DOI:10.1145/1866739.1866752

Guy L. Steele Jr.

Interview

An Interview with Frances E. Allen

Frances E. Allen, recipient of the 2006 ACM A.M. Turing Award, reflects on her career.

ACM FELLOW FRANCES E. ALLEN, recipient of the 2006 ACM A.M. Turing Award and IBM Fellow Emerita, has made fundamental contributions to the theory and practice of program optimization and compiler construction over a 50-year career. Her contributions also greatly extended earlier work in automatic program parallelization, which enables programs to use multiple processors simultaneously in order to obtain faster results. These techniques made it possible to achieve high performance from computers while programming them in languages suitable to applications. She joined IBM in 1957 and worked on a long series of innovative projects that included the IBM 7030 (Stretch) and its code-breaking coprocessor Harvest, the IBM Advanced Computing System, and the PTRAN (Parallel Translation) project. She is an IEEE Fellow, a Fellow of the Computer History Museum, a member of the American Academy of Arts and Sciences, and a member of the U.S. National Academy of Engineering.

ACM Fellow Guy L. Steele Jr. visited Allen at her office in the IBM T.J. Watson Research Center in 2008 for an extended oral interview. The complete transcript of this interview is available in the ACM Digital Library; presented here is a condensed version that highlights Allen's technical accomplishments and provides some anecdotes about her colleagues.

PHOTOGRAPH BY FRANK BECERRA, JR. / THE JOURNAL NEWS

Fran Allen on CS: "It's just such an amazing field, and it's changed the world, and we're just at the beginning..."

Your first compiler work was for IBM Stretch.[a]

Yes. In 1955, IBM recognized that to be 100 times faster than any machine existing or planned at the time, the major performance problem to overcome was latency to memory. The advanced ideas and technologies they put into Stretch were to address that problem. Six instructions could be in flight at the same time. The internal memory was interleaved, and data would arrive out of order—data and instructions were both stored in this memory. They built a very complex buffering system and look-ahead. John Cocke, when he came in 1956, was put in charge of the look-ahead for instructions. It was also architected to have precise interrupts. So

a See the December 2010 *Communications* Historical Perspectives column "IBM's Single-Processor Supercomputer Efforts" for more discussion of the IBM Stretch supercomputer.

the look-ahead unit was a phenomenal piece of hardware.

How many copies of Stretch were built?

Eight or nine. The original was built for Los Alamos and shipped late. Then they discovered its performance was about half of what was intended.

But still, 50 times...

Meanwhile, the underlying technology had changed. T.J. Watson got up at the Spring Joint Computer Conference and announced they would not build any more Stretch machines, and apologized to the world about our failure. But it was recognized later that the technology developed in building Stretch made a huge difference for subsequent machines, particularly the 360. A lot of people went from Stretch to the 360, including Fred Brooks.

What was your connection with Stretch?

My role was on the compiler. When I joined IBM in 1957, I had a master's degree in mathematics from the University of Michigan, where I had gone to get a teaching certificate to teach high school math. But I had worked on an IBM 650 there, so I was hired by IBM Research as a programmer. My first assignment was to teach FORTRAN, which had come out in the spring of that year.

Did you already know FORTRAN, or were you learning it a week ahead, as professors often do?

Yeah, a week ahead [laughs]. They had to get their scientists and researchers to use it if they were going to convince customers. I had a pretty unhappy class, because they knew they could do better than any high-level language could.

It was recognized later that the technology developed in building Stretch made a huge difference for subsequent machines.

Did you win them over?

Yes—and won myself over. John Backus, who led the FORTRAN project, had set two goals from the beginning: programmer productivity and application performance. I learned all about the compiler as part of teaching this course.

Did you ever work on that compiler yourself?

I was reading the code in order to do the training. It set the way I thought about compilers. It had a parser, then an optimizer, then a register allocator. The optimizer identified loops, and they built control flow graphs.

The Stretch group recognized that the compiler was going to be an essential part of that system. A bunch of us in research were drafted to work on it. The National Security Agency [NSA] had a contract with IBM to build an add-on to Stretch, for code-breaking. Stretch would host the code-breaking component, and there was a large tape device, tractor tape, for holding massive amounts of data.

This was Stretch Harvest?

Yes. There was going to be one compiler for Stretch Harvest that would take FORTRAN, and the language I was working on with NSA for code-breaking, called Alpha, and also Autocoder, which was similar to COBOL.

A single compiler framework to encompass all three languages?

Yes, three parsers going to a high-level intermediate language, then an optimizer, then the register allocator. It was an extraordinarily ambitious compiler for the time, when even hash tables were not yet well understood. One compiler, three source languages, targeted to two machines, Stretch and Harvest. In addition to managing the optimizer group, I was responsible for working with the NSA on designing Alpha. I was the bridge between the NSA team, which knew the problem...

And never wanted to tell you completely what the problem is.

They told me not at all, but it didn't matter. I was pretty clueless about everything down there, and on purpose. "NSA" was not a term that was known. While I was on the project, two guys went to Moscow, just left, and it hit the *New York Times*, and that's when I learned what it was about. It was a very carefully guarded activity. The problem was basically searching for identifiers in vast streams of data and looking for relationships, identifying *k*-graphs and doing statistical analysis. Any single Harvest instruction could run for days, and be self-modifying.

The most amazing thing about that machine is that it was synchronized. Data flowed from this tape system through memory, through the streaming unit, to the Harvest unit, the streaming unit, back to memory, and back out onto the data repository, and it was synchronized at the clock level.

The data was coming from listening stations around the world, during the Cold War. I spent a year at NSA installing the system; during that year, the Bay of Pigs and the Cuban Missile Crisis happened, so it was a very tense period. I assume most of the data was in Cyrillic. But Alpha could deal with any data that had been coded into bytes.

I wrote the final acceptance test for the compiler and the language. I wrote the final report and gave it to them and never saw it again, which I regret.

What did you do next?

John Cocke was enamored with building the fastest machine in the world, and Stretch had been an announced public failure. When I finished with Harvest, Stretch was already done. I could have gone and worked on the 360. I didn't particularly want to do that; it was a huge project spread around the world. John wanted to take another crack at building the fastest machine in the world, so I joined him on a project called System Y. This time the compiler was built first. Dick Goldberg was the manager and did the parser, I did the optimizer, and Jim Beatty did the register allocator. We had a very nice cycle-level timing simulator. We built what was called the Experimental Compiling System.

What became of System Y?

It changed into ACS [Advanced Computing System], which was even-

tually canceled [in 1969] by Armonk, by headquarters, which we should have known would happen, because it was not 360. But we developed things that subsequently influenced the company a lot. We did a lot with branch prediction, both hardware and software, and caching, and machine-independent, language-independent optimizers. John, after being very disappointed about not being able to build the fastest machine in the world, decided he would build the best cost-performance machine. That was where the PowerPC came from—the 801 project.

After ACS, I took an unhappy digression from my work on compilers. I was assigned to work on FS, the famous "Future System" of IBM. It was so bad on performance, I wrote a letter. FS took two round trips to memory to fetch any item of data, because it had a very high-level intermediate form as the architected form for the machine.

Should I be reminded of the Intel 432, the processor designed for Ada? It had a very high-level architecture that turned out to be memory-bound, because it was constantly fetching descriptors from memory.

Yes. We aren't very good about passing on the lessons we've learned, and we don't write our failures up very well.

It's harder to get a failure published than a success.

But there are a lot of lessons in them. After fuming about FS for a few months, I wrote a letter to somebody higher up and said, "This isn't going to work," and why, and that was the wrong thing to say. So I was kind of put on the shelf for a while. But then I did a lot of work with a PL/I compiler that IBM had subcontracted to Intermetrics.

The compilers you worked on—such as the ACS compiler and the PL/I compiler in the 1970s—what languages were those implemented in?

Some of them were implemented in FORTRAN, some in PL/I, and some were in assembly language.

How about Alpha?

That was in the assembly language for Stretch.

Your 1976 *Communications* paper with John Cocke contains some actual PL/I code that represents sets as bit vectors, and propagates sets around the program control flow graph. The intersections and unions of the sets were just PL/I & and | operators, which makes the code concise and easy to read. You have said that PL/I was a complicated language to compile, but it seems to have expressive power.

Any single Harvest instruction could run for days, and be self-modifying.

Yes, it was really very useful for writing optimizers and compilers. The data flow work came from early FORTRAN and their use of control flow graphs. On Project Y we built control flow graphs and developed a language about the articulation points on the graph, abstracting away from DO loops into something more general, then optimizing based on a hierarchy of these graphs, making the assumption that they represented parts of the program that were most frequently executed.

When did you first start using that bit-vector representation?

Right at the beginning of the ACS project. "Graph intervals" was a term that John had come up with, but then I wrote the paper and carried the idea further. Then Mike Harrison came, and we were struggling with the problem that we had no way of bounding the computation of the flow of information in such a graph.

In some of your papers, you talked about earlier monotonic relaxation techniques, but they had very large theoretical bounds.

Yes, but I wasn't much concerned, because I knew that real programs don't have those, and Mike agreed. Jeff Ullman did some analysis on programs. That did get a better bound, but that analysis didn't produce a structure against which one could actually make transformations.

The graph interval decomposition improved the theoretical cost bounds of the algorithm by guiding the order—but if I hear you correctly, the interval structure is just as important, perhaps more important, for guiding the transformations than for doing the analysis?

Yes. People who were focusing on the theoretical bounds missed, I think, the importance of leaving a framework in which one could make the transformations. But then something really exciting happened. A student of Knuth's, [Robert] Tarjan, developed a way to map this problem into a spanning tree.

Nodal graphs could be decomposed into spanning trees plus back edges.

Yes! It was startling. Great things sometimes look simple in retrospect, but that solved that part of structuring the bounds of subsequent algorithms' analysis and transformation.

So Tarjan's work played a role in this?

Yes, I don't think he knew it, but as soon as he published that, it was just obvious that we should abandon graph intervals and go there.

Could you talk about Jack Schwartz?

Jack spent a summer at ACS and had a huge influence. He wrote a number of wonderful papers on optimizing transformations, one being "Strength reduction, or Babbage's differencing engine in modern dress." Jack had a list of applications for strength reduction, which we in compilers never took advantage of. He and John wrote a big book, never published but widely circulated, on a lot of this work. I spent a year in the Courant Institute—I taught graduate compilers. And Jack and I were married for a number of years. So it was a good relationship all around.

What did you think about SETL [a programming language developed by Schwartz]?

It wasn't the right thing for that time, but it may be an interesting language to go back and look at now that we're mired in over-specifying.

Gregory Chaitin's classic PLDI paper on "Register Allocation and Spilling via Graph Coloring" contains a substan-

tial chunk of SETL code, four and a half pages, that implements the algorithm.

I liked SETL and was amazed that they got some good compiling applications out of it. In the context of multicores and all the new challenges that we've got, I like it a lot—it's one instance of specifying the problem at such a high level that there's a good possibility of being able to target multiple machines and to get high performance from programs that are easy to write.

I have a story about register allocation. FORTRAN back in the 1950s had the beginnings of a theory of register allocation, even though there were only three registers on the target machine. Quite a bit later, John Backus became interested in applying graph coloring to allocating registers; he worked for about 10 years on that problem and just couldn't solve it. I considered it the biggest outstanding problem in optimizing compilers for a long time. Optimizing transformations would produce code with symbolic registers; the issue was then to map symbolic registers to real machine registers, of which there was a limited set. For high-performance computing, register allocation often conflicts with instruction scheduling. There wasn't a good algorithm until the Chaitin algorithm. Chaitin was working on the PL.8 compiler for the 801 system. Ashok Chandra, another student of Knuth's, joined the department and told about how he had worked on the graph coloring problem, which Knuth had given out in class, and had solved it—not by solving the coloring problem directly, but in terms of what is the minimal number of colors needed to color the graph.

The Stretch look-ahead was designed on bar napkins, particularly in the Old Brauhaus in Poughkeepsie.

Greg immediately recognized that he could apply this solution to the register allocator issue. It was a wonderful kind of serendipity.

Anything else we should know about John Cocke?

He had a major impact on everybody. Let me talk about his style of work. He didn't write anything, and giving a talk was exceedingly rare and painful for him. He would walk around the building, working on multiple things at the same time, and furthered his ideas by talking to people. He never sat in his office—he lost his tennis racket one time for several months and eventually found it on his desk. If he came into your office, he would start drawing and pick up the conversation exactly where he had left off with you two weeks ago!

So he was very good at co-routining!

Yes, he could look at a person and remember exactly the last thing he said to them. And people used to save his bar napkins. He spent a lot of time in bars; he liked beer. He would draw complex designs on napkins, and people would take the napkins away at the end of the evening. The Stretch look-ahead was designed on bar napkins, particularly in the Old Brauhaus in Poughkeepsie.

You also knew Andrei Ershov.

He did some marvelous work in the Soviet Union. Beta was his compiler, a really wonderful optimizing compiler. He had been on the ALGOL committee.

He had an earlier project that he called Alpha, not to be confused with the Alpha language you did for Stretch, right?

No, it was totally unrelated. But later we read his papers. Then in 1972 he couldn't travel, because he wasn't a party member, so he had a workshop in Novosibirsk and invited a large number of people. It was broader than compilers, but there was a big focus on compilers, and we picked up some things from his work.

Ershov also worked with people in China. When the curtain came down between the Soviet Union and China, the Chinese group then didn't have access to Ershov's work. Jack and I were invited to China in 1973 to lecture. Mao was still alive, and a lot of the institutes and universities were pretty much closed. There was a science institute in Peking and in Shanghai, where we gave talks on compilers, and we looked at the machines there, which were really quite primitive. The compiler they were running on the machine in Peking was on paper tape. I recognized, looking at the code, that it was essentially Ershov's compiler. So the people in China were really quite concerned about being cut out of the advances in computing. This is a conjecture I've only recently arrived at, why we in particular in the U.S. were asked to come: it was a connection through the technology that the three groups shared. We were very involved with Ershov and his group. He and his family wanted to leave the Soviet Union, and they lived with us in our home for about a year.

You actually had two projects called "Experimental Compiling System." What was the second one like?

Its overall goals were to take our work on analysis and transformation of codes, and embed that knowledge in a schema that would advance compiling. I wish we had done it on Pascal or something like that.

PL/I was that difficult a language?

Yes, it was the pointers and the condition handling—those were the big problems. This was another bold project, and my interest was mostly in the generalized solution for interprocedural analysis—but also putting what we knew into a context that would make writing compilers easy and more formal, put more structure into the development of compilers. We already had a lot of great algorithms which we could package up, but this was to build a compiler *framework* where the methods that we already had could be used more flexibly.

Did lessons learned from this project feed forward into your PTRAN work?

The interprocedural work did, absolutely, and to some extent the work on binding. It sounds trivial, but constant propagation, getting that right, and being able to take what you know and refine the program without having to throw things away and start over.

Let's talk about PTRAN. Two papers came out in 1988: your "Overview of the PTRAN Analysis System" and "IBM Parallel FORTRAN". It's important to distinguish these two projects. IBM Parallel FORTRAN was a product, a FORTRAN augmented with constructs such as PARALLEL LOOP and PARALLEL CASE and ORIGINATE TASK. So the FORTRAN product is FORTRAN with extra statements of various kinds, whereas with PTRAN, you were working with raw FORTRAN and doing the analysis to get parallelism.

Right.

What was the relationship between the two projects? The IBM Parallel FORTRAN paper cites your group as having provided some discussion.

The PTRAN group was formed in the early 1980s, to look first at automatic vectorization. IBM was very late in getting into parallelism. The machines had concurrency, but getting into explicit parallelization, the first step was vectorization of programs. I was asked to form a compiler group to do parallel work, and I knew of David Kuck's work, which started in the late 1960s at the University of Illinois around the ILLIAC project. I visited Kuck and hired some of his students. Kuck and I had a very good arrangement over the years. He set up his own company—KAI.

Kuck and Associates, Inc.

Right. IBM, at one point later on, had them subcontracted to do some of the parallelism. They were very open about their techniques, with one exception, and they were the leaders early on. They had a system called Parafrase, which enabled students to try various kinds of parallelizing code with FORTRAN input and then hooked to a timing simulator back-end. So they could get real results of how effective a particular set of transformations would be. It was marvelous for learning how to do parallelism, what worked and what didn't work, and a whole set of great students came out of that program. In setting up my group, I mostly hired from Illinois and NYU. The NYU people were involved with the Ultracomputer, and we had a variant of it here, a project called RP3, Research Parallel Processor Prototype, which was an instantiation of their Ultracomputer.

Another thing I think was a very big step was not only identifying parallelism, but identifying *useful* parallelism.

The Ultracomputer was perhaps the first to champion fetch-and-add as a synchronization primitive.

Yes. A little history: The Ultracomputer had 256 processors, with shared distributed memory, accessible through an elaborate switching system. Getting data from memory is costly, so they had a combining switch, one of the big inventions that the NYU people had developed. The fetch-and-add primitive could be done in the switch itself.

Doing fetch-and-add in the switch helped avoid the hot-spot problem of having many processors go for a single shared counter. Very clever idea.

Very, very clever. So IBM and NYU together were partners, and supported by DARPA to build a smaller machine. The number of processors got cut back to 64 and the combining switch was no longer needed, and the project kind of dragged on. But my group supplied the compiler for that. The project eventually got canceled.

So that was the background, in IBM Research and at the Courant Institute. But then the main server line, the 370s, 3090s, were going to have vector processors.

Multiple vector processors as well as multiple scalar processors.

Yes. And the one that we initially worked on was a six-way vector processor. We launched a parallel translation group, PTRAN. Jean Ferrante played a key role. Michael Burke was involved; NYU guy. Ron Cytron was the Illinois guy. Wilson Hsieh was a co-op student. Vivek Sarkar was from Stanford, Dave Shields and Philippe Charles from NYU. All of these people have gone on to have some really wonderful careers. Mark Wegman and Kenny Zadeck were not in the PTRAN group but were doing related work. We focused on taking dusty decks and producing good parallel code for the machines—continuing the theme of language-independent, machine-independent, and do it automatically.

"Dusty decks" refers to old programs punched on decks of Hollerith cards. Nowadays we've got students who have never seen a punched card.

We also went a long way with working with product groups. There was a marvelous and very insightful programmer, Randy Scarborough, who worked in our Palo Alto lab at the time. He was able to take the existing FORTRAN compiler and add a little bit or a piece into the optimizer that could do pretty much everything that we could do. It didn't have the future that we were hoping to achieve in terms of building a base for extending the work and applying it to other situations, but it certainly solved the immediate problem very inexpensively and well at the time. That really helped IBM quickly move into the marketplace with a very parallel system that was familiar to the customers and solved the problem. Disappointing for us, but it was the right thing to have happen.

Did PTRAN survive the introduction of this product?

Yes, it survived. The product just did automatic vectorization. What we were looking at was more parallelism in general.

One particular thing in PTRAN was looking at the data distribution problem, because, as you remarked in your paper, the very data layouts that improve sequential execution can actually harm parallel execution, because you get cache conflicts and things like that.

Yes.

That doesn't seem to be addressed at all by the "IBM Parallel FORTRAN" paper. What kinds of analysis were you doing in PTRAN? What issues were you studying?

Well, two things about the project.

We worked on *both* the theory and abstraction, building up methods that were analyzable and could be reasoned about, *and* implementing them. I insisted in this project that, if someone on the systems side showed me a piece of code, I would say, "Can you describe this in a paper? How would other people know about this?" If they were on the theoretical side, I would say, "Can you implement it? Show me the implementation."

The idea of trying to change a program into a functional program was something that I was trying to push. We could do much better analysis even for just plain straight optimization if we could name the values but not burden it with the location, apply a functional paradigm to it.

We could trace that idea back to your early work on strength reduction, when you were making hash names for intermediate values.

Yes. The value contributes to the answer, but where that value resides should be irrelevant to the writer of the program.

Apparently, just convincing the early programmers of that was one of your early successes. FORTRAN is good enough; you don't need to keep track of every single machine register yourself.

That's right. So I had that challenge out there. We needed to try and recast the program as close as we could to functional.

Another thing I think was a very big step was not only identifying parallelism, but identifying *useful* parallelism.

Another problem: say that one of the optimizations is constant propagation. For some variable deep in the code,

That was a problem we struggled with early on: How do you avoid redoing the analysis?

there is a constant that you have recognized could replace the use of the variable. You then know, say, which way a branch is going to go. You've built up this infrastructure of analysis and you're ready to make the transformation—but then the results of the analysis are obsolete, so you have to start again. That was a problem we struggled with early on: How do you avoid redoing the analysis? It got particularly bad with interprocedural activities.

Is there some simple insight or overarching idea that helps you to avoid having to completely redo the computation?

Vivek Sarkar was one of the key people on that, but Dave Kuck—this is at the core of KAI's work, too. That group described it as "the oracle." You assign costs to each of the instructions, and you can do it in a hierarchical form, so this block gets this cost, and this block has that cost, and then do a cost analysis. This is the time it's going to take. Then there's the overhead cost of having the parallelism.

Earlier, you said that Kuck was very open about everything he was doing, with one exception—

The oracle! "What have you got in that thing?" [laughs] "We're not going to tell you!" So we built our own variant of it, which was a very powerful technique.

What else should we mention?

We talked about the NSA work that wasn't published. That was, for me, a mind-changer that led to my feeling very strongly about domain-specific languages.

Are you for them or against them?

For them!

Oh, okay. Let's be clear! [laughs]

Good! [laughs]

I'm going to try something very foolish: summarize your career in one paragraph, then ask you to critique it. *A major focus of your career has been that, rather than inventing new programming languages or language features and trying to get people to program in them, you focused on taking programs as they are written, or as programmers like to write them, and making them run really effectively and efficiently on target machines. One of the many ways you do this, but a very important one, is to do many kinds of sophisticated analysis and optimization of the code and to find out as much as you can about the characteristics of the program without actually running it. So these tend to be static techniques, and very sophisticated ones. While you have worked with and pioneered quite a number of them, some of the most interesting involve using graphs as a representation medium for the program and using a strategy of propagating information around the graph. Because a program can be represented as a graph in more than one way, there's more than one way in which to propagate that information. In some of these algorithms in particular, the information that's being propagated around the graph is in the form of sets—for example, sets of variable names. As a strategy for making some of these algorithms efficient enough to use, you've represented sets as bit vectors and decomposed the graphs using interval analysis in order to provide an effective order in which to process the nodes. In doing this, you have built a substantial sequence of working systems; these aren't just paper designs. You build a great system, and then you go on and build the next one, and so on. These all actually work on code and take real programs that aren't artificial benchmarks and make them run.*

That's really very good. There's one thing: the overall goal of all of my work has been the FORTRAN goal, John Backus' goal: user productivity, application performance.

Now, three goofy questions. What's your favorite language to compile?

FORTRAN, of course!

What's your favorite language to program *in*?

I guess it would have to be FORTRAN.

Okay, now, if you had to build a compiler that would run on a parallel machine, what language would you use to write that compiler?

Probably something like SETL or a functional language. And I'm very intrigued about ZPL. I really liked that language.

Any advice for the future?

Yes, I do have one thing. Students aren't joining our field, computer science, and I don't know why. It's just such an amazing field, and it's changed the world, and we're just at the beginning of the change. We have to find a way to get our excitement out to be more publicly visible. It *is* exciting—in the 50 years that I've been involved, the change has been astounding.

Recommended Reading

Buchholz, W., Ed.
Planning a Computer System: Project Stretch. **McGraw-Hill, 1962; http://edthelen.org/comp-hist/IBM-7030-Planning-McJones.pdf**

Allen, F.E. and Cocke, J.
A catalogue of optimizing tranformations. In R. Rustin, Ed., *Design and Optimization of Compilers*. Prentice-Hall, 1972, 1–30.

Allen, F.E.
Interprocedural data flow analysis. In *Proceedings of Information Processing 74*. IFIP. Elsevier/North-Holland, 1974, 398–402.

Allen, F.E. and Cocke, J.
A program data flow analysis procedure. *Commun. ACM 19*, 3 (Mar. 1976), 137–147; http://doi.acm.org/10.1145/360018.360025

Allen, F.E. et al.
The Experimental Compiling System. *IBM J. Res. Dev. 24*, 6 (Nov. 1980), 695–715.

Allen, F.E.
The history of language processor technology at IBM. *IBM J. Res. Dev. 25*, 5 (Sept. 1981), 535–548.

Allen, F.E. et al.
An overview of the PTRAN analysis system for multiprocessing. In *Proceedings of the 1st International Conference on Supercomputing* (Athens, Greece, 1988), Springer-Verlag, 194–211. Also in J. Par. Dist. Comp. 5 (Academic Press, 1988), 617–640.

Recommended Viewing

Allen, F.E.
The Stretch Harvest compiler. Computer History Museum, Nov. 8, 2000. Video, TRT 01:12:17; http://www.computerhistory.org/collections/accession/102621818

The IBM ACS System: A Pioneering Supercomputer Project of the 1960s. Speakers: Russ Robelen, Bill Moone, John Zasio, Fran Allen, Lynn Conway, Brian Randell. Computer History Museum, Feb. 18, 2010; Video, TRT 1:33:35; http://www.youtube.com/watch?v=pod53_F6urQ