Cray X1 Compiler Challenges (And How We Solved Them)

***Terry Greyzck**, Cray Inc.*

ABSTRACT: *The Cray X1 architecture presents new challenges to producing optimal code, including a decoupled scalar and vector memory architecture, full support for predicated vector execution, hardware support for efficient four-processor shared memory parallelism, and increases in memory latency. This paper describes how these issues were addressed through the development of new compiler optimizations, and describes what is done well and what needs improvement.*

KEYWORDS: Cray, X1, optimization, multistreaming, vectorization, SIMD, MIMD

1. Introduction

1.1. The Cray X1 Architecture

The Cray X1 is the first all-new vector architecture to be developed by Cray since the Cray-2 in 1985. Created based on experience accumulated over 30 years of building vector machines, and experience gained from massively parallel machines such as the Cray T3E, the Cray X1 is a worthy successor to the original vector line.

Designed to provide vector parallelism, tightly coupled shared memory parallelism, and massive globally addressable parallelism, this architecture required substantial advancements in compiler technology to fully exploit its capabilities.

1.2. New Hardware Challenges

Prior to the Cray X1, the most recent Cray vector architecture was the Cray SV1 series of computers. That technology was the last in a line that can be traced back to the original Cray-1, introduced in 1976. While very successful, the Cray SV1 took its design as far as it could go, so it was time for a clean break.

The Cray X1 architecture includes the following new characteristics that make it challenging for compilers:

- New Instruction Set Architecture (ISA)
- Hardware support for 32-bit operations
- The Multistreaming Processor (MSP)
- Decoupled scalar and vector operations
- Predicated vector execution

These changes (and others) from the legacy vector architectures provided the bulk of challenges for developing a successful Cray X1 compiler.

1.3. New Compiler Challenges

For the Cray X1 to be successful, both hardware and software had to be work well from the outset; and the compiler in particular had to produce high-performance code, correctly, immediately, and flawlessly. Therefore, several new approaches to optimization had to be developed.

Since most applications are written in a high-level language, the compiler is relied upon to automatically exploit hardware innovations. The change in architecture for the Cray X1 provided an opportunity to update the user programming model, bringing it in line with models provided by other vendors. This included adding support for:

- 8- and 16-bit integers
- Structure packing and alignment compatible with other major vendors
- IEEE floating point support
- OpenMP for C and C++
- Unified Parallel C (UPC)

These functions are now supported, in addition to ongoing support for Cray language extensions and programming models such as CoArray Fortran and *shmem*.

2. Cray X1 Hardware Differences from the Cray SV1

2.1. New Instruction Set Architecture

The Cray X1 instruction set was designed from scratch and is composed of 32-bit fixed-width instructions. Compared to previous Cray vector architectures, some major changes for the compiler are:

- New instruction set
- Globally addressable memory
- 32 vector registers
- 8 vector mask registers
- 64 A and 64 S registers
- No B, T, or shared registers
- IEEE floating point (last seen on the Cray T90 series as an option, and the Cray T3E)
- Scaled addressing
- Multistream processor support
- Atomic memory operations

2.2. Support for 32-bit Data

Hardware support is provided for 32-bit integer and floating-point operations. By contrast, the Cray SV1 provided limited 32-bit integer support, no 32-bit floating-point support, and almost everything took 64 bits of storage.

Vector registers on the Cray X1 are shared between 32- and 64-bit sizes. Therefore, great care has to be taken to avoid mixing a 32-bit data pattern in a vector register with a 64-bit pattern without an explicit conversion.

2.3. Multistreaming Processor

The Cray X1 Multi-Chip Module (*MCM*) consists of four closely coupled processors, referred to as Single-Streaming Processors (*SSP*). The four processors in the MCM are collectively referred to as a Multistreaming Processor (*MSP*).

Each SSP of an MSP can be controlled through the creative use of the *msync* synchronization instruction. Data is communicated primarily through cache memory.

2.4. Decoupled Architecture

The vector and scalar portions of each SSP are *decoupled*, as if there are two distinct processors: a scalar processor and a vector processor. Scalar and vector instructions can be intermixed, and it is up to the compiler to create the proper synchronization between the two.

The decoupling of the vector and scalar parts of the machine both simplifies the hardware and allows for more aggressive optimization. In previous vector implementations, the hardware provided the necessary synchronization. In order to do so, it had to be extremely conservative, as the hardware could not perform the type of analysis that a compiler could. By moving the burden of the synchronization to the compiler, the number of synchronization points can be reduced.

2.5. Predicated Execution

Vector mask registers control nearly all vector instructions on the Cray X1. For every element where the mask is *true*, the vector operation is performed. For every element where the mask is *false*, the vector operation is skipped. This provides a predicated execution capability for vectors, where the results of a conditional test control the execution of the statements to follow, rather than relying on branching logic to choose between different code paths. The important effect of this is to reduce the need for branching code in vectorized code, which greatly helps performance.

Additionally, the Cray X1 has a *vector length* register that can be used in some contexts for predicated execution. If the vector length register contains a value of zero, vector instructions become no-ops. This capability is used in many contexts, not just conditional code.

2.6. Multi-Level Cache

The presence of a primary and secondary cache is new with the Cray X1. The primary cache is coherent among the four processors of the MSP, and the secondary cache is coherent among the four MSPs of a node. This assists in providing shared memory parallel (*SMP*) models, such as multistreaming and OpenMP.

The hardware contains various mechanisms to control cache behavior, including cache hints for memory references and synchronization primitives.

2.7. Globally Addressable Memory

Cray X1 memory is globally addressable, in that any processor has direct access to the memory of any other processor. Of course the further two processors are from each other, the longer it takes to access memory. However, the hardware is robust to the point where you can actually vectorize memory references with every vector element residing on a different node on the machine (although for performance reasons, it is a bad idea to do this).

3. Levels of Parallelism

The performance of the Cray X1 comes from exploiting parallelism. Some of the parallelism is provided automatically by the hardware, but much depends on the compiler to recognize and use it. The primary levels of parallelism include the following, from lowest to highest level:

1. Multiple registers
2. Multiple functional unit groups
3. Decoupled vectorization
4. Multistreaming
5. Higher-level programming models

The compiler is primarily responsible for finding the parallelism in levels 1 through 4.

4. Compiler Requirements

4.1. Correctness and Stability

It is understood that two required attributes of a production-quality compiler are correctness and stability. In addition to being able to automatically optimize for multiple levels of parallelism, the compiler has to produce *correct* answers and keep on producing them as both the application and compiler evolves.

4.2. Automatic Multistreaming

Multistreaming was first developed for the Cray SV1 series of computers. Although the Cray SV1 had very little hardware support for multistreaming, it provided an excellent development base for the fundamental technologies necessary for multistreaming on the Cray X1.

Multistreaming on the Cray X1 requires the compiler to tie together the processing power of the four SSPs in each MSP. The compiler needs to simultaneously extract the maximum amount of MIMD parallelism from a code, while insuring correct and *repeatable* results.

Experience with automatic parallelism showed that most of our customers required repeatable results. Results that, while correct, vary from run to run are not acceptable. Repeatable results from a multistreamed application were a design goal from the start.

4.3. Improved Automatic Vectorization

The Cray X1 architecture has decoupled vector and scalar execution, predicated execution, and other features that required significant advances to be made in vectorization technology, in order to take full advantage of its performance potential.

4.4. Support for Parallel Programming Models

OpenMP, including OpenMP for C and C++; CoArray Fortran, Unified Parallel C (UPC), *shmem*, and MPI are all programming models that have to be supported. Additionally, a Cray parallel model that takes place at the multistreaming level (CSD) was required.

4.5. Support for Industry-Standard Data Models

The Cray X1 architecture provided an excellent opportunity to switch to more industry-standard data representations. This included adding support for not only 32-bit operations, but also full IEEE floating point support, 8- and 16-bit integer operations, and structure alignment rules that better reflect the expectations of our customers.

One benefit of this was that it became easier to port applications from other platforms to the Cray X1. However, porting incurs some conversion cost for "legacy" codes that have never run on anything but Cray hardware in the past.

5. Base Compiler Technology

5.1. Front Ends

Compiler front ends parse the source code and lower it to an intermediate form, which is then passed on to the common optimizer. Front ends know the minutia about language rules and syntax. Their main task, from an optimization standpoint, is to convert source code to a lower form that can then be optimized.

For the Cray X1, all front ends adopted the Cray X1 ABI. This ABI added 8- and 16-bit integers for Fortran, changed alignment and packing rules, plus other features.

5.1.1. Fortran

The Fortran front end for the Cray X1 was originally developed by Cray Research. As the Fortran standard evolves, the front end is enhanced with new features. Fortran 2003 features are currently being added.

Initially developed for Cray vector machines, the front end required little additional work specific to the Cray X1, beyond adopting the Cray X1 ABI.

Array syntax was formally added to the Fortran language with the Fortran 90 specification. Cray recognized early on that the array syntax mapped cleanly on to the vector architectures and decided to *not* lower most of the array syntax into loops during the front end phase. Instead, the array syntax is passed, unmodified, into the optimization phase. This allows sophisticated transformations to convert the array syntax into the best possible code. In addition, this delayed expansion allows for the vectorization of most array syntax *even with optimization disabled*. This is especially useful for debugging.

Fortran I/O operations are partially lowered by the front end, leaving much of the work to later phases of the compiler. This provides optimization opportunities for I/O that would not otherwise have been possible.

The Fortran front end also provides the language support and initial lowering for three programming models. The final processing of these models is performed in the optimization and code generation stages.

CoArray Fortran (*CAF*) is a language extension that provides a clean method of specifying parallelism. OpenMP is a shared memory parallelism model, and Cray Streaming Directives (*CSD*) is a new model for the Cray X1 that allows for user manipulation of multistreaming parallelism.

5.1.2. C and C++

The C and C++ languages share a common front end source. These languages use a commercial front end that is used by many hardware vendors and is considered one of the best in the industry.

The commercial front end has been adapted to lower its internal representation into the representation used by the common optimizer, and it has been modified to support various language extensions, including Gnu C extensions and Cray-specific pragma implementations.

Additionally, the C and C++ front ends also provide the language support and initial lowering for the OpenMP shared-memory parallel model and the Cray Streaming Directives multistreaming programming model.

Finally, the C front end supports the syntax and provides the initial lowering for the Unified Parallel C (*UPC*) programming model.

5.1.3. Functional Interface

All of the Cray X1 front ends communicate with the optimizer through a *functional interface*, a completely defined set of functions used to communicate program information to the optimizer. This design allows asynchronous development of the various front ends and optimizers, and nearly eliminates the need for expensive joint component integrations.

5.2. Interprocedural Optimization

The interprocedural component of the Cray X1 compilation system is shared by all compilers. It provides support for inlining (including cross-file inlining), cloning, and various optimizations such as tail recursion elimination and Fortran alternate entry point simplification.

This component is new with the Cray X1 system and was created to replace aging language-dependent inlining technology that previously resided in the front ends. Experience with previous inlining implementations was applied to the new component to provide a much more capable product.

5.3. PDGCS Optimizer

The mid-phase optimizer used for the Cray X1 is based on the same technology used for earlier Cray vector and scalar architectures, and is called *PDGCS* (Program Dependence Graph Compilation System). As the name indicates, the control flow is built into a program dependence graph (PDG) rather than traditional basic block structures. This greatly assists in making program transformations.

The development of this component started in the early 1990s, and PDGCS has been continuously updated as new optimizations, technologies, and methodologies have emerged. It provides the core optimization, parallelization, and early lowering capabilities for the compiler.

This component required significant updating for the Cray X1 architecture, but most of the vectorization and some of the multistreaming work was already in place due to the work performed for earlier architectures.

The following summaries are a simplification of the actual optimizer internal processing. The most significant phases are called out, but there are many other optimizations and necessary phases that are not specified here.

1. From the front end interface, a basic block representation of control flow is converted into a Program Dependence Graph (*PDG*) format. This assists the entire optimization process by simplifying transformations, and it helps developers by representing control flow in an intuitive, graphical representation.

2. Early optimizations are performed before the loop analysis and transformation phase. Most of these optimizations manipulate the intermediate text into a form more amenable to later optimizations.

3. Detailed loop nest analysis is then performed. This includes a range of loop analysis and transformation techniques. The results are loop nests that have been modified to remove dependencies, modified for better memory usage, and generally rewritten to improve the overall performance of the nest. The loop nests are marked as vectorization and multistreaming parallelization candidates for later processing.

4. While the intermediate form is still at a high level, the code transformations necessary to provide OpenMP parallelism are performed. This includes creating code for master and slave functions, and all of the associated synchronization logic.

5. The internal representation is then converted to a *linearized* form, where all high-level addressing is lowered to a representation that maps closely to what the target architecture – in this case the Cray X1 – supports. As this is a dramatic lowering step, great care is taken to preserve as much of the original high-level information as possible as an annotation to the newly lowered form. This allows later phases to regenerate much of the dependence information that would otherwise have been lost at this stage, leading to better overall optimization.

6. The multistream translator lowers loop nests marked for multistreaming into their almost-final form. This phase was initially written for the Cray SV1, but was greatly expanded and improved for the Cray X1. More information on this phase is provided later in this paper.

7. The vector translator is the optimization that converts loop nests marked for vectorization into a form that can map almost directly onto the hardware implementation. An original component of the PDGCS optimizer, this has been greatly improved to provide optimal performance for the Cray X1 architecture.

8. The last phase of PDGCS is to perform a great many "traditional" optimizations, such as common subexpression elimination, strength reduction, loop invariant hoisting, and so on. The primary change to this portion of the compiler for the Cray X1 was to modify these optimizations to take advantage of the greater number of vector registers available, compared to older Cray vector implementations.

5.4. Code Generation

With an entirely new instruction set, we decided to develop a new code generator for the Cray X1 and its derivatives. This new code generator uses knowledge gained from earlier implementations and adds support for new optimization technologies developed since the old code generator was written.

The primary lowering phase of the code generator takes the intermediate text produced by PDGCS, which is already at a low level, and completes the transformation to produce Cray X1 instructions. It also performs several optimizations during this process and completes the final lowering of multistreamed functions.

The Cray X1 instructions are then scheduled for the best performance. The scheduling methodology reflects the decoupled nature of vector and scalar architecture. As scalar and vector instructions are typically intermixed, this is a very complicated process.

After scheduling, the instructions are then tied to specific registers. This phase uses new technology for the Cray X1 and handles issues such as register spilling and constant regeneration.

For the Cray X1 we decided to use the Elf and Dwarf object representation methods rather than continuing to use proprietary Cray representations. This allows for better integration with third party applications and allows some use of publicly available libraries for producing the output files.

6. Support for a More Standard User Interface

6.1. Support for 32-bit Operations

Adding support for 32-bit operations created problems that were not immediately apparent. The previous vector architectures were very 64-bit oriented, so some development was necessary to add 32-bit support to the compiler and associated optimizations.

6.2. *Support for 16-bit Operations*

No Cray compiler (even for the Alpha and Sparc) had ever supported 16-bit integers, so this was a challenge. There is no direct hardware support for small data items on the Cray X1, except for an unaligned 64-bit load and byte and half word extraction instructions. This creates complicated instruction sequences for accessing these small data items, and all operations on them need to be performed in software. Significant effort has gone into making these code sequences as fast as possible.

For Fortran, the default compilation mode is to map one and two byte integers onto four byte integers, which is supported by the hardware. There is a compilation switch to disable this conversion, but the majority of codes behave well with the default. For C and C++, there is no such option. The best we can do is discourage the use of *char* and *short* data types in performance-critical areas of an application.

6.3. *IEEE Floating-Point Support*

The compiler already had support for 64-bit IEEE arithmetic from the earlier Cray T90 processor, where IEEE arithmetic was an option. 32-bit IEEE support was new, and full support had to be added. One problem area was that two 32-bit NaNs did not equal one 64-bit NaN, which made it difficult to implement the debug option that initialized all stack variables to the NaN pattern.

7. Vectorization Challenges

7.1. *32 Vector Registers*

After working on architectures that had eight vector registers, the jump to 32 vector registers on the Cray X1 almost seemed like an infinite number to the compiler developer. However, it is still a limited resource, and the registers needed to be managed to avoid oversubscription.

As each vector register has 64 elements, the register set as a whole can be viewed as a 32x64 = 2,048 element cache. Outer loop vectorization was modified to take advantage of the larger register set, using the vector registers as cache, thereby reducing memory bandwidth. In a loop nest, vectorizing an outer loop often leaves the opportunity to hoist memory references that are invariant with respect to the *inner* loop. Such references are loaded as a vector in the outer loop, and then reused throughout the inner loops. This is common with matrix multiplications and similar constructs.

Some other ways of using the larger vector register set were to hoist invariant vector expressions from loops, strength reduce vector expressions where possible, and be more aggressive about recognizing global vector common subexpressions.

7.2. *Predicated Execution*

One major improvement with the Cray X1 is the addition of predicated vector instructions. Earlier Cray vector architectures had one vector mask register, which was used by only one merge instruction. On the Cray X1, there are eight vector mask registers, and nearly all vector instructions use one of them to support predicated execution. As this was new with the Cray X1, significant development was necessary.

The addition of support for multiple vector mask registers was actually straightforward. Using them in the context of predicated execution had significant similarity to the *merge* technique of vectorizing conditionals on earlier Cray platforms, so much of the vectorization logic was adapted for the Cray X1. The resulting code is actually much simpler and easier to read than the older vector implementations.

By software convention, vector mask zero is expected to contain all *true* values. Vector mask zero is typically used for vector execution in code that is not controlled by a vectorized condition.

A major opportunity with true predicated execution is the ability to remove branching logic. On earlier vector platforms, a vectorized condition had to be protected by a scalar test and branch, to protect against the case where the vector condition is never *true*. For most cases on the Cray X1, these tests are no longer necessary, as executing the vector instructions with a mask of all *false* values is effectively the same as executing no-ops. However, significant compiler analysis was required to determine the safety of removing the test; not all conditions can have their test removed.

Another mode of predicated execution on the Cray X1 can be obtained by using zero for the vector length. Vector instructions executed with a zero vector length, regardless of the vector mask, are effectively no-ops. This capability is used to remove scalar branches in search loops and conditions that are vectorized using compression techniques. This is done with safety analysis very similar to what is used for vector masks. This addition of a ‘true’ zero vector length also allows for simpler vector unrolling, and for full unrolling of some loops even when the trip count is not fully known at compilation time.

One significant challenge introduced with vector mask predicated execution is hiding the latency that results from the mask creation. In an ideal world, the vector mask would be calculated and then used for all vector instructions controlled by that condition. Although this works, it is not the fastest solution for most codes. The fast solution is to execute the first several vector loads controlled by the condition as if they appeared in straight line code – with an all-*true* mask. This speculative technique avoids having to wait for the vector mask to be computed before issuing the vector loads and is used heavily by the compiler. This requires significant additional analysis to determine the safety of what amounts to hoisting the loads out of the conditional expression. The compiler actually does a very good job at determining the safety, but the *safe_address* compiler directive was added for those cases that could not be determined safe at compilation time.

7.3. Decoupled Vector and Scalar Operations

The decoupling of vector and scalar operations shifts the burden of correct synchronization from the hardware in the older Cray vector machines to the compiler on the Cray X1. From a compiler standpoint, if everything is done correctly, the resulting code is faster than if hardware was responsible for the synchronization. However, get just a little bit wrong and incorrect answers can result.

Cray decided to err on the side of correct answers. (Forgive us, high performance fanatics.)

The primary challenge here is to add the minimum synchronization necessary between scalar and vector memory references. This is accomplished using the *lsync* instruction. Performed by the code generator, the insertion of synchronization is delayed as long as possible. First, *lsync* instructions are placed at every point where vector and scalar memory references *may* require them. This guarantees functional correctness but is not particularly fast. The code generator then analyzes the generated instruction stream, and removes the *lsync* instructions that are not required for correctness. This compiler analysis uses information such as dependency information, alias analysis, and other tools to minimize the number of scalar/vector synchronizations necessary.

Additional work being performed in this area is aimed at reducing synchronization needs by expanding the region of vectorized code beyond loop nests, and using vector memory instructions with a vector length of one to process what was formerly scalar memory references. This reduces or eliminates the necessity of scalar/vector memory synchronization within that expanded vector region. This *vector region expansion* work is mature for reduction operations and continues for other constructs.

7.4. Longer Memory Latencies

It is a universal problem in high performance computing that the speed of memory access does not improve apace with processor speed. The net effect of this is that new generations of machines must contend with much longer apparent memory latencies. This is also the case with the Cray X1.

For this architecture, the solution is always the same: move the memory operation far enough back in the instruction stream so that, by the time it is needed for a computation, it is ready. This is known as *covering latency*. Many methods are used to accomplish this. Some are direct and some enabling. The hardware supports this by allowing many vector loads to be in progress simultaneously.

The most common direct latency covering optimization is *scheduling*, and the Cray X1 compiler has a new vector scheduler tuned specifically for the architecture. Within the limitations of the code it is presented with, it moves vector loads back as far as it can.

A second direct latency-covering optimization is vector *pipelining*. Although pipelining is common for scalar architectures, pipelining is new for the vector Cray line. Historically, any loop that could be pipelined in a scalar architecture would be pipelined by the *hardware* on a vector platform. Today however, the latencies have grown so large that you sometimes need both hardware (vector) *and* software pipelining to hide them. For many loops, vector pipelining has achieved performance improvements of 30% or more.

There are also indirect optimizations that reorganize the code, making it easier for the direct latency hiding optimizations to do their job. They all work the same way, by increasing the size of the *basic block* – a piece of code that has no branches. Some of these optimizations include the rewriting of conditions into inline *select* operations, the total unrolling (*unwinding*) of loops, and others.

8. Multistreaming

8.1. Automatic Parallelism

The development of automatic multistreaming technology for the Cray X1 was the single largest undertaking for that architecture's compiler. This is no less than automatic shared memory parallelism, in some ways beyond what can be achieved with OpenMP. It had to be fast, correct, and consistent. From a compiler developer's standpoint, the most daunting aspect of this is that automatic multistreaming would be the *default* optimization level on the Cray X1.

This was a long development project. Early multistreaming was first introduced for the Cray SV1 line, as much to develop the technology as to provide higher performance. For the Cray SV1, multistreaming was only available only via an option and was not enabled by default.

Cray X1 multistreaming started where Cray SV1 multistreaming left off. In some ways, it was easier on the Cray X1, as there were hardware primitives to assist in synchronization, and the caches appear coherent to each of the processors. However, the expectations were much higher for the Cray X1 than they ever were for the Cray SV1.

8.2. Multistreaming Constructs

At the heart of multistreaming is the selection of the constructs that will run in parallel. Like vectorization, the primary focus is on loop nests. Most of the same transformations that are performed for vectorization also apply to multistreaming.

Because of the fast hardware synchronization on the Cray X1, a larger variety of loops can be profitably parallelized than with most other architectures. These include *update* loops, conditional last value captures, and others. Additionally, fast synchronization allows multistreamed reductions to return the same answer – every time. This repeatability in some ways makes the multistreamed processor with four two-pipe vector processors appear to work like an eight-pipe vector processor. While not a true eight-pipe vector processor, it still achieves automatic multi-processor parallelism that is as reliable and repeatable as vectorization.

8.3. Minimizing Synchronization Overhead

While the Cray X1 has additional hardware support for multistreaming processor synchronization, the costs associated with it are still significant and need to be minimized. The primary method employed by the compiler is to expand the region of multistreaming beyond individual loops and loop nests, running as much code as possible on all four SSPs of the MSP.

This *streamed region expansion* is accomplished by running serial code redundantly on each SSP. This allows each SSP to fetch its own data and make its own calculations without waiting for an individual processor to complete and broadcast its results.

During the phase of compilation that determines which loops to vectorize and which to multistream, the compiler determines how many of the non-parallel loops in the nest can be run redundantly and what lightweight synchronization, if any, is required.

A later optimization takes this further and attempts to run the purely non-parallel sections of code redundantly to avoid synchronization costs. Because of these efforts, entire subroutines occasionally can be run fully multistreamed with no synchronization, except at the entry and exit points.

Other methods of reducing synchronization overhead are also used, including two independent execution streams, one for processor zero and one for processors one, two, and three. Other methods of synchronization are also available, and only the least costly method necessary for correctness is used.

8.4. Minimizing Negative Optimization Effects

With earlier automatic parallelization implementations, the addition of synchronization barriers and scoping boundaries had severe negative effects on other optimizations. The effect of automatically parallelizing a loop nest could degrade the performance of other parts of the same function, primarily by preventing the code motion necessary for general purpose optimization.

Given a clean slate for the multistreaming implementation, the solution to this challenge for the Cray X1 was to introduce weak markers to indicate the boundaries of the multistreamed area. These weak markers contain information on what absolutely could not be moved across them (generally a very small list), allowing all other data to be free to move across these barriers.

This allows for optimization in multistreamed code to be nearly as good as in single processor code. However, it is somewhat controversial in that the free code motion can actually result in an inconsistent program during some phases of optimization by violating data scoping rules. A late phase corrects any data scoping problems introduced by optimization. Overall, this has been a very successful implementation.

9. Balancing SIMD and MIMD Parallelism

9.1. Multiple Levels of Parallelism

One of the most challenging and labor-intensive aspects of optimizing for the Cray X1 series is to balance vector parallelism with multistream parallelism. While vectorization can achieve greater than an order of magnitude performance improvement, multistreaming is limited by the number of processors and can achieve a maximum speedup of four. Clearly, this means that it is more important to pursue vector parallelism than multistreaming. However, since we have both, the compiler attempts to use *all* of the resources of the machine for maximum performance. This means that loop nests are both vectorized *and* multistreamed where possible.

9.2. Inner Vector, Outer Multistreamed

The preferred use of multiple levels of parallelism is to have a bop nest that consists of two or more loops, vectorize one of the inner loops, and multistream one of the outer loops. This method exploits multi-level parallelism, and it works very well, when applicable. However, it is not a universal solution as there may be only one loop, or the trip count of the inner loop may be small. In many of these cases, loop transformations such as loop interchange can be beneficial, but sometimes other methods must be considered.

9.3. Inner Multistreamed, Outer Vector

The Cray X1 compiler has the ability to run a vectorized outer loop redundantly, containing an inner multistreamed, partitioned loop nest. This is useful when the 32 vector registers of each processor of the Cray X1 can be used as an extremely fast cache: the vector registers are loaded redundantly in the outer loop and then reused many times in the inner, partitioned loops. This can greatly reduce bandwidth requirements and improve latency issues. One example of where this is effective is for common *dgemm* matrix multiplications, where this technique allows the code to achieve 99% of peak performance. The compiler uses this technique, when profitable.

9.4. Small Trip Counts

If a loop has a very small number of trips, say 100 or fewer, it is clearly better to simply vectorize it rather than multistream *and* vectorize it. The compiler uses range analysis to generate estimated and guaranteed maximum ranges for the loop trip count and incorporates that knowledge into its parallelization decision making process.

10. Giant Applications!

The applications that are run on Cray computers have evolved over the years from fairly small, home-grown programs, to applications that include millions of lines of code – and are getting larger all the time. These applications require a compiler that both conforms to language standards for ease of porting, and is exceptionally reliable. In addition, top performance is a requirement for the Cray X1. Any one of these requirements is difficult to achieve, but attaining language conformance, absolute functional correctness, and extreme performance is a challenge.

10.1. 10,000 Vector Loops in One Subroutine

One case in point is an application that ran into a compiler limit of 10,000 vector loops in *one subroutine*. That limit of 10,000 was considered essentially infinite from a compiler standpoint, but a real application managed to exceed it without much effort.

Ten thousand vector loops in one subroutine is a *lot*. The limit has been extended to 2,000,000,000. We hope that that limit will hold us for a while.

10.2. 1,000,000 Unique Expressions

A recent application managed to create over one million unique one-operand expressions in a single subroutine. This is another limit the compiler never expected to see.

10.3. Exponentially Increasing Application Size

An emerging trend in application development is the automatic generation of high-level Fortran code – quite a challenge for a compiler that was designed for hand-written code in mind. This can lead to an expansion in the size of applications that cannot be achieved by hand coding alone. One popular application has a growth curve, in lines of automatically generated Fortran, which nearly doubles in size every six months. That will test the limits of any compiler.

11. Conclusion

The production of a compiler that is robust, reliable, stable, and creates the fastest possible code is a challenge for any architecture. For the Cray X1, new hardware made this even more difficult. Additionally, trends in increasing application size and complexity also create their own problems.

The Cray X1 series of compilers recognizes these challenges, and strives to meet all of them. Developments in the academic world, and innovative, Cray-developed solutions were necessary to meet the requirements of the

Cray X1. Continuing development further solidifies the abilities of the compilation environment.

Acknowledgements

Sincere thanks to all who reviewed this document, and who provided information for it. Thank you to Cray for allowing my colleagues and me to work for twenty years on what continue to be the fastest and most innovative computing systems in the world. There is always another challenge to conquer.

Reference Materials

The following Cray reference manuals are useful for obtaining further information:

- S-2315-54: Optimizing Applications on Cray X1 Series Systems
- S-3901-54: Cray Fortran Compiler Commands and Directives Reference Manual
- S-2179-54: Cray C and C++ Reference Manual

In addition, the following links are useful:

www.co-array.org	CoArray Fortran
www.openmp.org	OpenMP
upc.gwu.edu	Unified Parallel C

About the Author

Terry Greyzck is a graduate of Michigan State University, and has worked for Cray Inc. for 20 years. He started by providing field support at the Livermore Department of Energy laboratories, and eventually migrated to compiler optimization work, with an occasional management stint thrown in for good measure.

Terry's first vector translator was written for the Cray Ada compiler. From there, the next step was to develop vector translation, multistreaming technology, and late-stage optimizations for the current Cray X1 common optimizer, which is used for Fortran, C, and C++.

Terry currently works out of the Cray Inc. office in Mendota Heights, Minnesota. He can be reached at:

Terry Greyzck, Cray Inc.
1340 Mendota Heights Road
Mendota Heights, MN 55120
(651) 605-8979
tdg@cray.com.