

Is Pair Programming an Effective Way to Learn Computer Architecture?

Edward F. Gehring
North Carolina State University
efg@ncsu.edu

Abstract

Pair programming is a concept where two programmers work side by side at the same computer, writing code jointly. One of them, called the *driver*, is in control of the keyboard and mouse. The other, called the *navigator*, observes what the driver is doing and offers advice. It is the driver's job to write the code. The navigator has a chance to observe the larger picture, evaluating the driver's code for correctness of design and implementation. Studies have shown that pair programming is very effective. Two programmers can finish a task in little over half the elapsed time that a single programmer takes. And the quality of the code—measured in terms of absence of defects—is much higher.

In the past few years, pair programming has made inroads into industry and into programming courses. However, it has not typically been used in courses that teach subjects other than programming or software engineering, nor has it been used in the analysis of experimental results. This paper reports on an experiment in a combined senior/masters level computer architecture class, using Hennessy & Patterson's *Computer Architecture: A Quantitative Approach* as a text. Students were required to implement three projects simulating various aspects of a microarchitecture (cache, branch predictor, dynamic instruction scheduler). Then they engaged in an experimental analysis to find the best configuration in a design space. They were encouraged to pair-program, and data were gathered on their experience.

1. Introduction

Pair programming is one of the twelve practices of Extreme Programming (XP), which is the best known of the “agile” software-development methodologies that have gained widespread attention in recent years. Agile methodologies attempt to mitigate some of the up-front design costs of heavyweight methodologies, which expend a lot of effort on design before code is written, and to adapt more gracefully to change in project requirements or scope.

Pair programming [1] is a style of programming in which *two* programmers work side by side at *one* computer, continuously collaborating on the same design, algorithm, code or test. One of the pair, called the *driver*, is typing at the computer or writing down a design. The other partner, called the *navigator*, has many jobs. One of the roles of the navigator is to observe the work of the driver, looking for tactical and strategic defects in the work of the driver. Tactical defects are syntax errors, typos, calls to the wrong method, etc. Strategic defects are said to occur when the team is headed down the wrong path — what they are implementing won't accomplish what it needs to accomplish. Any of us can be guilty of straying off the path. A simple, “Can you explain what you're doing?” from the navigator can serve to bring the driver back onto the right track. The navigator has a much more objective point of view and can better think strategically about the direction of the work. The driver and navigator can brainstorm on demand at any time.

An effective pair-programming relationship is very active. The driver and the navigator communicate at least every 45 seconds to a minute. It is also very important for them to switch roles periodically. Note that pair programming includes all phases of the development process — design, debugging, testing, etc. — not just coding. Experience shows that programmers can pair at any time during development, in particular when they are working on something that is complex. The more complex the task, the greater the need for two brains [2, 4].

Previous research [2, 3] has indicated that pair programming is better than individual programming in a collocated environment. Research has shown that pairs finish in about half the time of individuals and produce higher quality code. The technique has also been shown to assist programmers in enhancing their technical skills, to improve team communication, and to be more enjoyable [2, 4, 5, 6].

The observed success of pair programming in industry has led to its introduction in many computer science courses. Most experiments on introductory programming courses [7, 8] report great success. In 2001 and 2002, the author conducted an experiment in his graduate-level object-technology course that again showed [9, 10] that the pair-programming teams wrote code more quickly and reported better communication with teammates than those teams that did not pair-program.

2. Background

The author taught ECE 463, Advanced Microprocessor Design and ECE 521, Computer Design and Technology for the first time in Fall 2002. This is a combined senior-level/beginning-graduate course that is taught in a single lecture hall with the same homework and exams, but extra requirements for the graduate students. Traditionally, three simulation projects had been assigned: a cache, a branch predictor, and a dynamic instruction scheduler. These were fairly long projects, and constituted the majority of the homework in the course.

Based on the experience in his other class, the author hypothesized that pair programming should be more effective than solo programming in courses that use, but do not teach, programming skills. Students appeared to learn more with less effort, so why not?

Previous instructors cautioned the author against this approach. In previous semesters where the students had been allowed to team up in pairs, sometimes one of the members did not do any of the work. This was often revealed on the second or third project. The need to guard against freeloaders had led the instructors either to limit team programming to the undergraduates, or disallow it altogether.

However, pair programming and the tools developed for it seemed to address these concerns.

- The students would be encouraged to pair-program. With both of them in front of the same monitor, it was far less likely that one could escape doing any of the work.
- In order to record their progress, the students utilized an online tool called Bryce [11], a Web-based software-process analysis system used to record metrics for software development. The tool required times to be recorded for each session. So a student who was

going to slack off would have to lie about it frequently, and the lies would be apparent to his/her partner.

- After each project was due, each team member was asked to evaluate the other member's contribution. This appears to provide a way to assign a lower grade to a student who did little work (although in fact it was not used that way because in the case of conflicting reports, it is difficult to tell who is telling the truth).
- In most cases, the same students were not allowed to pair up for more than one project. The need to choose a different partner each time would prevent a student from taking advantage of the same student time and again. Moreover, the research on pair programming seems to indicate that it is best to switch partners frequently. (This is because one learns different things from different partners.)

In about a half-dozen cases, though, two students were allowed to pair together more than once. These were cases in which the students had a reason for wanting to maintain a partnership (e.g., they had the same class schedule) and they had submitted similar evaluations of each other on past projects.

The students were shown a short video on pair programming, and then allowed to choose whether or not to pair-program. The majority chose to pair-program on the first and third projects, but ironically not on the second project.

3. Student perceptions

All students were surveyed after the end of the semester. Responses were received from 59 of the 96 students who participated in the projects, a response rate of 61%. Students who had pair-programmed viewed the experience in a very positive light. When asked, "How was the experience of pair programming?" they ranked it 3.17 on a scale of 1 to 4. Ironically, this was the exact same value as the computer-science students gave it last year. Only 2 out of 42 this year's respondents rated it poor.

When asked about cooperation among team members, the students were even more enthusiastic. They rated it 3.48 on a scale of 1 to 4 (compared to 3.51 by the CS students). Only one respondent rated it poor. Moreover, none of the 96 students complained to the instructor about their partnership. Students were similarly positive about the communication with their partners, rating it as 3.36 on a scale of 1 to 4 (compared to 3.43 for the CS students).

Table 1. Student Perceptions of Pair Programming

	Vy. Good	Good	Fair	Poor	Avg.
How was the experience of pair programming?	20	11	9	2	3.17
How was the cooperation between your team members?	26	11	4	1	3.48
How was the communication with your team?	20	18	3	1	3.36

4. Quality of work

In a course such as this, grades are a good measure of the quality of student work. A statistical analysis was performed using analysis of variance (ANOVA) with student grades as the response variable and paired or not paired as the explanatory variable. We counted each grade of a non-paired individual, and each grade of a programming pair as a single observation. The instructor reserved the right to assign different grades to each member of a programming pair, based on feedback from the two members. However, there was no case in which different grades were assigned to the two partners.

We were not sure how to handle the separate projects. First, we decided to do a separate analysis for each project assuming that the results would be the same. This was not the case. For Project 1, pair programming produced significantly better results (p -value = 0.0066) than solo programming, but there was no significant difference for Projects 2 and 3. To obtain a single result, all of the data was merged, and a new variable was created to indicate the project. Then ANOVA was run treating the project as a replication variable (three replications). This analysis resulted in no significant difference at the 95% confidence level (p -value = .065) between pair programming and regular programming. There was, however, significance at the 90% confidence level.

Table 2. Grade Comparison for Pair vs. Solo Programming

	Project 1				Project 2				Project 3			
	<i>n</i>	Mean grade	Std. dev.	<i>p</i>	<i>n</i>	Mean grade	Std. dev.	<i>p</i>	<i>n</i>	Mean grade	Std. dev.	<i>p</i>
Paired	41	94.7	0.92	0.007	23	99.33	3.95	0.664	28	94.68	9.37	0.17
Solo	19	90.2	1.39		29	96.28	7.87		17	91.18	15.75	
All	101	93.2	8.87		75	97.54	8.53		73	91.42	17.00	

Note that the n for all students is derived by taking the number of pairs \times 2 + the number of solo programmers.

However, it should be noted that statistical significance is difficult to obtain in pair-programming experiments. First there is the limited number of observations: Each project was done by 73 to 101 students, which means that even in the best case we would have only about 30 pairs' performance to compare with the performance of about 30 solo programmers. Then there is the fact that grades for programming projects usually do not vary much: For Project 2, the first quartile, median, third quartile, and high grade were all 100%. The situation was not quite so uniform for Project 1 (first quartile of 90, third quartile of 98) or Project 3 (first quartile of 94, third quartile of 100). In this environment, the fact that statistical significance was found for one of the projects is surprising. Project 1 (a simulator of a cache with prefetching) was definitely the most difficult of the projects, and this may have served to spread out the grades enough for significance to be obtained.

From Table 2, we also observe that fewer students paired for Projects 2 and 3 than for Project 1. Table 1 would suggest that this is not due to any disenchantment with pair programming. Three factors may be at work. First, after Project 1, students were more comfortable with the computing environment and with writing simulation programs, leading to less perceived "need"

to pair with someone else. Second, as noted above, Projects 2 (branch prediction) and 3 (out-of-order execution) were not as difficult as Project 1. Finally, students' schedules fill up as the semester progresses, and they are less able to make time to meet with a partner.

Further support for the theory that pair programming helped is the fact that the students who did not pair for any of the three projects had an average score for the rest of their homework and exams in this class of 79.4%, while those who paired for all three projects had an average of only 77.0% on their other work. This suggests that while the solo programmers were in some sense "better computer architecture students," they still scored lower on the projects. However, this difference was not significant ($p = .531$). We might be able to establish significance with an analysis of covariance of the project scores and final grades with the final adjusted grade as the covariate, but this analysis could not be performed in time for publication.

5. Conclusions

Our experience shows that pair programming is a useful technique even in courses where programming is not taught. Through the process of pair programming and the tools to track progress, our students managed to avoid the problems that had plagued programming teams in this class in earlier semesters. The pair programmers received a higher average grade on all three projects, and despite the fact that most of the grades were quite high, the difference between pair programmers' grades and solo programmers' grades is significant at the 95% confidence interval for Project 1, and at the 90% confidence level for all the projects combined.

Acknowledgments

The author wishes to acknowledge the contributions of Computer Science graduate students Nachiappan Nagappan, who wrote the Bryce tool for data collection, and Vinay Ramachandran, who wrote the Web-based program that collected the reports on students' perceptions of their partners, as well as Dr. Laurie Williams, their supervisor. Nagappan was funded by NSF DUE CCLI grant #0088178. Ramachandran was funded by the Center for Advanced Computing and Communication (CACC), a membership-based industry/university cooperative research center co-located at North Carolina State University and Duke University. The statistical analyses were performed by Janet L. Bartz, a graduate student in statistics, under the supervision of Dr. Christopher Basten.

Bibliography

- [1] L. A. Williams, "Pair programming," <http://www4.ncsu.edu/~lawilli3/PP/PairProgramming.pdf>
- [2] L. A. Williams, "The Collaborative Software Process PhD Dissertation", Department of Computer Science, University of Utah. Salt Lake City, 2000.
- [3] J. T. Nosek, "The case for collaborative programming", *Communications of the ACM* 41:3, March 1998, p. 105-108.
- [4] L. A. Williams, and R. Kessler, *Pair Programming Illuminated*, Boston, MA: Addison Wesley, 2002.
- [5] L. Williams, R. Kessler, W. Cunningham, and R. Jeffries, "Strengthening the case for pair-programming", *IEEE Software* 17:4, July/Aug 2000, pp. 19-25.

- [6] A. Cockburn, and L. Williams, "The costs and benefits of pair programming", in *Extreme Programming Examined*, Succi, G., Marchesi, M. eds., pp. 223–248, Boston, MA: Addison Wesley, 2001.
- [7] H. Bullock, J. Fernald, C. McDowell, and L. Werner "The effects of pair programming on performance in an introductory programming course," *Proc. 33rd SIGCSE Technical Symposium on Computer Science Education*, Covington, KY, Feb. 27–Mar. 3, 2002, pp. 38–42.
- [8] H. Bullock, J. Fernald, C. McDowell, and L. Werner, "The impact of pair programming on student performance and pursuit of computer science-related majors," *Proc. International Conference on Software Engineering 2003* [to appear].
- [9] P. P. Baheti, E. F. Gehringer, and David Stotts, "Exploring the efficacy of distributed pair programming," *Proc. XP Agile Universe 2002*, Lincolnshire, IL, Aug. 4–7, 2002
- [10] P. P. Baheti, L. A. Williams, Edward F. Gehringer, and David Stotts, "Exploring pair programming in distributed object-oriented team projects," *Proc. OOPSLA (Object-Oriented Programming Languages, Systems, and Applications), Educators' Symposium*, Seattle, Nov. 5, 2002.
- [11] <http://bryce.csc.ncsu.edu/tool/default.jsp>

EDWARD F. GEHRINGER

Edward Gehringer is an associate professor in the Department of Electrical and Computer Engineering and the Department of Computer Science at North Carolina State University. He has been a frequent presenter at education-based workshops in the areas of computer architecture and object-oriented systems. His research interests include architectural support for persistence and large object systems, memory management and memory-management visualization, and garbage collection. He received a B.S. from the University of Detroit(-Mercy) in 1972, a B.A. from Wayne State University, also in 1972, and the Ph.D. from Purdue University in 1979.