An Adaptive Cost System for Parallel Program Instrumentation *

Jeffrey K. Hollingsworth[1] and Barton P. Miller[2]

[1] University of Maryland, hollings@cs.umd.edu
[2] University of Wisconsin, bart@cs.wisc.edu

Abstract. We present a new data collection cost system that provides programmers with feedback about the impact data collection is having on their application. We allow programmers to define the level of perturbation their application can tolerate and then we regulate the amount of instrumentation to ensure that threshold is not exceeded. Our approach is unique in that the type of data gathered remains constant; instead we regulate when it is collected. This permits programmers to trade speed of isolation of a performance problem for less application perturbation. We implemented this cost system in the Paradyn Performance Tools and present case studies demonstrating the accuracy of the cost system.

1 Introduction

We present a new way to manage the perturbation caused by software data collection. Our approach is based on an instrumentation cost system that ensures that data collection and analysis can be accomplished while controlling the performance overhead of the instrumentation. The unique feature of our approach is that it lets the programmer see and control the overhead introduced by monitoring rather than simply being subjected to it.

The best way to handle instrumentation overhead is to avoid introducing it in the first place. In a previous paper [4], we described a new approach to performance monitoring called *Dynamic Instrumentation*. Dynamic Instrumentation delays instrumenting an application until it is in execution, permitting dynamic insertion and alteration of the instrumentation during program execution. Enabling instrumentation only when it is needed greatly reduces the amount of data collected, and thus the overhead due to the instrumentation system.

We have developed an instrumentation cost system to ensure that data collection and analysis does not excessively alter the performance of the application being studied. The system associates a cost with different resources. Possible resources include processors, interconnection networks, disks, and data analysis workstations. The cost system is divided into two parts: predicted cost and

* Supported in part by Wright Laboratory Avionics Directorate (WLAD), Air Force Material Command, USAF, grant F33615-94-1-1525 (ARPA order B550), NSF Grants CCR-9100968 and CDA-9024618, DOE Grant DE-FG02-93ER25176, and ONR Grant N00014-89-J-1222. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of WLAD or the U.S. Government.

observed cost. Predicted cost is computed when an instrumentation request is received, and observed cost while the instrumentation is enabled.

By computing the predicted cost of instrumentation before data collection starts, it is possible to decide if the requested data is worth the cost of collection. This predictive information can be used as feedback to reduce or defer an instrumentation request. Our higher-level performance analysis tools use the cost prediction to control how aggressively they instrument a program in search of performance bottlenecks. In many cases, control of instrumentation overhead permits our tools to more quickly isolate a performance problem (see Section 6).

Although predicting the cost of data collection prior to instrumentation execution provides useful data, it is important to make sure that the actual cost of data collection matches the predicted cost. The observed cost tracks the impact the currently enabled instrumentation has on the application. To be useful, our observed cost system needs to be both cheap to compute and accurately reflect the true impact of data collection. If the observed cost exceeds predefined limits, feedback is provided to the user or higher-level tool; this feedback allows us to dynamically maintain (approximately) a fixed level of instrumentation overhead.

2 Dynamic Instrumentation and W^3 Search Model

Our recent work in performance monitoring tools has focused on two areas. First, how can we efficiently collect performance data for large, long running applications? Second, how can we help programmers to understand the source of their performance problems rather than providing them raw performance data.

Our approach, called Dynamic Instrumentation, defers instrumenting the program until it is in execution. This approach permits dynamic insertion and alteration of the instrumentation during program execution. At any time during a program's execution, a consumer of performance data can start collecting a metric for a particular combination of resources. To satisfy this request, instrumentation code is generated and inserted into the program. When performance data is no longer required, its instrumentation code is removed from the program.

Dynamic instrumentation is designed to be usable for a variety of high level tools, and so it has a simple interface. The interface is based on two abstractions: *resources* and *metrics*. Resources are the objects about which we gather performance information. Resources include processors, interconnection networks, processes, procedures, and synchronization objects. Metrics are time varying functions that characterize a program's performance; they can be computed for any subset of the resources in the system. For example, CPU time can be computed for a single procedure executing on one processor or for the entire application.

We have also been investigating how to help programmers interpret the collected performance data. The W^3 Search Model[6] is a structured methodology for programmers to quickly and precisely isolate a performance problem without having to examine a large amount of extraneous information. It is based on answering three separate questions: *why* is the application performing poorly, *where* is the bottleneck, and *when* does the problem occur. By iteratively refining the

answer to these three questions, we can precisely describe to programmers the reason their program is not performing as expected. Refining the answer to these questions requires testing different hypotheses about the source of performance problems. To deliver answers rather than asking the questions, we automate this search process. In an automated search, the tool refines the answers to these three questions by enabling and disabling the collection of performance data. The module that implements our model is called the *Performance Consultant*.

3 Cost System

With Dynamic Instrumentation, the data collected at a particular point in the program no longer remains fixed for the entire program's execution. Each time a new request for instrumentation is received, the instrumentation overhead for that point can change. Our system associates an instrumentation cost with different resources. Possible resources include processors, interconnection networks, disks, and data analysis workstations. The cost system is divided into two parts: *predicted cost* and *observed cost*. Predicted cost is computed when an instrumentation request is received, and can be used to estimate the overhead for the desired instrumentation. Observed cost is computed while the instrumentation executes and provides notification if the overhead has exceeded the user's expectations. In essence, observed cost is just another performance metric; we use the W^3 Search Model to implement much of the observed cost system.

3.1 Predicted Cost

Predicted cost is the expected overhead of collecting the data necessary to compute a metric for a particular combination of resources. We compute the predicted cost when an instrumentation request arrives, but before the instrumentation is inserted into the application. The predicted cost is expressed as the utilization of each measured resource required to collect the desired data.

In Dynamic Instrumentation, CPU time overhead is due to the insertion of instrumentation primitives at various points in the program's executable. To predict this overhead at a single point in the program we need to know: what instrumentation will be inserted, the cost of executing that instrumentation, and the frequency of execution of that point. We multiply the instrumentation overhead by the point's expected execution frequency to compute the predicted overhead. The sum of the overhead for all points is the predicted cost for an instrumentation request. Based on measurements, we know the cost of each instrumentation primitive; the difficult part is estimating their execution frequency.

Data about the frequency of instrumentation execution comes from a static estimate of procedure call frequency. This approach is at best a wild guess, but since we adjust this value based on runtime data the initial value does not need to be accurate. We associate with every point in the program an expected frequency. The initial value for each point is based on the point's type (e.g., user vs. system call). We denote the predicted cost for the application as C_{pred}.

3.2 Observed Cost

The observed cost monitors the effect of data collection on the application. Its purpose is to check that the overhead of data collection does not exceed predefined levels, and if it exceeds these levels, report it to the higher level consumers of the data. If the predicted and observed costs differ significantly, we can adjust the amount of instrumentation enabled.

Actual cost also might differ from predicted cost because of resource contention between the application and the data collection. For example, the predicted cost system does not include the memory hierarchy (e.g., caches and TLB). If the application is constrained by that resource then the impact on the performance could be significant.

Our current implementation includes only the direct CPU and cache pollution costs. Conceptually, computing the cost of executing the instrumentation is easy: we record the time spent executing the primitives. However, computing the cost of cache pollution is problematic. The difficulty is that it is impossible, without sophisticated hardware instrumentation, to know if cache lines used by the instrumentation will cause subsequent cache misses for the application. However, we can compute a bounds for the impact of cache pollution. The lower bound is that there was no cache pollution. This happens when none of cache items replaced due to instrumentation were subsequently used by the application. An upper bound is that every cache item loaded by the instrumentation code will result in a subsequent cache miss for the application[3].

Our observed cost has two values reflecting the lower and upper bound of the cache pollution. The actual cost of instrumentation should lie within this range. To compute the observed cost range we use two values:

C_{obs_direct}: The measured time spent executing the instrumentation. To efficiently compute this value we use statistical sampling.

C_{obs_cache}: The waiting time for cache misses during instrumentation code.

To compute C_{obs_cache}, we use C_{obs_ideal}. C_{obs_ideal} is the time required to execute the instrumentation assuming an ideal memory model (i.e., all memory requests are satisfied by the cache). Differences between the measured and ideal times are due to the memory hierarchy. So: $C_{obs\_cache} = C_{obs\_direct} - C_{ideal}$. To compute C_{obs_ideal}, we added an additional instruction at each instrumentation point to record the number of machine cycles required for the primitives at that point. The cycle count provides a precise measure of the instrumentation instructions executed. However, we still need to convert instruction counts to time. For the processors used in this case study, we divide the cycle times of the instrumentation instruction sequences by the clock frequency of the machine[4].

We then compute the lower and upper bounds for the observed cost as: $C_{obs\_low} = C_{obs\_direct}$, and $C_{obs\_high} = C_{obs\_direct} + C_{obs\_cache}$.

[3] While this one-to-one ratio is the worse case for direct mapped caches, for set-associative caches, the cache pollution penalty is also a function of the associativity.

[4] For super-scalar processors an approach similar to [8] will be required.

Observed cost can be viewed as just another a performance metric to characterize a type of bottleneck in a parallel program. The only difference is that the bottleneck in which we are interested was created by the data collection system rather than the programmer. We treat instrumentation as a potential bottleneck like an application bottleneck (such as too much synchronization blocking time) and use the W^3 Search Model to look for it. In the W^3 Search Model, the observed cost is expressed as additional hypotheses along the "Why" axis, that can be isolated to specific resources.

4 Evaluation of the Cost System

We ran three sequential and three parallel applications, and then compared the predicted and observed costs to the actual perturbation. For each program, we measured its performance with four different levels of instrumentation enabled:

Base: The minimum dynamic instrumentation is inserted (recording the start and end of the application).

Procedure: CPU time metrics for each procedure computed for the lifetime of the application (similar to the UNIX prof utility).

PC Base: The initial instrumentation used by the Performance Consultant to search for a bottleneck in the application is enabled.

PC Full: The Performance Consultant run in automated mode, turning on and off instrumentation as needed.

Since we were interested in assessing the accuracy of the cost system, we did not want to use the cost system to control the number of refinements being considered. However, we also did not want to overwhelm the application with instrumentation by enabling all refinements at once. As a compromise, we configured the Performance Consultant to consider ten refinements at once.

For each of the four levels of instrumentation, we recorded C_{obs_low}, C_{obs_high}, and C_{pred}. In addition we recorded two additional values:

T_{obs}: the user CPU time of the application program with the dynamic instrumentation, as measured by UNIX timing commands.

$C_{observed}$: the *timed* cost of the instrumentation, calculated as difference between T_{obs} for the current and base levels of instrumentation.

The range between C_{obs_low} and C_{obs_high} represents the bounds on the instrumentation overhead. If $C_{observed}$ is inside this range, our system accurately computed the instrumentation overhead. Differences between $C_{observed}$ and C_{pred} represent the inaccuracies in our calculations of the predicted cost. Accurate calculation of observed cost is crucial; accurate calculation of predicted cost is less critical since it can be corrected by feedback from the observed cost system.

The three sequential applications we measured (Ear, Fpppp, and Doduc) are from the floating point SPEC92 benchmark suite. Since instrumentation is currently inserted at procedure boundaries, we wanted a cross section of procedure size and procedure call frequency. The programs were run on an otherwise idle

SPARCstation 5 running at 85Mhz. PVM[2] versions of three parallel Computational Fluid Dynamics (CFD) kernels from the NAS parallel benchmarks[1] were also run.

4.1 Observed Cost

The results for the first sequential application, Ear, are shown at the top of Figure 1. The base time for this program is about 11.5 minutes, and averages 11,000 procedure calls per second during its execution. The values in the table show that the measured observed cost is within the range between C_{obs_low} and C_{obs_high} as is true for all three benchmarks. The total instrumentation overhead ($C_{observed}$) ranged from 9% to just over 42% of the CPU time of the base time to run the program in Paradyn. The Performance Consultant overhead was larger than we expected due to our naive code generator inserting duplicate copies of instrumentation to satisfy different metric requests.

Program		$C_{observed}$		C_{obs_low}			C_{obs_high}		
Version	**Time**	**Time**	**Percent**	**Time**	**Percent**	**Delta**	**Time**	**Percent**	**Delta**
Ear	687.9								
Procedure	753.4	65.5	9.5%	52.1	7.6%	(2.0)	87.9	12.8%	(-3.3)
PC Base	838.6	150.7	21.9%	124.0	18.0%	(3.9)	213.6	31.0%	(-9.1)
PC Full	978.2	290.2	42.2%	261.2	38.0%	(4.2)	453.2	65.9%	(-23.7)
Fpppp	293.8								
Procedure	309.4	15.6	5.3%	11.1	3.8%	(1.5)	19.5	6.6%	(-1.3)
PC Base	308.4	14.6	5.0%	11.9	4.1%	(0.9)	20.7	7.0%	(-2.1)
PC Full	314.8	21.0	7.1%	14.6	5.0%	(2.1)	24.2	8.2%	(-1.1)
Doduc	58.0								
Procedure	109.7	51.7	89.2%	48.1	82.9%	(6.3)	74.2	128.0%	(-38.8)
PC Base	61.2	3.2	5.6%	2.1	3.6%	(2.0)	3.6	6.2%	(-0.7)
PC Full	67.3	9.3	16.1%	7.1	12.2%	(3.8)	10.2	17.5%	(-1.4)

Fig. 1. Observed vs. Timed Overhead (seconds).

The second program was Fpppp, a quantum chemistry benchmark that does electron integral derivatives. It averaged 2,100 procedure calls per second.

The third program is Doduc, a Monte Carlo simulation of the time evolution of a thermo-hydraulical model of a nuclear reactor. This program averages 107,000 procedure calls per second. The timed observed cost for this program ranged from 5 to 89 percent of the base time to run the program using Paradyn. This program has the largest difference between C_{obs_low} and C_{obs_high}, indicating that the program is sensitive to cache perturbation.

Next, we tested our cost system with three NAS CFD benchmarks running on a network of SPARCstations 5's connected by Ethernet. The applications were configured with one master and four worker processes. We ran each program with the same same four levels of instrumentation that we used before.

A comparison of the low and high values of the observed cost and measured perturbation for these program appears in Figure 2. The Time column shows the

total time of all processes. $C_{observed}$ ranged from 1.8% to 12.7% for these three programs. For all of the programs at all levels of instrumentation, the value of $C_{observed}$ was in the range from C_{obs_low} to C_{obs_high}.

Program		$C_{observed}$		C_{obs_low}			C_{obs_high}		
Version	**Time**	**Time**	**Percent**	**Time**	**Percent**	**Delta**	**Time**	**Percent**	**Delta**
PVM_BT	892.5								
Procedure	1005.8	113.3	12.7%	96.3	10.8%	(1.9)	161.2	18.1%	(-5.4)
PC Base	914.4	21.9	2.5%	13.8	1.6%	(0.9)	26.2	2.9%	(-0.5)
PC Full	925.5	33.0	3.7%	18.6	2.1%	(1.6)	33.4	3.7%	(0.0)
PVM_LU	99.0								
Procedure	105.2	6.2	6.3%	5.2	5.3%	(1.0)	9.0	9.1%	(-2.8)
PC Base	103.4	4.4	4.4%	3.3	3.3%	(1.1)	6.3	6.3%	(-1.9)
PC Full	104.6	5.6	5.7%	3.2	3.2%	(2.4)	5.9	6.0%	(-0.3)
PVM_SP	474.5								
Procedure	483.1	8.6	1.8%	7.4	1.6%	(0.3)	9.1	1.9%	(-0.1)
PC Base	501.7	27.2	5.7%	15.7	3.3%	(2.4)	29.6	6.2%	(-0.5)
PC Full	509.2	35.0	7.4%	20.2	4.3%	(3.2)	35.0	7.4%	(0.0)

Fig. 2. Timed vs. Observed Overhead for NAS CFD Kernels (seconds).

4.2 Predicted Cost

To gauge the effectiveness of the Predicted Cost, we ran the same applications we used to study the observed cost metric, and measured the predicted cost metric. These results are based on using the static predicted cost information and do not include any compensation based on the observed cost.

The predicted cost for the Ear program is shown at the top of Figure 3. The value for the Procedure and PC Base cases are each within 6% of the observed cost. The value for the PC Full case had an error of almost 40%. This is due the Performance Consultant inserting instrumentation into a single procedure that is called thousands of times a second. The middle section of table shows the predicted cost for the Fpppp application. For all three cases, the estimated cost was within 6% of base time to run the application using Paradyn. The last part of table shows the predicted cost for the Doduc application. The errors in the Predicted Cost ranged from 4.7% to 73.1% in this case. The largest error occurred instrumenting the CPU time for all procedures.

We also compared the predicted cost data for the three PVM applications. The results are shown in Figure 3. The difference between the actual predicted running time for all three applications was within 6% for all three levels of instrumentation.

5 Using Cost to Control Searching

We now describe how the predicted cost can be used with the W^3 Search Model. First the programmer sets the tolerable instrumentation overhead. We use this

Program Version	$C_{observed}$ Time	Percent	C_{pred} Time	Percent	Delta
Ear					
Procedure	65.5	9.5%	97.3	14.4%	-4.6
PC Base	150.7	21.9%	111.6	16.2%	5.7
PC Full	290.2	42.2%	18.8	2.7%	39.5
Fpppp					
Procedure	15.6	5.3%	7.3	2.5%	2.8
PC Base	14.6	9.4%	9.4	3.2%	1.8
PC Full	21.0	7.1%	5.6	1.9%	5.2
Doduc					
Procedure	51.7	89.2%	9.3	16.1%	73.1
PC Base	3.2	5.6%	0.6	0.9%	4.7
PC Full	9.3	16.1%	0.5	0.9%	15.2

Program Version	$C_{observed}$ Time	Percent	C_{pred} Time	Percent	Delta
PVM_BT					
Procedure	113.3	12.7%	112.7	12.6%	0.1
PC Base	21.9	2.5%	1.6	0.2%	2.3
PC Full	33.0	3.7%	4.1	0.5%	3.2
PVM_LU					
Procedure	6.2	6.3%	9.6	9.7%	-3.4
PC Base	4.4	4.4%	0.0	0.0%	4.4
PC Full	5.6	5.7%	0.2	0.2%	5.4
PVM_SP					
Procedure	8.6	1.8%	0.9	0.2%	1.6
PC Base	27.2	5.7%	2.5	0.5%	5.2
PC Full	35.0	7.4%	16.2	3.4%	4.0

Fig. 3. Observed Cost vs. Predicted Cost for Six Applications (in seconds).

value to moderate how much instrumentation gets inserted. In manual search mode, the predicted cost acts as a check to see if the request associated with a hypothesis can be satisfied without undue perturbation. In automated search mode, we enumerate possible refinements, and then work down that list adding instrumentation and evaluating the results. When a test request pushes the instrumentation overhead too high, we delay requesting new instrumentation. Thus the perturbation threshold regulates how many hypotheses (potential performance problems) are considered simultaneously. Raising the threshold permits the search system to try more tests at once, but with a higher overhead. However, changing the threshold does not change what hypotheses get tested; it simply changes when they get tested.

We quantified how well our cost system regulated the perturbation of the Performance Consultant. For each application, we ran the Performance Consultant three times. The first time was with an overhead limit of 10% perturbation, the second time for a fixed limit of three refinements to the current hypothesis, and third with no limit on the refinements to the current hypothesis. The limit of three refinements provides a comparison to an alternative strategy for controlling the cost of data collection. The unlimited case measures the worst case impact of instrumentation for each application.

We evaluated two criteria about the effectiveness of our search system. First, we verified that the instrumentation cost was held within the limit set by the user (10% in this case). Second, we compared how quickly a performance problem could be isolated using each method.

For each run of an application, we compared the bottlenecks identified by the Performance Consultant. For Ear, the same performance bottleneck was found for all three cases, but the order was different. For the Doduc application, the same performance bottleneck was found in the 10% limit and three hypothesis limit, but the perturbation was so high in the unlimited case that no bottleneck was identified[5]. For Fpppp, the hypothesis limit and unlimited cases identified one procedure as the bottleneck and the cost limit identified another procedure. A CPU time profile for the application showed that both procedures consumed

[5] The application finished execution before the search was completed

enough CPU time to be flagged as bottlenecks according the thresholds.

The results for Doduc, Ear, and Fpppp are shown in Figure 4. The Search Time column reports the amount of elapsed time required for the Performance Consultant to execute its search. For all the applications, the time required for the search was least when cost was used to regulate hypothesis evaluation. The improvement in search time ranged from 29% (Fpppp) to 71% (Ear) compared to the limit of three hypotheses. The cost-based limit was able to evaluate the available hypotheses faster because different hypotheses have different costs and the cost-based limit permits evaluation of more hypotheses simultaneously, while keeping the overhead within the limit. The cost-based limit was able to identify a problem faster than the unlimited search case because it saved time by not inserting instrumentation for all possible refinements.

Program	Search	C_{obs_ideal}		
Control Method	Time	Avg	Max	Std. Dev.
Doduc				
10%	48.2	3.1%	8.6%	2.3%
3 Hypotheses	77.1	3.1%	5.6%	1.7%
unlimited	*258.7*	6.2%	41.6%	9.3%
Fpppp				
10%	52.0	1.2%	3.7%	0.9%
3 Hypotheses	73.2	0.5%	1.3%	0.3%
unlimited	227.2	1.9%	3.7%	0.4%
Ear				
10%	37.9	2.7%	8.1%	3.3%
3 Hypotheses	130.8	5.1%	17.5%	6.3%
unlimited	226.5	15.4%	17.2%	3.5%

Fig. 4. Summary of fixed vs. cost-based hypothesis evaluation.

6 Related Work

Perturbation compensation[5] reconstructs the performance of an un-perturbed execution from a perturbed one. These techniques require a trace based instrumentation system and post-mortem analysis to reconstruct the correct ordering of events. Our approach does not try to factor out perturbation; instead we try to avoid it using the predicted cost, and quantify it using the observed cost.

Pablo[7] uses an adaptive instrumentation system. The programmer specifies events to log for post mortem analysis. If the volume of data collected exceeds certain thresholds, the system will stop producing event logs and instead produce summary information. Pablo leaves the underlying instrumentation in place and controls the logging of data. However, our technique has the advantage that disabling data collection completely removes the instrumentation code and so

there is no latent perturbation due to instrumentation code that is disabled but must execute code to learn that it is disabled.

Goldberg and Hennessy[3] used the difference between the measured and predicted time of a code region to quantify the affects of the memory hierarchy. Our approach differs in two ways from theirs. First, since we need to be able to characterize the impact of small, but (potentially) frequently accessed instrumentation code blocks, we use statistical sampling instead of timers. Second, our goal is to compute the impact of the instrumentation on the original code rather than the impact of the cache on a single block.

7 Conclusion

Our cost system controls software instrumentation overhead based on feedback. We predict the amount of overhead we will cause and then use our instrumentation facility to provide information about the actual costs. The mechanisms that we have built give the programmer direct control over their instrumentation. We expose the overhead of data collection as a first class metric in Paradyn. The programmer is also given explicit control of the overhead, which controls the rate at which the performance tool searches for bottlenecks. We evaluated our model with six programs, and demonstrated that the actual instrumentation overhead was within the range of our observed cost model.

References

1. D. H. Bailey, E. Barszcz, J. T. Barton, and D. S. Browning. The NAS parallel benchmarks. *Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.
2. J. Dongarra, A. Geist, R. Manchek, and V. S. Sunderam. Integrated PVM framework supports heterogeneous network computing. *Computers in Physics*, 7(2):166–174, March-April 1993.
3. A. J. Goldberg and J. L. Hennessy. Performance debugging shared memory multiprocessor programs with MTOOL. *Supercomputing 1991*, pages 481–490, Nov. 18-22 1991.
4. J. K. Hollingsworth, B. P. Miller, and J. Cargille. Dynamic program instrumentation for scalable performance tools. *1994 Scalable High-Performance Computing Conf.*, pages 841–850, May 1994.
5. A. Malony. *Performance Observability.* PhD Dissertation, Department of Computer Science, University of Illinois, Oct. 1990.
6. B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The paradyn parallel performance measurement tools. *IEEE Computer*, 28(11), Nov. 1995.
7. D. A. Reed, R. A. Aydt, R. J. Noe, P. C. Roth, K. A. Shields, B. W. Schwartz, and L. F. Tavera. Scalable performance analysis: The pablo performance analysis environment. In A. Skjellum, editor, *Scalable Parallel Libraries Conference.* IEEE Computer Society, 1993.
8. K.-Y. Wang. Precise compile-time performance prediction of superscalar-based computers. *ACM SIGPLAN'94 Conf. on Programming Language Design and Implementation*, pages 73–84, June 20-24 1994.

This article was processed using the LaTeX macro package with LLNCS style