

---

# AMD DEVELOPER INSIDE TRACK

## *AMD X86 OPEN64 COMPILER SUITE TEAM INSIGHTS*



This video features AMD's Open64 Compiler Team talking about Open64 origins, unique features of this compiler, such as multi-core scalability optimizations and other important optimization flags you should not miss such as loop nest optimizations (LNO), and interprocedural analysis (IPA).

---

### TRANSCRIPTS

#### ***Mike Vermeulen, Engineering Manager for the Open64 Development team***

Hi I'm Mike Vermeulen. I'm the engineering manager of the Open64 Compiler Team. Myself, Roy and Michael will be telling you a little bit about Open64 and also answering some of the questions that we've gotten from Developer Inside Track.

So, to start with actually, the first question is, "Well what is Open64?"

Open64 is a compiler suite. It's got C, C++ and FORTRAN compilers. These compilers are set up to have a pretty advanced set of optimizations and actually runs on a number of different architectures and gets high performance code on those architectures. Open64 started as the MIPS Pro compiler, so it was a proprietary compiler set created by SGI. SGI released these compilers, actually relicensed them in 2000 under the GNU public license, the GPL. After that point a number of university researchers and companies actually picked up these compilers. They continued to add improvements and enhancements

to advance it forward as a set of technology. AMD actually also joined this community in 2008 picking up the Open64 compilers and our work initially has been on using the compilers and improving, particularly getting high performance on AMD platforms to sort of being able to exploit the hardware platforms for Shanghai, Barcelona, Istanbul and newer platforms as well.

### **Roy Ju, Architect in the Open 64 compiler team**

*Sharon: Roy, can you tell us what do you do here at AMD and how long have you been with AMD?*

Roy: My name is Roy Ju, and I am an AMD Fellow. I have been working on compiler optimizations for about 20 years. I have been with AMD for over 3 years and currently I am an architect in the Open64 compiler team.

*Sharon: So, the Open64 compiler, can you describe any of the unique features that are in that compiler?*

Roy: Open64 has many state-of-the-art compiler technologies. But we have put a big focus on the compiler optimizations to reduce memory bandwidth consumption. We group such optimizations under a new **-mso** option, which stands for **multi-core scalability optimizations**.

*Sharon: How does the memory bandwidth issue arise?*

Memory bus is often a performance bottleneck, in particular in many data center server applications. This problem is particularly pronounced in a multi-core processor, where some resources, such as memory bus, L3 cache, and DRAM controller, are shared among different cores within a processor.

*Sharon: Ok, so how does an application incur a memory bandwidth issue more than just needing a lot of data?*

Roy: When an app needs to access data, it will go through the memory sub-system to get it. The quantity of data that gets fetched each time is fixed. It could be a page or a cache line. If you use only a small fraction of the data, you are wasting some memory bandwidth to bring in the data that you don't really need. When the memory traffic is light, it's probably no big deal. However, in a multi-core system, many applications are running at the same time among different cores. They all fire off requests of data to the same memory sub-system. This is like rushed hours on a highway, and the traffic is jammed.

*Sharon: What can a compiler do to address this problem?*

Roy: Dealing with memory bandwidth has been traditionally seen as a platform issue and it's beyond compiler's control. However the Open64 compilers take up the challenge. It performs a number of aggressive loop optimizations and data layout optimizations to improve the data locality. We use the data fetched from memory as many times as possible before they are replaced or put frequently used data next to each other so that more data are used within a fetched quantity. This is like carpooling on highway. We don't build more lanes in the highway but we can squeeze more people into a car to reduce the number of cars on the highway.

*Sharon: Interesting, ok so how come these aren't these optimizations widely available in other compilers?*

Roy: In some cases, while these optimizations can greatly reduce memory bandwidth consumption, they could potentially lead to the execution of a larger number of instructions as well. When you run a single application on a multi-core system, where bandwidth is not a bottleneck, such optimizations might

actually slow down a program. This is like taking a step beyond carpooling to take a bus. But bus makes stops and passengers getting in and out of a bus take extra time. In our optimizations, this means we shuffle data around or rearrange them, which could lead to overhead during execution. Most compilers focus on the traditional single-thread performance, and not addressing platform performance issues. Therefore, they don't pay enough attention to this type of optimizations.

*Sharon: How does Open64 avoid these potential slowdown problems, the bus analogy?*

Roy: To avoid possibly slowing down programs during light memory traffic, we group these optimizations under the `-mso` option so that a user can decide whether to apply this option based on the expected execution environment, e.g. how heavy the memory traffic may be. This allows you to get the fast speed of a car when the traffic is light and the benefit of carpooling or sharing a bus to reduce traffic jam when the traffic is heavy. This is a unique feature in the Open64 compilers. We encourage you to try our new `-mso` option in our upcoming release.

*Sharon: Since Open64 is an open source compiler, does AMD get involved in any Open64 related community effort?*

Roy: Yes, in addition to AMD, there are many other companies which deliver compilers based on various forms of Open64 technology. HP, ST-Microelectronics, Nvidia, Pathscale, Absoft to name a few. There are also many university research groups around the world using Open64 for various compiler and computer architecture research. Many of the industrial and university parties have joined an open source community based on [open64.net](http://open64.net).

*Sharon: Wow that excellent, that's a lot of community members. How they all work together?*

Roy: Various Open64 developers exchange technical info and development experiences at a mailing list. There is also a source code repository at [open64.net](http://open64.net) and people can follow steps to download the compilers and source code, and make contributions.

*Sharon: What has AMD contributed to the community?*

Roy: AMD has merged its changes in the Open64 compilers that it released into the open source code repository at [open64.net](http://open64.net). All industrial and academic community members can benefit from that.

*Sharon: How does the community coordinate among members?*

Roy: There is an Open64 Steering Group which I serve on along with 10 other members from industry and academia. This is a policy making body of [open64.net](http://open64.net). We are rolling out policies and processes to allow collaboration, joint developments, and information sharing among community members. Please browse the [open64.net](http://open64.net) web site if you are interested in the community efforts.

### **Michael Lai, Software Engineer, Open64 Project**

Hi, my name is Michael Lai, and I am a software engineer on AMD's Open64 compiler team. I have been working on compilers for many years, mainly in the area of optimizations.

So today I want to talk about some of the optimizations available in the Open64 compiler. The basic function of a compiler is simply to compile a user's source program and eventually link it into an executable file that he or she can then run on a system. To many users this is indeed enough. But there

are also a lot of users who are very keen on how fast their programs run. These performance users require not only that their programs run correctly, but demand that they run fast. That's where compiler optimizations come in. Available in the Open64 Compiler is a large collection of optimizations, ranging from the basic, traditional scalar optimizations, to some very specific ones, such as profile guided optimizations, all the way to many aggressive, state-of-the-art optimizations that only recently appear in publications.

The default optimization level, that is, even if you don't specify any optimization level on the compilation command line, is **O2**. At this level you already get a very solid set of basic, traditional, safe optimizations. They include the familiar scalar optimizations such as constant propagation, where you replace variables in the program by their constant values, if you know them. Or dead code elimination, where you identify all the parts of the program that are simply not needed and so can be deleted. In fact the compiler can even do better than that. There is a class of optimizations called redundancy elimination where the compiler can identify all the parts of the program that are redundant and you can move or delete them. Again, all the optimizations I just mentioned, and many others, occur at the default, or O2, level.

Another set of optimization, called **Loop Nest Optimization**, or **LNO**, occurs at **O3**. O3 turns on LNO, which specializes in analyzing iterative constructs called loops in the program. LNO can be especially effective on scientific and compute intensive programs. These optimizations can even restructure the loops, often to take advantage of the many levels of cache hierarchy available on the processor. Included with LNO are other important optimizations such as vectorization and software prefetching, where the compiler may take advantage of the benefits of some special machine instructions. In the case of vectorization, special SIMD instructions may be generated to replace many equivalent scalar instructions, and in the case of software prefetching, the compiler can use the hardware prefetch instruction to fetch the required data from memory right before it is used by other instructions, eliminating the need to wait for the data, which sometimes can take a long time.

Another class of optimization, called **interprocedural analysis**, or **IPA** for short, can also be very important. IPA is a framework that allows the compiler to get information from many functions across many different files so that it can make some very intelligent optimization decisions. For example, with this interprocedural knowledge, we can perform more effective function inlining, propagate constants more deeply into functions, and do more precise alias analysis. Another aggressive optimization that IPA allows us to do is structure layout optimization. Sometimes, with IPA, the compiler can find out all the uses of say, a particular structure in the program, and may decide to relayout the members of this structure in such a way to achieve much better cache utilization. This kind of optimization requires knowledge beyond just a single function residing in one single file, and this global knowledge is what IPA can provide.

The last optimization I want to talk about is a feature we added just a few months ago, in our 4.2.2 release, and that is the support for huge pages. Traditionally, the size of a page is 4K. But with so many applications nowadays with large data set size, 4K is simply not enough. The next step up is 2M and that's the huge page support we added. This will take full advantage of the increased number of 2M TLB entries the AMD processors have. The compilation flag to turn on **huge pages** is **-HP**, and I would encourage performance users to try it out, especially if your applications deal with large data sets.

There are many, many other optimizations that we won't have time to discuss here: optimizations such as profile guided, feedback directed optimization, or many code generator specific optimizations. In summary, there are a lot of optimizations available in the Open64 Compiler. Detailed descriptions of these optimizations can be found in the compiler user's guide. In addition to that, there are also many knowledge base articles written about some of these optimizations in [developer.amd.com](http://developer.amd.com). Please let us know what you think.

I hope you find this introduction useful. Thank you for watching.