p-ray: a Parallelization Analysis Tool for Multicore Software

Saturnino Garcia, Christopher Louie, Donghwan Jeon,
Sravanthi Kota Venkata, Anshuman Gupta
Michael Bedford Taylor
Computer Science and Engineering
University of California, San Diego

August 10, 2009

Abstract

In this paper, we propose a novel set of techniques that allows multicore programmers and architects alike to rapidly estimate the availability of parallelism in their target programs. We present p-ray, which, given an application and the input, shows the nested relationship and availability of parallelism across different regions in complex programs, using a parallelism chart, or p-chart. P-ray also prioritizes the regions, suggesting the ordering that the programmer should attack them in, based on the estimated overall program speedup and the number of lines of code. P-ray can also estimate the kind of parallelism (TLP, DLP or ILP) that is found in each region, which allows the user to determine the available techniques are likely to work.

In this paper, we show three case studies, including SpecInt's gzip, and Mat2C's capacitor, and Nasa7's Vpenta, and follow up with results which compare actual parallelization results of a benchmark suite against the results predicted by p-ray.

1 Introduction

The recent emergence of multicore and manycore processors as the new vector for improvements in microprocessor performance has created a mandate to move traditionally serial programs to multicore processing substrates. No longer can we just wait for higher frequencies. These substrates range from 4 to 64 cores on general purpose platforms such as Intel Core2Duo, Sun Niagara [7], and Tilera TILE64 [3]; and to even greater numbers of cores on more restrictive computing models such as Nvidia GPUs [12].

Unlike in the past, where high processor counts were the domain of super-computers and scientific and parallel computing experts; multicore and manycore processors are now mainstream, and the need exists to reduce the bar for parallelization to these new platform. At the same time, traditional single-threaded applications present new challenges in terms of complexity for discovering and exploiting parallelism. Both of these challenges argue for the creation of new tools to push both the frontiers of facilitating less-experienced programmers, and for extending our techniques to consider applications that are larger and more complicated.

In this paper, we introduce *p-ray*, a tool which allows programs to rapidly diagnose the structure of parallelism within complex applications. In general, programmers would like to ascertain three things:

1. Which regions of the single-threaded code are the best candidates for parallelization?
2. What are the potential benefits of parallelizing those regions?
3. Which parallel techniques should be applied to parallelize a given region?

p-ray facilitates all three goals by analyzing the intrinsic parallelism of an application at multiple levels of granularity, and by displaying the structure of this parallelism in a graphical representation. Given the p-ray output, called a parallelism chart or *p-chart*, the programmer can easily determine which regions of the program are likely to be most fruitful to parallelize, and what the expected benefits are. In order to further assist them, p-ray generates a parallelization effort chart or *PEC* to help the user prioritize which regions to parallelize first. Finally, p-ray generates a *PT Scan* of the program which estimates the kind of parallelism in the candidate regions.

The rest of the paper procedes as follows. Section 2 shows a high-level overview of p-ray. Section 3 shows three case studies, analysing SpecInt's gzip, nasa7, and the Mat2C benchmark capacitor. Section 5 and Section 6 describes p-ray's architecture in more depth. Section 7 presents related work and then we conclude.

2 How P-ray is Used

The goal of p-ray is to provide architects and parallel programmers with a tool that enables them to effectively assess the parallelism of a program. For the architect, p-ray allows them to better understand the structure of parallelism in the programs that they are trying to tune their architecture or compiler for. For the programmer, p-ray provides a suite of tools that help to identify and exploit the parallelism available in a program.

After running the tool through the p-ray profiler, the user's first interaction with p-ray will be through the inspection of a p-chart for the program that needs to be parallelized. Figure 1 shows the p-chart for the *capacitor* program from the Mat2C [6] benchmark suite. p-charts are intended to give the user an intuitive feel for the structure of parallelism in the program. The x-axis of a p-chart tracks the total execution time of the program. The p-chart breaks the execution time into regions that correspond to the function and loops in the program. In the example p-chart, the main function (region F) encompasses the entire program while the seidel() function (region J) occurs twice during execution (starting around 20%) and takes up approximately 35% of the execution time for each instance. Each instance of a region has a parallelism score, which is charted on the y-axis. This parallelism– which we will describe in a later section–is roughly approximate to the speedup possible on a machine with infinite hardware resources. The main region, F, has an estimated parallelism of approximately 32, meaning that in the best case scenario, a 32X speedup could be obtained through parallelization.

The structure of the regions are hierarchical because functions can be called from other functions/loops and loops are nested within functions. From Figure 1, we can see that regions H, K, and L occur during the execution of region J. Thus, these regions are children of region J in the graph representation of the program. As we will later see, it is this nesting relationship that can be used, in part, to determine the type of parallelism in region J: either thread-, data-, or instruction-level parallelism (TLP, DLP or ILP).

With experience, a p-ray technician may be able to inspect a p-chart and quickly identify the best regions to parallelize and also the type of parallelism that may be present. However, for p-chart novices as well as for more difficult programs, such easy identification may not be possible. For that reason, we provide a parallelization effort chart (henceforth, PEC) to guide in the selection of regions to parallelize. The PEC provides a road map for parallelization based on the amount of parallelism and also the lines of code in the regions. Lines of code is factored in to give a crude approximation of difficulty in parallelizing a region. The PEC is a step function, where each each step indicates the region to parallelize and both the relative (compared to previous parallelization efforts) as well as overall speedup

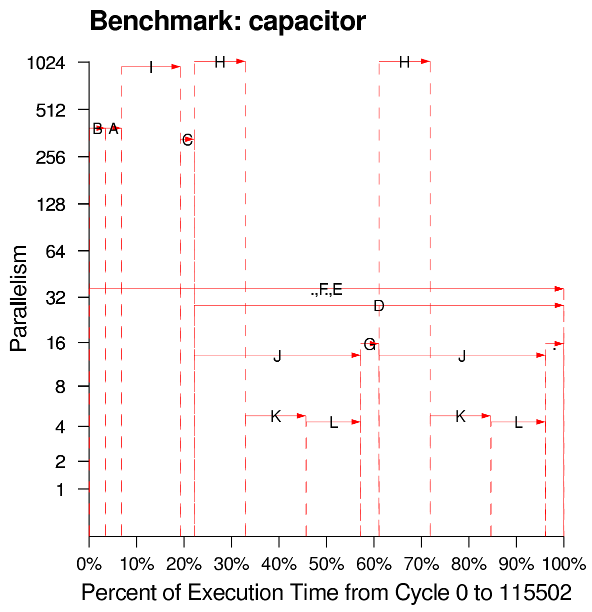

Figure 1: **p-chart for capacitor.** The X-axis shows the passage of time as the program executes serially. The Y axis shows that estimated average parallelism in a given region (loop or procedure) of the program. The letter identifies the region; details for the regions are shown in Table 1 ("PT-Scan for Capacitor"). A "." indicates a region with the same name as the one ahead of it.

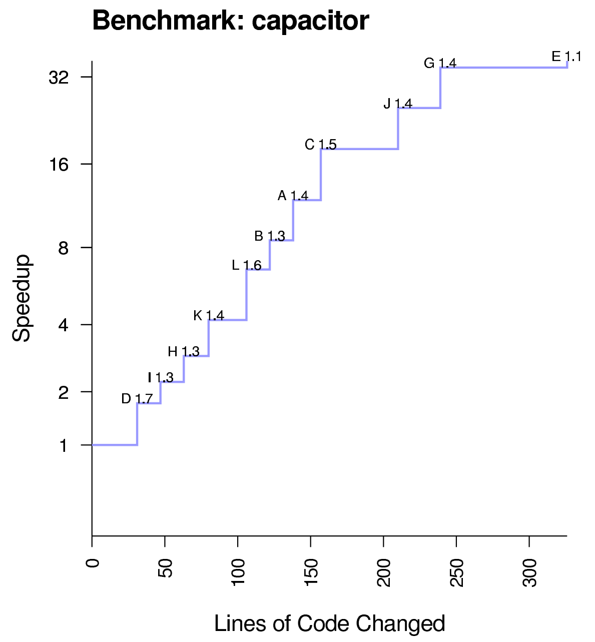

Figure 2: **PEC for capacitor.** A PEC provides recommendations for which regions to parallelize and the expected speedup from parallelizing them.

assuming that the regions in previous steps have all been parallelized.

Figure 2 gives the PEC for the capacitor program. In it, the first step corresponds to region D, whose instances in the p-chart take up nearly 80% of the total execution time (starting around 20% execution time). Based on the PEC, the user is told that a relative speedup of 1.7X is possible if region D–containing only 31 lines of code–is parallelized. The next step on the PEC shows that region I is the region to parallelize next, achieving a maximum relative speedup of 1.3X over the case of only parallelizing region D. A detailed description of the PEC algorithm and how it makes recommendations will be described in a later section.

While the PEC gives the user a region to parallelize, it does not provide any insight into how that region should be parallelized. Based on the three types of parallelism (TLP, DLP, and ILP) the architect will need to provide different architectural/compiler features while the programmer will need to perform different parallelization tasks. For this reason, p-ray provides the parallelism type scanner (PT scanner). After the PEC selects a region to parallelize, a PT scan is performed on the region. The PT scan will indicate whether the parallelism in the region is from TLP, DLP, or ILP. While the programmer can not do much for ILP regions (aside from running on a superscalar processor), she is in direct control of parallelizing both TLP and DLP regions. For TLP, she may divide the child regions of the selected region into separate threads. For DLP, the data parallel operations

Label	Function	Line Range	DLP/TLP/ILP
D	capacitor	178-209	TLP
I	mtimes	ALL	DLP
H	move	ALL	DLP
K	seidel	124-178	ILP
L	seidel	230-256	ILP
A	ones	ALL	DLP
B	zeros	ALL	DLP
C	capacitor	123-153	DLP
J	seidel	ALL	DLP
G	gauss	ALL	no-TLP
E	capacitor	ALL	mixed-TLP
F	main	ALL	DLP

Table 1: **PT Scan for Capacitor**

may be extracted using SIMD processor extension or run on a vector machine. Table 1 gives the type of parallelism that the PT scanner reports for each of the PEC recommendations. It identifies D has have thread-level parallelism. Regions J and G, which produce the sub-regions of D, correspond to the main seidel() and gauss() functions of the benchmark. This correlates well with the PT scan output as regions J and G could be made into their own threads to provide parallelization of region D. Again, we delay detailed description of the PT scan algorithm until a later section.

After a selected region has been parallelized (if not ILP), the PEC can be consulted once again for the next recommended region to parallelize. The enterprising manager may find the N next best regions to parallelize and give them to separate programmers to work on in parallel.

So far we have given an overview of the diagnostic procedure and the tools available to the architect and the programmer to help with parallelizing a given program. In the next section we present two case studies (Vpenta and gzip) in order to clarify the utility of or p-ray diagnostic procedure.

3 Initial Case Studies

In this section we present case studies of two benchmarks: Vpenta and gzip. In these case studies, we will discuss the p-ray diagnostic procedure in detail and explain its results. These two benchmarks provide an interesting juxtaposition with Vpenta proving to be highly parallel while gzip remains notoriously serial in nature.

3.1 Vpenta

In this section, we will walk through the diagnosis for Vpenta, a nasa7 kernel that simultaneously inverts three pentadiagonal matrices. The program can be divided into three phases : initialization phase (region F), construction of pentadiagonal matrices(region O) and computation of inverse.

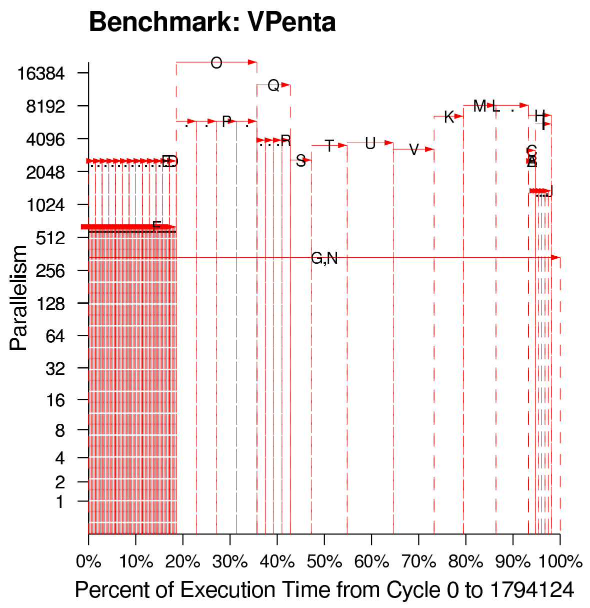

Figure 3: **p-chart for Vpenta**

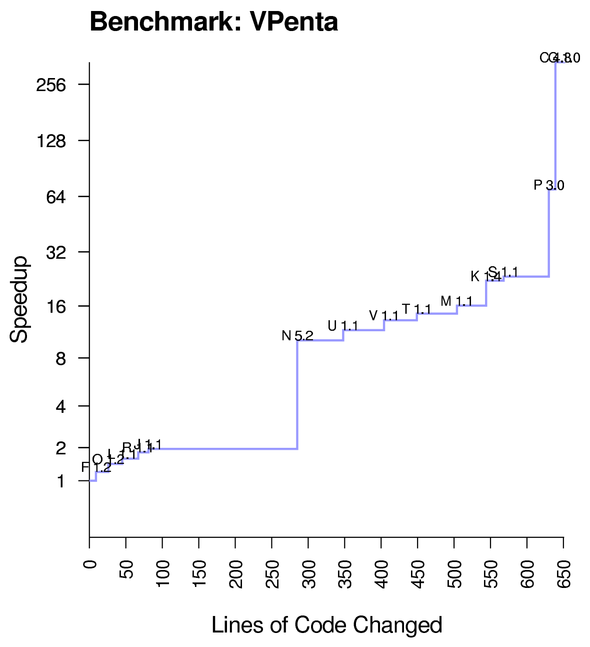

Figure 4: **PEC for Vpenta**

Figure 3 shows the p-chart for Vpenta benchmark. The working set size was set to 4x1024. The total execution time of the program is represented on x-axis and y-axis denotes parallelism. As we sweep across time in the p-chart, we see that region E occupies close to 20% of the execution time. A closer look at the chart would show tiny regions (region F) enclosed by E. The amount of parallelism across each tiny region is same, close to 512, suggesting these regions are DLP. Further, the combined parallelism of E is more than the individual child region. This suggests E, the initialization phase, is a DLP region. This is in tandem with the code, where this phase fills 13 different matrices with zeros.

Phase 2 involves construction of the matrices, as depicted by region O. The prototype of the code body is:

```
for i=1:4
  for j=1:512
    a = (i*j+1)*constant;
    A[j,i] = a-round(a);
    repeated for 8 arrays ....
```

Looking at the p-chart, we see that region O has very high parallelism (close to 16384) and the amount of work spans close to 20%. Similar to region E, O has 4 child regions (region P) enclosed within whose parallelism is close to 6000. The overall parallelism of region O is as much as twice that of its children, making it a DLP case. This observation from p-chart can be verified. The prototype shows that the outer loop is executed 4 times, with each iteration performing heavy parallel computations in the Do-All loop.

The crux of the program lies in the inverse computation phase. This spans regions Q, S, T, U, V, K and M accounting for approximately 60% of the execution time. As we see, Q encloses four instances of region R and has high parallelism, making it a DLP loop. Region S has high parallelism despite little work (5%), which would mean S has a small critical path and high parallelism, making it DLP as well. This trend is observed in all the other regions - T, U, V, K M and L. From the p-chart, these regions appear to have considerable DLP component.

Despite having so many DLP regions, the overall parallelism of the program (370), represented by regions G and N, is lower than that of any of its children. This rules out DLP or perfect TLP scenario for phase 3 of the program. Consuming as much as 65% cycle count, the low parallelism can be attributed to its long critical path that passes through all the regions, marking a long dependency chain. This makes phase 3 partial-TLP, i.e., only limited TLP can be found, involving partial parallelism between sub-regions, based on data dependences.

Combining this analysis from p-chart with PEC recommendation chart, see Figure 4, will guide us in the selection of appropriate regions to parallelize. Intuitively, the best regions to parallelize would be loops with small dependency chain and heavy work. Tranformations such as loop interchange, unrolling could be good candidates to improve parallelism. Regions with do-across computations are limited by the length of their critical path and thus they require more effort for parallelization. From the discussion of p-chart, improving 40% of do-all computations in initialization (region F) and construction (region O) phase could improve the overall parallelism.

Label	Function	Line Range	DLP/TLP/ILP
G	drv_vpenta	ALL	partial-TLP
F	zeros	57-66	DLP
O	vpenta	498-577	DLP
L	vpenta	1059-1133	DLP
R	vpenta	600-622	DLP
J	susbref	65-79	DLP
N	vpenta	ALL	partial-TLP
U	vpenta	862-925	DLP
V	vpenta	938-994	DLP
T	vpenta	676-721	DLP
M	vpenta	1074-1129	DLP
K	vpenta	1006-1046	DLP
S	vpenta	639-663	DLP
P	vpenta	511-573	DLP
A	ones	ALL	DLP
D	zeros	ALL	DLP
H	susbref	ALL	DLP
B	ones	44-70	DLP
C	ones	57-66	DLP
E	zeros	44-70	DLP
I	subsref	65-79	DLP
Q	vepnta	587-626	DLP

Table 2: **PT Scan for Vpenta**

Figure 4 gives the PEC for Vpenta program. The first step corresponds to region F, whose instances in the p-chart take up nearly 20% of the total time. PEC suggests that improving 11 lines of code in F gives nearly 1.2 speed-up. As we move ahead, PEC suggests region O (construction phase), region L and region R as the next candidates, amounting to 1.2x, 1.1x and 1.1x incremental speed-up. From the p-chart we can see that all these regions are Do-all candidates with short critical path. These recommendations by PEC confirm our theory of parallelizing heavy work, do-all regions ahead of do-across. Once we have exhausted the do-all regions, PEC suggests we move ahead with the partial-TLP cases - region U,V,T,M,K and S.

To evaluate the performance and efficiency of PEC and p-chart, we will compare the predicted results with actual ones. Table 2 and 4 summarizes the results of Vpenta benchmark. The benchmark performance was analyzed on our manycore simulator and the speed-up numbers were normalized to single core execution. As per PEC's recommendation, region Q, O and D (do-all) were parallelized using transformations such as unrolling, loop interchange, induction variable replacement, resulting in speed-ups as high as 66x. For the remaining regions that were part of the partial-TLP block, transformations such as reduction operator optimization, privatization, scalarization were applied to improve performance. The overall performance of the program achieved a speed of 58.20x on 32 core machine.

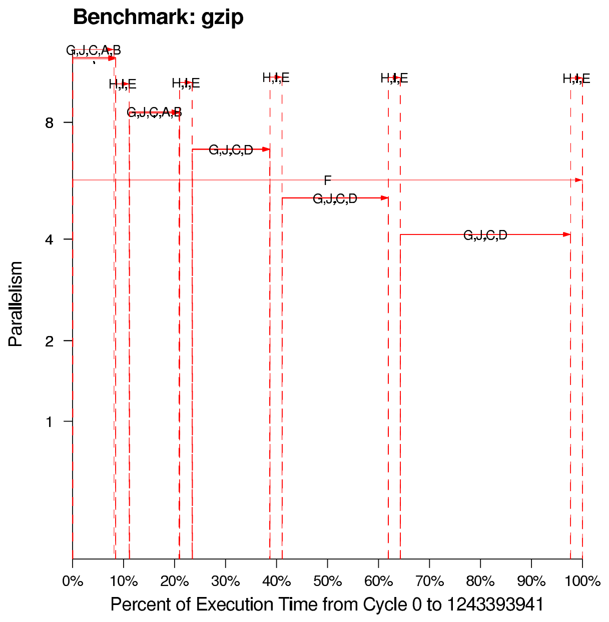

Figure 5: p-chart for gzip

3.2 gzip

For our next case study, we will look at the gzip program from the Spec2000 benchmark suite. gzip is a program that performs compression and decompression based on the LZ77 algorithm. Figure 5 gives the p-chart for gzip while Figure 6 shows the PEC. In comparison with Vpenta, whose overall parallelism is close to 256, gzip's main region, F, has a parallelism of only 6. In the Spec2000 version of gzip, compression/decompression (regions G, J, and C) happens at 5 levels in sequence, with the result being checked (regions H, J, and E) after each level.

Intuition would tell us that to parallelize this benchmark, we would want to perform the various levels in parallel. Referring to the p-chart in Figure 5, we can see that F is recommended first, for a potential speedup of 3X. Consulting the PT scan results in Table 3 indicates that the region contains only ILP. The critical path through region F is nearly equal to the sum of the critical paths of all the sub-regions, which is a direct indication that they are all serially dependent upon one another (i.e. there is no-TLP). While slightly disappointing, this confirms the conventional wisdom about the serial nature of the unmodified gzip algorithm.

According to the PEC, the only other profitable region

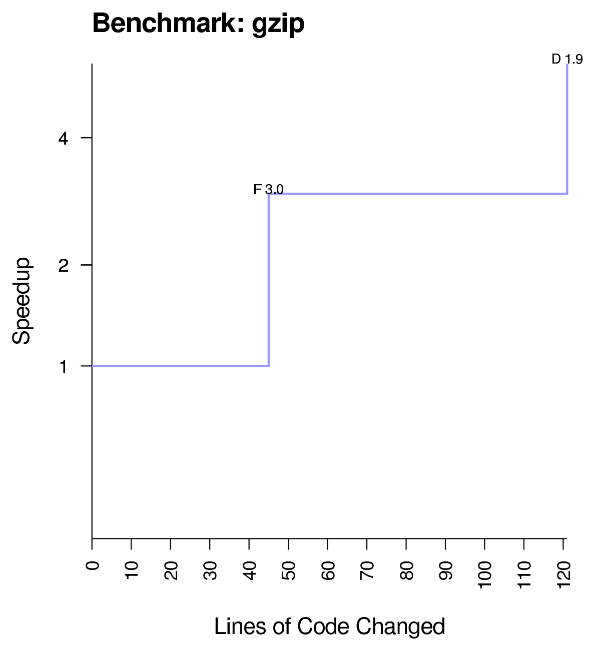

Figure 6: PEC for gzip

Label	Function	Line Range	DLP/TLP/ILP
B	fill_window	545-547	TLP
F	inflate	930-935	TLP
H	spec-main	297-344	TLP
L	util-xmalloc	ALL	DLP
C	deflate_fast	ALL	DLP
G	spec_putc	ALL	ILP
I	spec_initbufs	ALL	ILP
D	deflate_fast	653-653	ILP
E	inflate_block	ALL	DLP

Table 3: **PT Scan for gzip**

for parallelization is D. Once again we consult Table 3 for the type of parallelism available in region D. Sadly, once again the region is found to contain only ILP and is thus not a good candidate for parallelization. While D does have some modicum of DLP through calls to sub-regions (pruned out in the p-chart), they amount of work spent in these sub-regions pales in comparison to the total amount of work in D.

The overall conclusion of parallelization efforts of gzip is therefore grim according to both conventional wisdom and the p-ray tool.

4 Experimental Results

4.1 Evaluation of p-ray

So far we have discussed the p-ray tool and its utilities in great detail with case studies. In this section we will evaluate the effectiveness of p-ray by comparing the predicted results with experimental results obtained through real simulations. For this purpose, we obtained p-charts and PEC graphs for a few interesting benchmarks(see Table 4). At the same time, we used these benchmarks, and the results to tune our parallelizing compiler for generation of code for a tiled multi-core simulator.

Based on the recommendation of the PEC and the contour of the p-chart, we were able to target the regions with parallelism, and implement or augment transformations in the compiler in order to extract the PEC-promised speed up. We applied transformations such as privatization, scalarization, loop interchange, and loop invariant code motion in order to allow the compiler to realize the parallelism indicated by p-ray.

Our compiler identifies regions that exhibit DLP and ILP and exploits parallelism using a combination of unrolling and space-time scheduling over a scalar operand network [15]. We were not able to parallelize all of the regions indicated by the tool because our current infrastructure is not capable of exploiting thread level parallelism. Also, our tool chain currently does not associate any cost for library function calls, in particular sin and cos. Thus finediff and crnich benchmarks show varied cycle count for initialization regions, which involve sin() calls.

Vpenta We will start our analysis with Vpenta. As discussed in the case study, Vpenta has a heavy presence of DLP across its individual regions and the enclosing body exhibits partial TLP due to dependence across each child region. For child regions, that are suggested as primary candidates for parallelization effort focus, our simulator gave speed up as high as 90x on 32 core. The high amount of work (see Table 4) in each child region and predicted parallelism contribute towards the speed up. The overall speed up is a fraction of many of the children regions because of the long dependency chain that runs through them. This phenomena is well captured in the speed up trend across regions. Also notice that the initialization phase of Vpenta, that consists of more than ten different array fills, does not show considerable speed up. Intuitively, we would identify the fills as TLP. But since our compiler does not support TLP, we do not get good speed-up numbers in that region.

Capacitor Capacitor is a benchmark with high DLP and ILP content. Region D, which has calls to seidel and gauss, encloses two instances of region J. The p-chart identifies this as TLP and suggests there be a speed up of 2.0x on region D. Intuitively, we can claissfy this behavior as pipelined parallelism case. But from table it is evident that region D only manages 1x speedup. Scheduling the two threads could extract good parallelism across the entire seidel function, but we are limited by our compiler's restriction on TLP. Similarly,for Region J enclosing K and L that comprise the seidel function of the benchmark, the predicted

Benchmark	Region	Lines	Work		Speedup		Cores Used	Parallelism
			Predicted	Actual	Predicted	Actual		
Vpenta	N		1,794,124	1,945,675	370	58.20x	32	partial-TLP
	Q	39	143,529	139,656	12000	66.41x	32	DLP
	U	63	179,412	193,617	4100	98.33x	32	DLP
	V	52	161,471	152,929	3900	86.35x	32	DLP
	T	45	134,559	132,705	4000	85.12x	32	DLP
	M	153	215,294	195,962	8192	73.28x	32	DLP
	K	40	89,706	78,926	7000	82.30x	32	DLP
	D	40	435,971	479,232	2500	20.00x	32	DLP
	O	79	269,118	294,912	16384	40.30x	32	DLP
	S	24	71,764	67,681	3000	65.08x	32	DLP
Capacitor	F		115,502	97,867	35	1.73x	32	partial-TLP
	I	20	23,100	28,478	1000	10.77x	32	DLP
	C	30	3,465	2,469	370	0.45x	32	DLP
	G	55	6,930	6,492	16	4.52x	32	partial-TLP
	D	55	6,930	6,492	16	4.52x	32	TLP
	J	160	75,076	64,666	12	2.68x	32	partial-TLP
Crnich	P		880,171	1,119,035	10	2.54x	32	partial-TLP
	N	20	44,008	59,125	900	84.95x	32	DLP
	HB	32	158,430	258,159	2.5	1.51x	32	ILP
	JB	20	183,835	189,939	3	2.14x	32	ILP
	E	40	44,008	36864	2048	20.00x	32	DLP
	U	44	17,603	18,432	1500	20.00x	32	DLP
	W	44	17,603	18,432	1500	20.00x	32	DLP
	Y	44	17,603	18,432	1500	20.00x	32	DLP
	L	22	44,008	18,432	1500	20.00x	32	DLP
	H	22	44,008	244,671	1024	23.68x	32	DLP
Finediff	C		435,620	2,350,407	6144	106.16x	32	partial-TLP
	E	26	239,591	301,724	5000	101.01x	32	DLP
	A	40	67,634	92,160	4096	20.00x	32	DLP
	D	67	143,754	1,954,286	8192	116.50x	32	DLP
Nbody1d	C		2,122,719	2,828,3317	2000	43.77x	32	partial-TLP
	E	133	1,804,371	2,396,000	1600	44.26x	32	DLP
	(A-D)	40	43,054	55,296	4096	20.00x	32	DLP
	(D-E)	116	297,180	370,756	2000	44.94x	32	DLP

Table 4: **Comparison of predicted and actual results.**

speed up of region J is almost twice that of region K, making it a TLP scenario. Though our compiler extracts as much as 10x parallelism on initialization regions of capacitor, it is severely limited by the TLP-only nature of seidel and gauss (region G). This reflects in the speed up numbers of J and G, that cap at merely 2.6x and 4.4x despite the fact that they have large working set and enough work since only ILP can be extracted from them. This fact is reflected in the PEC chart, which suggest we start with DLP loops to extract parallelism and then try to venture into the do-across realm.

Crnich Crnich finds Crank-Nicholson solution to one-dimensional heat equation. This benchmark is a classic case of regions with TLP and DLP. The initialization phase which involves array fill, matrix multiply and negate operations on different arrays would be considered a TLP scenario. Each of these regions (U,W,L) are DLP, if looked individually, but they can be scheduled in parallel. But the actual speed up numbers do not match these justify this. If TLP was handled in the compiler, we could have achieved PEC promised speed up on these regions. For the DLP regions, the compiler achieves speed up as high as 85x.

Nbody1d Nbody1d exhibits pipe-lined parallelism in its region E. This region comprises of three loops and the iteration count for each loop remains the same and the data computed in the the first loop is used by the second, so on. Ideally we could visualize this as a TLP case and schedule the threads in pipe-lined fashion. This fast is reflected in the p-chart which computes the parallelism on region D greater than region E. Other than these regions, Nbody1d handles DLP regions effectively.

Finediff Finediff comprises two loops, one the initialization loop and the other being the actual computation. The initialization loop has heavy sin() operation computations, region E. Region E is a work intensive DLP loop that has high parallelism. The compute loop is a do-across loop with heavy work, region E. The heavy work accounting from this loop justifies high parallelism despite the long critical path. The overall parallelism of the program, region C, is a typical partial-TLP case. Exploiting pipe-lined parallelism and TLP is the compiler can give us promising results for the speed up numbers. The PEC recommendations provide useful insights into improving parallelism across the program.

The analysis we have made for each benchmark provides a useful insight into parallelism and the efficiency of p-chart and PEC with respect to analyzing programs and suggesting the effort curve.

5 Algorithms for Creating P-charts

The first interaction that a programmer will have with p-ray is through a p-chart. These p-charts are visual representations of the parallelism of a program during different stages of execution. As all subsequent tools utilize the information presented in the p-chart, it forms the core of p-ray. In this section we will talk about the design and implementation of p-charts.

5.1 Region-based Profiling

p-charts track the parallelism through regions of a program as execution proceeds. While previous work has calculated the critical path and work through a whole program [8, 13, 16, 10, 9], they do not convey the fluctuating nature of parallelism between and across program regions. As we will see, while some parts of the program are highly serial in nature, others can be highly parallel. Looking only at the average parallelism across a whole program, it is difficult to differentiate the highly parallel from the serial regions of the program.

P-ray implements an LLVM-based [11]-based source-to-source instrumentation pass that instruments the programmer's code to record critical path and work information as it executes. This information is processed as the program executes in order to prevent the need for large log-files.

We define a region as any single-entry piece of code. It may have any number of exit points. Under this definition, many different structures of a program may be considered a region. These range all the way from whole functions down to loops, basic blocks, or even individual instructions.

For p-ray we consider only two types of regions: loops and functions. We consider loops because they are a natural source of parallelism, as evidenced by the fact that data-level parallelism is often referred to as loop-level parallelism. We include functions because they form natural divisions of the program and often encompass a single task to be performed. In other words, they are prime candidates for task-based parallelism (i.e. thread-level parallelism). We exclude individual instructions as region candidates because they offer almost no opportunity for parallelism. While considering basic blocks as regions would provide us information about instruction-level parallelism, we do not track them because they offer only limited opportunities for parallelism and the overhead for tracking them would be excessive. To reiterate, when we use the term region in this paper, we use it to mean either a function or a loop.

As we previously mentioned, regions form a natural hierarchy. To visualize this, we imagine the regions to form a graph with nodes being regions and an edge from region A to region B indicating that region B is contained *directly* within region A. If a loop region is contained within another region, it means the code for the loop is found in that region. On the other hand, if a function is contained within another region, it indicates that this function is called from the containing function. The region graph can be thought of as a call graph where loops are also callable (albeit, only from one place).

We should take the time now to reinforce the concept that the region graph is a static construct. It does not take into account the actual execution of the program. As we will see in the following sections, a dynamic region *tree* is formed that can be used for pruning out uninteresting information and also for diagnosing possible TLP regions.

5.2 Region-tree Pruning

During execution of a large program, many regions are encountered. Some of these regions may be trivially small or otherwise uninteresting. Because a p-chart is intended to aid a programmer in visualizing the parallelism across the whole execution of a program, it is important only to include those regions that are interesting and helpful. To accomplish this, we create a region-tree based on the execution of the program and then prune that tree in order to show only those regions which are useful.

The region-tree is based on the dynamic execution pattern of the program. There is one node for each dynamic execution of a region (e.g. function call for a function region) and the children of each node follows the same containing semantics as with the region graph. The leaf nodes of the tree are loops or functions that do not contain any other loops/function calls within them while the root of the tree is the main() function of the program.

For large programs, there are too many regions to visualize and many of them are not worth exploring. In this section, we will discuss how we prune the region tree down. We find that we can trim down the number of regions to display so the information is easily digestible using the following two rules:

1. Eliminate any dynamic region whose parent has parallelism of nearly 1 (i.e. is serial)
2. Eliminate any dynamic region individual work makes up an insignificant portion of the total work of the program

The first rule keeps the top most regions that are practically serial using its current algorithm and eliminates any of its sub-regions to show that these regions would be hard to parallelize in their current state. The sub-regions which are quite serial as well are not displayed because they would be providing redundant information. The second rule eliminates all the regions that would not greatly speedup the program if they were parallelized. By using these rules, only the important regions remain.

5.3 Calculating Parallelism

In order to calculate the parallelism in a region, we need two pieces of information:

1. The total amount of work performed in the region.
2. The minimum possible time to execute the region (i.e. the critical path of the region).

Using this information we are left with the following simple equation for parallelism, p:

$$p = \frac{work}{criticalpathlength} \tag{1}$$

This equation tells us that in order to perform the total amount of work in the region, we would need to execute, on average, p work cycles for every cycle. In other words, we could achieve p-way parallelism in the best case. If the amount of work is similar to the critical path length then there is not much parallelism to be exploited. In this case, p will approach 1. If, on the other hand, we have a loop with n iterations that can all be performed in parallel, then p will be approximately n.

In order to measure the work and critical path, we use a custom instrumenting infrastructure to annotate C source code. We use the LLVM infrastructure [11] to perform this instrumentation. This instrumented source code is compiled and executed and the output data parsed to gather the data required as input to all the p-ray tools. The use of this dynamic, run-time analysis allows us to discover information that would be difficult or impossible at compile time, and allows us to capture actual memory dependencies in the program, as opposed to rely on a conservative pointer analysis. Although this opens the possibility of overly optimistic estimations of parallelism, it provides information that a conservative analysis would not be able to.

As we will see in the following subsections—describing how work and the critical path are calculated—dynamic profiling is key to disambiguating pointer aliasing and allowing us to find the true critical path through a program.

5.3.1 Work

We define the work of a region as the total number of *useful* cycles needed to execute that region of the program. While many architectural factors (e.g. register spilling because of a limited number of registers) will impact the total number of cycles to execute, we are interested only in those that make progress toward completing the program. The work done can be calculated by tracking the number of dynamic instances of each type of instruction (e.g. integer add, floating point divide, etc.). For each of these types of instructions, we define the number of cycles that it takes to execute this instruction. The number of cycles required for a given type of instruction is normalized to the time taken for an integer addition. Simple instructions, such as integer subtraction and comparison, should take a single cycle while complex instructions such as floating point divide may take many cycles.

Note that the cycles per instruction is defined under the assumption that all operands are available and there is no contention for hardware needed to execute the instruction. The total amount of work in a region is roughly equivalent to running this part of the program on a single-issue, in-order processor. The total work therefore provides a baseline for a completely non-parallel implementation of the program. As we will discuss in a later section, all types of parallelism—ranging from instruction-, to data-, to thread-level—will result in a speedup over this baseline.

5.3.2 Critical Path Algorithm

As we alluded to earlier, the critical path through a region is the minimum amount of time (in cycles) needed to execute that region. In order to calculate this path, we need to define the time when each executed instruction is available to be used.

In order for an instruction to be executed, it must have all of its inputs available. The input that is available at the latest time is the *critical input* and will define the time ($t_{avail,crit}$) at which the instruction may begin executing. Note that there may be multiple critical inputs for any instruction. As we discussed in the previous subsection, each type of operation has an associated execution time (t_{inst}). The time at which the instruction is available (t_{avail}) is therefore governed by the following equation:

$$t_{avail} = t_{avail,crit} + t_{inst} \tag{2}$$

For assignments of the form A = B, there is no operation being performed, t_{inst} can be view as being 0. The availability time of the left hand side (A) is equal to the availability of the right hand side (B). Function calls that return a value (e.g. A = func()) are handled similarly to assignments. Before the function is called, the variable where the return will be stored is noted and right before the function returns, an assignment between the return value and the noted value is made. If an instruction uses a constant (e.g. A = B + 1) then only the time of B is considered when calculating t_{avail}.

In order to avoid problems with pointer aliasing, we keep a mapping from address to the time it is available. When we perform an operation (e.g. A = B + C), we update the mapping of the address where the operation (e.g. A) is stored. Similarly, when looking up critical input time, we use the address of the inputs (e.g. B and C). In this way, if another variable, D, is aliased to A, then executing A will update the address associated with both so using D later (e.g. E = D + F) will find the correct availability time for D.

As we are executing and updating time mappings, we check for the maximum availability time that we encounter. It is this time that will define the minimum amount of time required to execute the region and therefore this is the critical path length. By starting at the operation and tracing back through the critical inputs, we can find the critical path through the region.

Control Dependencies Consider the following code example:

```
a = b + 1;
c = 0;

if(x > 10)
  c = a + 1;
else
  c = a - 1;

d = c + 5;
```

If we consider only the data dependencies (as is the case for t_{avail} above), then we may miss the fact that the availablility of c is also determined by the outcome of x $>$ 10. In order to account for this, we define a minimum time, t_{min}, that a t_{avail} can have in a control dependent region. This t_{min} is set to the availability of the branch condition. In the above example, the branch condition is the calculation of x $>$ 10 so t_min for the then and else branches is set to that availability time. We therefore need to update our t_{avail} equation as such:

$$t_{avail} = max(t_{avail,crit} + t_{inst}, t_{min}) \tag{3}$$

With this updated equation, we are now able to handle both control and data dependencies in the code.

Loop Induction Variables Loops are one of the most fertile areas to look for parallelism. Consider, for example, the following loop:

```
for(i = 0; i < N; ++i)
  a[i] = b[i] + c[i]
```

Assume that arrays a, b, and c do not alias to one another then it is easy to see that all iterations of this loop can execute in parallel. We would thus expect the critical path length of this region to be 1 (the time for an integer addition). However, when this loop is executed, the induction variable, i, is updated with every iteration. Because i at each iteration depends on the value of i in the previous iteration, a data dependency would be noted and the critical path length would N. Luckily, LLVM is able to identify loop induction variables and we can force our profiler to ignore this false dependency on i.

Region-based Critical Path In order to isolate the critical path of a region, we first have to ensure that all the inputs to the region are assumed to be available at the same time. In order to achieve this, whenever we enter a new region, we create a new address to time map. If an instruction looks up the address for operand and finds that it is missing from the address to time map, it is assume that the input is available at time 0. In this way, all instructions that depend on instructions outside of the region are guaranteed to be able to start at the same time.

6 P-Ray Diagnostic Tools

At the heart of p-ray is a set of tools, which a programmer can utilize to quickly diagnose the location and type of parallelism in a single-threaded program. Having already seen how p-charts are created, we will now move on to the other tools that p-ray offers. Using these tools, a programmer is able to iteratively select promising regions for parallelism (with a PEC) and identify the type of parallelism in those regions (using the PT scanner).

6.1 Parallelization Effort Charts (PECs)

The PEC for a program is generated based on the potential benefit of parallelizing a region of code vs. the number of lines of code in that region. We have omitted the pseudocode for this algorithm but will now describe it at a high-level.

We start off with an empty list of regions that we are going to parallelize and try adding each region to that set and see what marginal benefit we would get. After trying all the regions, we pick the one that provided the best speed up per line of code and pernamently add it to the end list of regions that we are going to parallelize. Next, we repeat with the remaining regions until we run out of regions to parallelize or we are only left with serial regions. The resulting list is the order in which we would like to parallelize all the regions.

To find the marginal benefit of adding a particular region, we begin by recursing down to all of the leaves of the region-tree and calculate their new execution times first. If they are in the set of static regions that we are going to parallelize, their new execution time is their critical path time (i.e. the theoretical parallelization time). Otherwise, their new execution time is still their serial execution time.

After we calculate all the leaf region execution times, we can continue calculating up the region tree. For non-leaf regions, we gather all the new execution times for all the children. If we are going to parallelize this region, then its new execution time is the max of its children's new execution times and its critical path time. We consider the children's execution time in the max function because the fastest execution time for this region would be the greatest of these times for any child we have not parallelized yet. Any child that we have already parallelized will be less than or equal to this region's critical path, so those ones will not affect the result. In addition to this purpose, we need to include the region's critical path time in the max function so that we capture any dependencies between the children and in the work done just inside this region.

If we are not going to parallelize this region, then the new execution time is the sum of the execution times of the children plus the work done just in this region. Since the work of a region includes all the work of its children, we have to subtract out the children's work from the current region's work to get the work just in this region.

6.2 Parallelism Type (PT) Scanner

Having used the PEC to select one (or more) regions to parallelize, we would like to quickly diagnose the type of parallelism that is available in this region(s). To accomplish this, we have developed the PT scan. Using data available from the p-chart and corresponding region tree, the PT scanner utilizes several heuristics to infer the type of parallelism (data-, thread-, or instruction-level) in the region. In the following subsections, we will discuss the characterestics that the varying types of parallelism will have that allows us to easily identify them. Afterwards, we present the algorithm for classifying a region's parallelism.

6.2.1 Identifying Data Level Parallelism (DLP)

The first type of parallelism which we will look at is that at the data level. The hallmark of DLP is that the same instructions are performed on multiple data streams. In trying to identify DLP, there are several cases to look for. The first, and most obvious, is if the region has a high parallelism score. In the event that a region has no children, we can say that a parallelism higher than P_{DLP} is a DLP region. In this no-child case, having a parallelism lower than P_{DLP} indicates that only ILP is present in that region. In practice, as ILP is usually highly constrained, the value for P_{DLP} can be as low as 8-16.

In the case that a region does have children, a high parallelism score alone is not enough to determine if the region has DLP. As we will see in the next subsection, a TLP region may also have a high parallelism score. If a region has children and a high-DLP score then we can look at the nature of the children to determine if the region has DLP. As we mentioned earlier, DLP involves repeating the same operation on different sets of data. With this is mind, if the children meet the following criteria, we say the region is DLP:

1. All children are dynamic instances of a single static region
2. All children have the same amount of work and parallelism (P_{child})
3. The parallelism of the parent P_{parent} is $n * P_{child}$, where n is the number of children

The first two criteria ensure than each iteration of the loop performs the same work while the last criterion ensures that the children can be executed in parallel (i.e. this is not a DO-ACROSS loop).

6.2.2 Identifying Thread Level Parallelism (TLP)

The next type of parallelism we will look at is TLP. While multiple threads may operate on the same code, we focus on the type of TLP that arises from doing work on differing regions of code. If we are interested in the former, we note that there is significant overlap with between identical threads and DLP. Any region we identify as having DLP may easily be converted to TLP by having a gang of threads execute the iterations of the loop in parallel. This is especially true for loops with longer bodies or for the nested loop case described in the DLP section.

When we encounter that has multiple subregions as potential threads, there are several scenarios that are possible. In the best case, the work performed by the subregions is independent of one another so that all subregions may be executed in parallel. We will refer to this as the perfect-TLP case. On the opposite end of the spectrum is when each subregion depends on the previous subregion so that they must be executed serially. This case will be referred to as the no-TLP case. The final possibility is when the subregions are partially dependent upon those before them; the so-called partial-TLP case. We will now discuss how we differentiate between these three scenarios.

In the perfect-TLP case, the critical path time of the enclosing region will be dominated by the time of the longest critical path length among the subregions. For the no-TLP case, the critical path of the enclosing region will be the sum of critical path lengths of the children. These two values provide the range for possible critical path lengths for region under consideration for TLP. The actual critical path length of a region will lie somewhere along this spectrum. In order to determine the "fitness" for a region to be considered as TLP, we normalize the actual critical path length to where it lies along this spectrum. The following equation governs this fitness metric:

$$f = \frac{CP_{no-TLP} - CP_{actual}}{CP_{no-TLP} - CP_{perfect}} \tag{4}$$

We can set some minimum fitness value, f_{min}, below which there are too many dependencies among subregions for the enclosing region to have TLP. If a region fails to meet this minimum fitness, then it serial and contains only ILP.

6.2.3 PT Scan Algorithm

The algorithm for the PT scanner–omitted for brevity–follows the same process of elimation as we have seen in the previous subsections on identifying the types of parallelism. If a region has no children, it is either DLP or ILP, depending on the level of parallelism in the region. If there are children, if they are from heterogeneous static regions, then the TLP fitness metric is consulted to label the region either as TLP (if above f_{min}) or ILP (if below). If the subregions are homogenous then the region's parallelism is checked against that of the children to verify the status of criteron 3 above. If this criterion fails, then the TLP fitness metric is once again consulted to see if there is TLP or whether ILP is the only type of parallelism available.

7 Related Work

Many of the earliest efforts on analyzing parallelism in single-thread programs focused on evaluating whether enough parallelism existed to justify parallel execution of programs on machines. Kuck et al. [8]) used static analysis to estimate the parallelism available in Fortran codes, incorporating the estimated effects of transformations such as tree height reduction and forward substitution. Nicolau et al. [13] used a dynamic, trace-based analysis, and included efforts to estimate the impact of memory disambiguation. More recent efforts such as Wall et al. [16] extended these analyses to look at the impact of wide-issue superscalar parameters; such as window-size, ability to predict branches, and resolve memory dependences. Lam et al. [10] followed on with more sophisticated versions of this style of analysis to establish the relative benefits of successively more powerful control flow models.

There have also been more recent efforts to use critical-path style analyses for analyzing and improving the performance of parallel programs; focusing on the synchronization and communication of pre-parallelized threads rather than trying to discover innate properties of the program, or trying to make suggestions to the programmer on how to optimize single-threaded code. Overeinder et al. [14] created a critical path analysis tool to analyze the parallelism between communicating message-passing MPI-threads. Yang et al. [17] created an analysis to examine communication and scheduling related bottlenecks in distributed programs.

The Cilk system [5] partly inspired this work; it analyzes the dynamic execution and synchronization of threads, employing critical path and work as metrics to assess the effect of creation and synchronization of threads on whole-program execution of parallelized Cilk code.

Austin et al. [1] presented Paragraph, a tool that tracked various dependences to find the available parallelism in an application. However the tool did not take into account memory aliasing, which is usually very hard to resolve automatically. Moreover the tool presented parallelism against the depth in the data dependence graph, hence making it very tough for the user to pin point the code regions which can or cannot be parallelized.

Eager et al. [4] presented an analytical model that attempts to bound the inefficiency caused by uneven parallelism in a program. Banerjee et al. [2] examined the analysis of parallelism in loops for the purpose of performing loop transformations.

Perhaps the most related work is COMET [9]. It uses a trace-based analysis, similar to Lam's and Wall's approach, but using source to source transformations on Fortran to extract critical path information, for the purpose of aiding the development of parallel processors. The approach created graphs that assumed that every operation is scheduled as early as its latest dependence. This effectively models the execution of the program on an infinitely wide dataflow machine. The resulting graphs tend to have spikes early on in the beginning, as instructions from many different unrelated parts of the program "fire" because they don't have input dependences. This smearing of instructions, and lack of information about which instructions come from which part of the program, makes it difficult for the programmer to use the program for advice on how to parallelize a program, although it does attest to the existence of parallelism, in some form. Our approach attacks this problem by running many many such critical path analyses across all of the program's nested regions; and by correlating and analyzing the resulting data.

8 Conclusion

In this paper we have presented p-ray, a suite of tools whose goal is to allow quick diagnosis of the parallelism of a program. p-ray is useful for both architects, who wish to gain a better understanding of the types of parallelism they will be optimizing for, and programmers, who wish to improve the performance of single-threaded programs on multi-core processors.

References

[1] T. Austin et al. “Dynamic dependency analysis of ordinary programs.” In *In Proceedings of the 19th Annual International Symposium on Computer Architecture*, 1992.

[2] U. Banerjee, et al. “Time and parallel processor bounds for fortran-like loops.” *IEEE Transactions on Computers*, Sept. 1979.

[3] S. Bell, et al. “TILE64 - Processor: A 64-Core SoC with Mesh Interconnect.” In *IEEE Solid-State Circuits Conference*, Feb. 2008.

[4] D. Eager, et al. “Speedup versus efficiency in parallel systems.” *IEEE Transactions on Computers*, Mar 1989.

[5] M. Frigo, et al. “The implementation of the cilk-5 multithreaded language.” In *PLDI 1998: Proceedings of the Conference on Programming Language Design and Implementation*, 1998.

[6] P. G. Joisha. *A type inference system for matlab with applications to code optimization*. Ph.D. thesis, Evanston, IL, USA, 2003.

[7] P. Kongetira, et al. “Niagara: a 32-way multithreaded Sparc processor.” *IEEE Micro*, March-April 2005.

[8] D. Kuck, et al. “On the number of operations simultaneously executable in fortran-like programs and their resulting speedup.” *IEEE Transactions on Computers*, Dec. 1972.

[9] M. Kumar. “Measuring parallelism in computation-intensive scientific/engineering applications.” *IEEE Transactions on Computers*.

[10] M. S. Lam et al. “Limits of control flow on parallelism.” In *ISCA*, 1992.

[11] C. Lattner et al. “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation.” In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO’04)*, Mar 2004.

[12] E. Lindholm, et al. “NVIDIA Tesla: A Unified Graphics and Computing Architecture.” *IEEE Micro*, March-April 2008.

[13] A. Nicolau et al. “Using an oracle to measure potential parallelism in single instruction stream programs.” In *MICRO*, 1981.

[14] B. Overeinder, et al. “Parallel performance evaluation through critical path analysis.” In *Lecture Notes on Computer Science 919*, 1995.

[15] M. B. Taylor, et al. “Scalar Operand Networks.” In *IEEE Transactions on Parallel and Distributed Systems*, February 2005.

[16] D. W. Wall. “Limits of instruction-level parallelism.” In *Proceedings of the Conference on Architectural support for programming languages and operating systems*, 1991.

[17] C.-Q. Yang et al. “Critical path analysis for the execution of parallel and distributed programs.” In *International Conference on Distributed Computing Systems*, Jun 1988.