APRIL: A processor architecture for multiprocessing," in *Proc. of ISCA-17*, Seattle, Washington, pp. 104–114, May 1990.

[2] T. M. Austin and G. S. Sohi, "Dynamic dependence analysis of ordinary programs," in *Proc. of ISCA-19*, Gold Coast, Australia, pp. 342–351, May 1992.

[3] M. Butler, T.-Y. Yeh, Y. Patt, M. Alsup, H. Scales, and M. Shebanow, "Single instruction stream parallelism is greater than two," in *Proc. of ISCA-18*, Toronto, Ont., pp. 276–286, May 1991.

[4] D. E. Culler, A. Sah, K. E. Schauser, T. von Eiken, and J. Wawrzynek, "Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine," in *Proc. of ASPLOS-IV*, Santa Clara, Calif., pp. 164–175, Apr. 1991.

[5] 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 *Proc. of ISCA-14*, Pittsburgh, Pennsylvania, pp. 189–195, Jun. 1987.

[6] N. P. Jouppi and D. W. Wall, "Available instruction-level parallelism for superscalar and superpipelined machines," in *Proc. of ASPLOS-III*, Boston, Mass., pp. 272–282, Apr. 1989.

[7] D. J. Kuck, R. Kuhn, D. Padua, B. Leasure, and M. Wolfe, "Dependence graphs and compiler optimizations," in *Conf. Rec. of POPL-8*, pp. 207–218, Jan. 1981.

[8] 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-TC*, vol. C-21, pp. 1293–1310, Dec. 1972.

[9] M. Kumar, "Measuring parallelism in computation-intensive scientific/engineering applications," *IEEE-TC*, vol. C-37, pp. 1088–1098, Sep. 1988.

[10] M. S. Lam and R. P. Wilson, "Limits of control flow on parallelism," in *Proc. of ISCA-19*, Gold Coast, Australia, pp. 46–57, May 1992.

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

[12] R. Nikhil and Arvind, "Can dataflow subsume von Neumann computing?" in *Proc. of ISCA-16*, Jerusalem, Israel, pp. 262–272, May–Jun. 1989.

[13] G. M. Papadopoulos and D. E. Culler, "Monsoon: An explicit token-store architecture," in *Proc. of ISCA-17*, Seattle, Wash., pp. 82–91, May 1990.

[14] E. M. Riseman and C. C. Foster, "The inhibition of potential parallelism by conditional jumps," *IEEE-TC*, vol. C-21, pp. 1405–1411, Dec. 1972.

[15] J. E. Smith, "A study of branch prediction strategies," in *Proc. of ISCA-8*, Minneapolis, Minnesota, pp. 135–148, May 1981.

[16] K. B. Theobald, G. R. Gao, and L. J. Hendren, "On the limits of program parallelism and its smoothability," ACAPS Tech. Memo 40, Sch. of Comp. Sci., McGill U., Montréal, Qué., Jun. 1992.

[17] G. S. Tjaden and M. J. Flynn, "Detection and parallel execution of independent instructions," *IEEE-TC*, vol. C-19, pp. 889–895, Oct. 1970.

[18] D. W. Wall, "Limits of instruction-level parallelism," in *Proc. of ASPLOS-IV*, Santa Clara, California, pp. 176–188, Apr. 1991.