Using the Gnu C++ Compiler to Generate Code for a Massively Parallel SIMD Processor

James H. Burrill

May, 1993

This work was supported in part by DARPA Project#DAAL02-91-C-0051.

DRAFT

1 Introduction

The Gnu C++ compiler is designed to be easily ported to many different computers[**?**]. The compiler can be used both to generate code for CISC architectures, such as the Intel 386 and DEC VAX, and for RISC architectures such as the MIPS R3000 and the SPARC$^{\mathrm{TM}}$. These architectures all share some common attributes. They are all byte addressable and use 32-bit words. They have one other attribute that is not mentioned in any Gnu documentation. All of these computer architectures are *single instruction, single data* (SISD). In contrast, this project required that code be generated for a *single instruction, multiple data* (SIMD) computer, called the CAAPP (Content Addressable Array Parallel Processor)[**?**], that is part of the Image Understanding Architecture (IUA)[**?**].

The CAAPP is composed of special processor chips and a large amount of off-chip memory. Each processor chip contains 256 processing elements. Each processing element has a small amount of on-chip memory and a much larger amount of off-chip memory. Data transfers on and off the CAAPP chips take significantly longer than data transfers within the chips. Consequently, it is important to maximize the residency of data in the on-chip memory. This situation is the same as that for any standard processor such as the SPARC. To increase the resident time, standard processors are designed with registers and compilers for them are created to generate code that efficiently uses those registers.

The IUA project had already developed a set of C++ classes that embodied the concepts inherent in using a massively parallel, SIMD processor for image understanding applications[**?**][**?**]. The accompanying method library generates the necessary instructions and sends them to the CAAPP. This set of classes is referred to as the IUA Class Library. It defines objects (called *planes*) that are analogous to C++ scalars of type `int, char`, etc. Unlike the C++ scalar types, planes hold thousands of values — one for each processing element of the SIMD computer.

Along with the definitions of the object classes are the definitions of the associated class methods. The run-time library, implementing the class methods, uses run-time decisions to allocate the CAAPP registers. While this approach produces usable object code, the overall performance is suboptimal because of the enormous overhead of generating the CAAPP instructions at run time. Thus, it is important to utilize all of the optimization methods employed by the Gnu compiler to generate efficient code at compilation time. In particular, the Gnu compiler should allocate registers for the CAAPP at compile time.

The CAAPP is composed of *thousands* of simple bit-serial processors, each with its own data memory, arranged in a rectangular grid. None of the individual processors in the CAAPP have traditional control logic. The instructions are generated by a controller called the Array Control Unit (ACU) and broadcast to all of the SIMD processors of the CAAPP. This controller is composed of a standard processor (called the Control Processor) and an attached coprocessor. The coprocessor is a microprogrammed controller (called the Micro Controller) created for this purpose.

The user's program runs on the Control Processor (a SPARC) and sends commands to the

Micro Controller which generates the bit-serial instructions and sends them to the CAAPP. For example, the CP might command the Micro Controller to issue instructions to the CAAPP processing elements to add two values of type `char`. The Micro Controller would convert the command into a sequence of 17 bit-serial instructions that accomplish this task, and send these instructions to the CAAPP.

The interface between the Control Processor and the Micro Controller is implemented using memory mapped control. The CP writes a word to a location in its memory space. The Micro Controller accesses the word and interprets it. How the words are interpreted is determined by the microcode for the Micro Controller.

The microcode implements an *abstract machine*[**?**] that understands common operations such as add, subtract, and move. Control of the program flow remains with the Control Processor. The abstract machine was designed to provide as convenient a target as possible for the Gnu compiler. The abstract machine has registers that can be allocated and is an implementation of a normal RISC model with load/store and register-register operations. These registers are implemented with the CAAPP on-chip memory. Like most RISC machines, transfers between registers and memory are much more expensive than transfers between registers.

Despite the simplification provided by the abstract machine, many complex problems remained in order to use the Gnu compiler with a parallel processor. This paper explains what these problems are and how the compiler has been changed to solve them. The problems encountered are introduced here and described in detail in later sections of this paper.

- Parsing the Program

  The user's program is written with a set of C++ *classes* that define the operations that may be performed on the CAAPP. This approach allows us to provide a programming structure that can be used without requiring that CAAPP hardware be available; programs may be written, compiled, and executed on a normal workstation[1]. The classes, called the IUA Class Library, define normal operations, on instances of these classes, such as add and subtract. Other operations that facilitate image processing operations are also defined. All of these definitions are in a header file.

- Bit-serial CAAPP ALU

  The C++ language assumes that the target computer has an ALU that is large enough to operate on values of type `int` (usually 32-bits). A C++ compiler promotes all integer values to this size before doing an arithmetic operation. Adding two eight bit values on the CAAPP requires 17 instructions. Adding two 32-bit values requires 65 CAAPP instructions.

---

[1] A program, written with the IUA Class Library and compiled with a standard C++ compiler, may be linked with either of two libraries. One library implements the methods using sequential operations and runs on a standard workstation. The second library implements the methods by using routines that send abstract machine instructions to the CAAPP.

If all integer values are first converted to 32-bits, peak efficiency will not be achieved. This difference is significantly greater when doing bit-serial multiplies and divides.

- Dual-instruction Flow

  Abstract machine instructions for controlling the CAAPP must be sent to the Micro Controller from the CP. Therefore, the compiler must generate both the instructions for the abstract machine and the CP instructions to send those abstract machine instructions to the Micro Controller.

- Activity

  As each CAAPP processor does the same operation at the same time only using different data, branching can not be used for data dependent algorithms. An individual CAAPP processor *cannot* execute a branch. Instead, an individual processor may be *active* or *inactive*. Where a sequential processor uses *if-then-else* control, the CAAPP executes both the *then* and the *else* clauses. A subset of the CAAPP processing elements is active for the *then* clause and a different subset is active for the *else* clause.

  The IUA Class Library changes the semantics of the C++ assignment operation on planes to utilize the concept of activity. The assignment operation is accomplished only at active CAAPP processors. The semantics of this new assignment is closer to the traditional *merge* operation where a value is selected from either one operand (the destination) or the other (the source) depending on the activity (a third operand).

```
procedure assignment (destination, source, active)
  for each processor i
    if active[i] then
      destination[i] = source[i]
```

- Virtualization & Compatibility

  An additional attribute of *shape* must be considered for variables along with the attributes of *type* and *value*. Typical images used in image understanding are at least 256 × 256 pixels in size. The largest CAAPP currently has 128 × 128 processors and many smaller configurations are possible. Users should be able to write their programs without assuming any particular size for the CAAPP. In addition, an object module created for one size of CAAPP should be usable for any other size of CAAPP, and different image sizes should be usable in the same program.

  The compiler and related software are responsible for mapping the user's program to a particular CAAPP. The result is that a larger plane, representing an image, must be partitioned into a set of tiles and the operations must be repeated for each tile. The compiler also must detect changes in the shape or number of tiles.

When two planes are used in the same operation, they must be compatible. That is, they must have the same shape — the same number of rows and columns. When they do not conform, an error must be signaled.

Since it is important to reduce the CP overhead, as many decisions as possible should be made at compile time. When a compile time decision is not possible, the compiler must generate code to make these decisions at run time.

Sections **??** through **??** discuss each of these problems in more detail and provide a description of the problem solution. Section **??** gives a quick overview of how the Gnu C++ compiler works.

2 Phases of the Gnu C++ Compiler

This section briefly describes the structure and the phases of the Gnu C++ compiler. This information will be referred to in the other sections of this paper.

The compiler uses two major representations of the user's program. The first is a tree structure that is used to represent each expression in the user's program. This tree structure is language dependent.

The second representation is a list of instructions. These instructions are language independent. They are also independent of any particular computer. Each instruction is represented using a *lisp-like* expression called an `insn`. The expressions are written using the Register Transfer Language or RTL. The RTL specifies transfers and operations between registers of a pseudo machine with an unlimited number of registers. There are a set of standard operations that the compiler understands. The RTLreferences registers that may or may correspond to registers in the target computer. Registers that do not correspond are called "pseudo" registers and have register numbers that are greater than the number of actual registers in the target computer. Each RTL expression has an associated *mode*, called the *machine mode*, which represents the type of value of the expression.

The compiler performs its work in a number of "passes":

Parsing: Convert each expression and statement into a tree representation. Convert each tree into a machine independent list of instructions. The result is a list of instructions (`insns`) for each function in the user program.

Jump Optimization: Simplify the jump structure to reduce the number of jump `insns` and to detect and eliminate "dead" code.

Register Scan: Scan the `insn` lists and classify the use of each register for use in later passes.

Common Subexpression Elimination: Eliminate duplicate sequences of `insns`.

Loop Optimization: Move loop invariant `insn` sequences from inside loops.

Data Flow: Divide the `insn` list into blocks based upon branch points and analyze the register usage in each block.

Combine: Attempt to combine two or more `insn`s into a single instruction using substitution and algebraic simplification.

Scheduling: Reorder `insn`s to overlap memory data transfers and functional units.

Local Register Allocation: Assign the pseudo registers used only in a basic block to actual hardware registers of the target computer.

Global Register Allocation: Assign the pseudo registers used in more than one basic block to actual hardware registers of the target computer.

Reload: Modify register assignments previously made to provide temporary registers for use for pseudo registers not previously assigned.

Assembly Language Output: Convert the `insn` list into an assembly language program for the target machine.

This list is simplified for this paper. In general, a pass is logic that scans the `insn` list of a function. The logic of some passes is very simple.

Parsing

As the user's program is being parsed, the compiler builds a language dependent, tree representation of the program. In this tree, all identifiers have been converted to pointers to tree nodes and all constants have been converted to a binary representation.

Each node of the tree represents a construct such as a binary operation (i.e. add), an identifier, a control structure, or some other structure that the compiler understands.

Each node has a `type` which is a pointer to the structure that defines the type. Many of these type structures are predefined and represent standard C++ types such as `char` and `int`. The node type represents the type of the *value* obtained by evaluating the tree at that point.

If the node is not a simple construct, it will be represented as a sub-tree rather than a leaf. C++ classes are represented as a complex structure whose parts have their own types. If the class is a derived class, it will have a *basetype* that is the type of the base class.

Figure **??** shows the tree representation of the expression `z = x + y` where `x, y,` and `z` are defined by

```
char  x;
int   y;
short z;
```

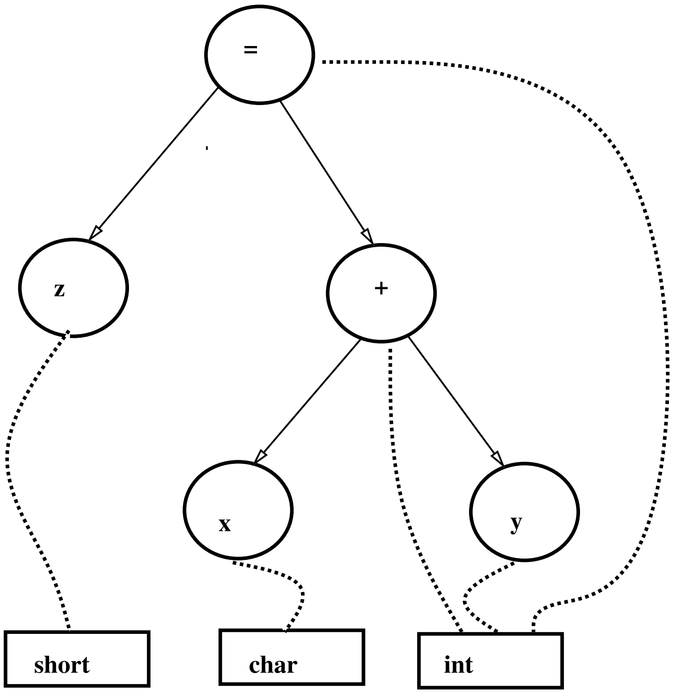


Figure 1: Tree Representation of C++ Expression z = x + y.

Instruction Generation

```
(insn 11 8 12
  (set (reg:QI 169)
       (mem:QI (plus:SI (reg:SI 165)
               (const_int -1)))))
(insn 12 11 13
  (set (reg:SI 170)
       (ashift:SI (subreg:SI (reg:QI 169) 0)
                  (const_int 24))))
(insn 13 12 15
  (set (subreg:SI (reg:HI 168) 0)
       (ashiftrt:SI (reg:SI 170)
                    (const_int 24))))
(insn 15 13 16
  (set (reg:HI 171)
       (mem:HI (plus:SI (reg:SI 165)
                        (const_int -6)))))
(insn 16 15 18
  (set (reg:SI 172)
       (plus:SI (subreg:SI (reg:HI 168) 0)
                (subreg:SI (reg:HI 171) 0))))
(insn 18 16 20
  (set (mem:HI (plus:SI (reg:SI 165)
                        (const_int -10)))
       (subreg:HI (reg:SI 172) 0)))
```

Figure 2: `insns` representing the C++ expression `z = x + y`.

The tree structures are converted into `insns`. The C++ expression `z = x + y` shown in Figure **??** is converted to the RTL shown in Figure **??**. In this figure there are six `insns`. Each `insn` has a number – the number of the first `insn` in the figure is 11. Each `insn` points to the `insn` preceding it and following it in the list of `insns` for a function. Each `insn` contains one RTL expression. The RTL in `insn` 11 from Figure **??** represents the loading of a byte into pseudo register 169 from the memory location addressed using the value from pseudo register 165 and the offset -1. The QI, HI, and SI designations represent the *machine modes* of quarter-word, half-word, and word.

`insn` 12 represents an arithmetic left shift of pseudo register 169 by 24. The `subreg` converts the mode from quarter-word to word. `insn` 13 shifts right by 24. The net result is to sign-extend a byte to a word although the result is used as a half-word. `insn` 16 adds two half-words (using 32-bit addition) and `insn` 18 stores the result as a half-word.

The compiler represents the target computer with RTL also. Every instruction of the target computer, that the compiler uses, is represented by one RTL pattern. These patterns are converted to several different forms by various tools that are used to build the Gnu C++ compiler.

When a target machine is defined, part of the definition consists of *named* procedures for accomplishing standard operations that the compiler understands. These procedures generate `insn`s for instructions that are more specific to the target computer. At various points in the compilation process these `insn`s are matched against patterns for the actual instructions of the target computer.

Optimization

Optimization is performed for each function of the user's program and operates on the `insn`s for the entire function. The compiler changes the `insn`s to optimize the function. The optimizations include common sub-expression elimination, jump optimization, instruction scheduling for functional units, and loop optimization. `insn`s may be reordered by optimization. If an `insn` is not dependent on a preceding `insn` *in a way understood by the compiler*, it may be moved such that the order of the two `insn`s *is reversed.*

Register Allocation

The compiler attempts to assign registers of the pseudo-machine to the hardware registers of the target computer after most optimizations have been performed. There are various phases and modules of the compiler that do this. The particular phases invoked depend on the type of optimizations that the user requested and determine how much effort the compiler uses to avoid transfers between registers and memory. The result is still RTL but most registers are now assigned to physical registers of the target machine.

The hardware register definitions are included in the definition of the target computer. This definition specifies not only the number of registers in the target computer but also what restrictions may exist on the use of each register (e.g., some registers may not be loadable directly from memory or may not contain floating point values). These restrictions are specified in different ways. One way is that the allowable machine-modes are specified for each hardware register.

The compiler distinguishes between registers of the pseudo-machine and registers of the target computer by the register number. If a target computer has n registers, then register numbers $\geq n$ are pseudo-registers. For example, the SPARC has 32 general purpose registers and 32 floating point registers. Figure **??** shows the result of register allocation on a SPARC for the RTL shown in Figure **??**.

```
(insn 11 8 12
  (set (reg:QI 8)
       (mem:QI (plus:SI (reg:SI 30)
                        (const_int -17)))))
(insn 12 11 13
  (set (reg:SI 9)
       (ashift:SI (subreg:SI (reg:QI 8) 0)
                  (const_int 24))))
(insn 13 12 15
  (set (subreg:SI (reg:HI 8) 0)
       (ashiftrt:SI (reg:SI 9)
                    (const_int 24))))
(insn 15 13 16
  (set (reg:HI 9)
       (mem:HI (plus:SI (reg:SI 30)
                        (const_int -22)))))
(insn 16 15 18
  (set (reg:SI 8)
       (plus:SI (subreg:SI (reg:HI 8) 0)
                (subreg:SI (reg:HI 9) 0))))
(insn 18 16 20
  (set (mem:HI (plus:SI (reg:SI 30)
                        (const_int -26)))
       (subreg:HI (reg:SI 8) 0))
```

Figure 3: RTL for z = x + y after register allocation.

Reload

When assigning the registers of the pseudo-machine to the hardware registers of the target computer, the compiler sometimes fails to assign all of the pseudo registers. The *reload* phase, which may be run more than once, assigns stack slots for these registers, and generates `insn`s that move values between the slots and the hardware registers. As a result, some pseudo registers, that had been assigned a hardware register before, are reassigned to stack slots. The compiler assumes that the target machine is stack-based and that transfers, between the stack and a hardware register, generally do not affect any other registers [2].

This reload phase causes two problems. First, due to the limited number of abstract machine registers available[3], any reloading tends to generate other reloading resulting in significant loss of run-time efficiency. Second, reloading generates new `insns` *after* all the other `insn`s have been generated. `insn`s for the abstract machine require additional processing beyond what is required for SPARC `insns`.

Assembly Language Generation

After all the optimizations, register assignments, reloading, etc have been accomplished, the assembly language code for each `insn` is generated. An assembly language statement is given as part of each instruction pattern definition. Each pattern is meant to represent an actual instruction of the target machine. For example, the following is the pattern for the SPARC instruction that adds two full word integers.

```
(define_insn
  [(set (match_operand:SI 0 "register_operand")
        (plus:SI (match_operand:SI 1 "register_operand")
                 (match_operand:SI 2 "register_operand")))]
  "add %1,%2,%0")
```

An actual assembly language statement may be supplied as shown. Or, C code to generate multiple assembly language statements may be supplied instead. This is shown below for the pattern that generates a function return for a SPARC.

```
(define_insn
  [(return)]
  "* return output_return (operands);")
```

---

[2] For the SPARC, the description of its registers does not allow the frame pointer register to be used for any other purpose. As long as the stack offset is small, no additional registers are used. This eliminates most possible conflicts with other, already assigned, registers.

[3] The number of registers is limited by the amount of CAAPP on-chip memory.

Note that these patterns do not look exactly like the `insn`s shown before in Figure **??**. Instead of specifying registers, these patterns specify how to match one of the `insn`s generated previously. For example, the `insn`

```
(insn 16 15 18
  (set (reg:SI 8)
       (plus:SI (reg:SI 8)
                (reg:SI 9))))
```

matches the pattern for addition shown above and instructs the compiler to generate the assembly statement

```
  add 8,9,8
```

The actual matching of `insns` to instruction patterns may occur in many of the phases of the compiler after the `insns` has been generated. Each `insn` is changed to include a pointer to the pattern it matched. During the very last phase of the compiler, the list of `insns` is scanned in sequence. The assembly language is generated using information from the instruction pattern (i.e. assembly language statement(s)) and from the matching `insn` (e.g. register numbers). If no match has been found for an `insn`, the compiler *aborts* [4].

3 Parsing the Program

The compiler generates the tree shown in Figure **??** for the C++ expression `z = x + y` when `z, x, & y` are standard scalar types. When the variables `z, x,` and `y` are defined to be instances of C++ classes shown below, the unmodified compiler generates the tree shown in Figure **??**.

```
CharPlane x;
IntPlane y;
ShortPlane z;
```

The goal is to generate the same type of tree for planes that the compiler generates for C++ scalars. This tree structure does not result in `insns` that can be used to allocate registers for the abstract machine or to perform any optimizations. This tree represents the expression using structures that do not allow the variables `x, y,` and `z` to be mapped to registers. Instead of generating an instruction to add `x` and `y`, the compiler generates a method call whose arguments are pointers to structures representing the instances of the classes defined by the IUA Class Library. Instead of generating an instruction to move the result of the addition from a register to a memory location, the compiler again generates a method call.

---

[4] This is an indication that the compiler was ported incorrectly to the target computer.

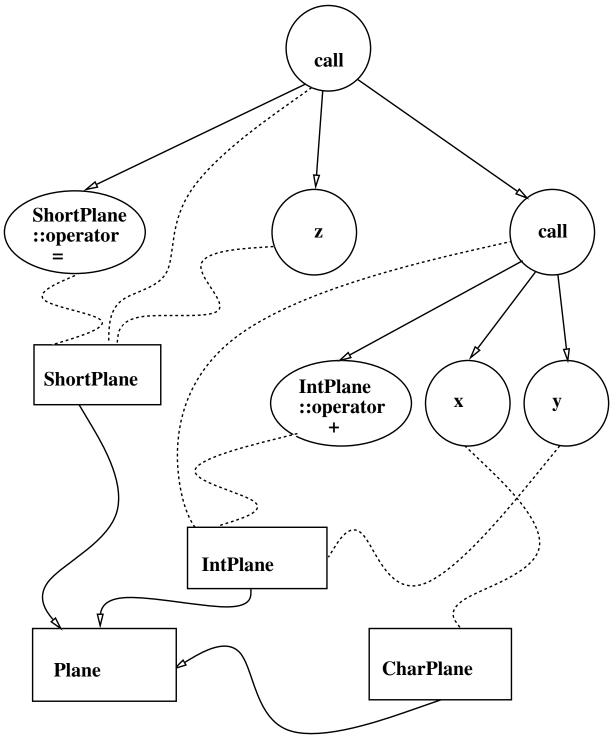


Figure 4: Tree Representation of IUA Class Library Expression

mode	size	class	for type
Scalar Modes			
QI	8 bits	Integer	char
HI	16 bits	Integer	short
SI	32 bits	Integer	int
SF	32 bits	Float	float
Plane Modes			
BP	1 bit	Integer	BitPlane
CP	8 bits	Integer	CharPlane
DP	8 bits	Integer	UCharPlane
SP	16 bits	Integer	ShortPlane
TP	16 bits	Integer	UShortPlane
IP	32 bits	Integer	IntPlane
FP	32 bits	Float	FloatPlane

Figure 5: Machine Modes

When the parser recognizes that a reference is made to a plane in an expression, it replaces that reference with a reference to a new type (internal plane) that does not generate method calls. This new type is detected at those places in the compiler at which special processing is required. The new types are treated in the same manner at every other place as the original C++ scalar types.

The new types are implemented by defining new machine-modes and adding registers in to the target machine definition that are used only for these modes. Thus, the compiler allocates these registers using the normal register allocation and optimization logic.

The modified compiler uses seven new *machine modes* that correspond to the seven types of planes defined by the IUA Class Library[5]. Each mode, new and existing, has an additional attribute of either *scalar* or *plane*. A scalar is just one value. A plane has many values, one for each processor in the SIMD grid. Figure **??** shows the new machine modes along with some of the original modes. The description of the target computer specifies which hardware registers may contain values with which modes. The compiler guarantees that it will not place a value in a register if the mode is not allowed.

The compiler generally processes planes as if they were scalars. However, there are places in the compiler where there is special code to process planes.

It is necessary to use some of the IUA Class Library method definitions in the standard way. The parser recognizes instances of the IUA classes specially for some things and uses the normal C++ language facilities to handle the rest. Consequently, the compiler converts IUA Class Library

---

[5] The reason for using so many new modes is due to the bit-serial nature of the ALU used in the CAAPP processors, as explained in the next section.

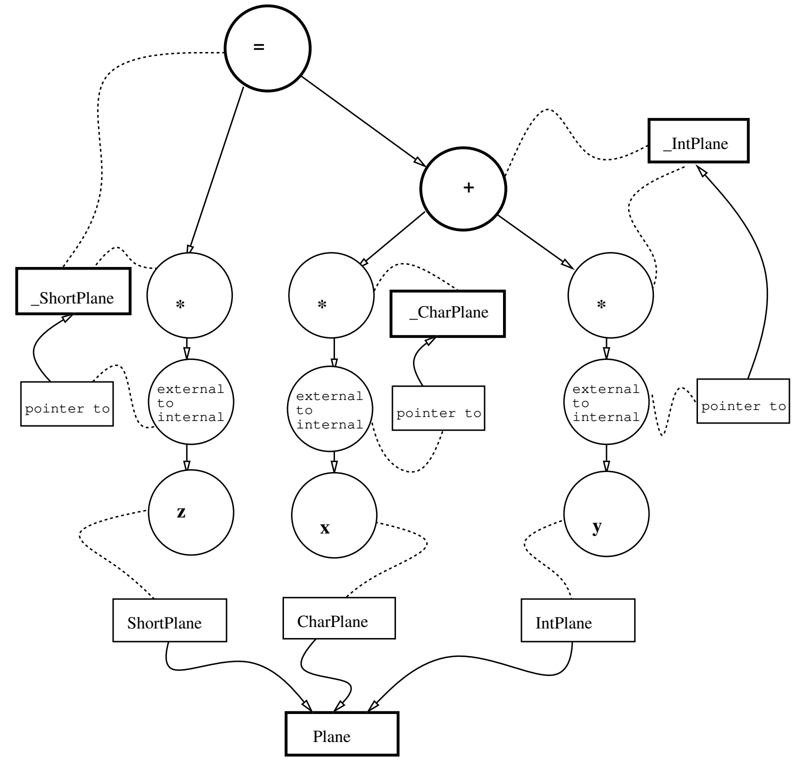


Figure 6: Tree Representation of IUA Class Library Expression

plane instances to internal planes at some places and converts internal planes to IUA Class Library plane instances at other places.

The references to IUA Class Library class instances are converted to the new type by creating a tree structure that causes the proper code to be generated for this conversion. The result of this is shown in Figure **??**. While this may look more complicated, note that the top of the tree has the same structure as the top of the tree in Figure **??**. The compiler has the ability to generate `insn`s for this structure using the registers defined for the abstract machine.

`insn` 98 in Figure **??** obtains the address of the plane. The memory referenced using the plane machine-modes is not in the Control Processor's address space. It is in CAAPP off-chip memory. The address is kept in a field of the structure that represents the instance of the plane. The reference operation generates an instruction to load a plane register using an address loaded into a *scalar* register. `insn` 99 loads a `CharPlane` into pseudo register 103 which has the proper mode (CP). `insn` 101 loads an `IntPlane` into pseudo register 104. `insn` 102 performs the addition operation. `insn` 104 truncates the `IntPlane` to a `ShortPlane` and `insn` 105 stores the result into

```
(insn 98 97 99
  (set (reg:SI 99)
       (plus:SI (reg:SI 81)
                (const_int -16))))
(insn 99 98 101
  (set (reg:CP 103)
       (mem:CP (reg:HI 99))))
(insn 101 99 102
  (set (reg:IP 104)
       (mem:IP (reg:HI 101))))
(insn 102 101 104
  (set (reg:IP 102)
       (plus:IP (reg:CP 103)
                (reg:IP 104))))
(insn 104 102 105
  (set (reg:SP 105)
       (truncate:SP (reg:IP 102) 0)))
(insn 105 104 106
  (set (mem:SP (reg:HI 88))
       (reg:SP 105)))
```

Figure 7: RTL for z = x + y using new modes

CAAPP off-chip memory.

Note the machine modes specified in the instruction patterns. The pseudo registers are assigned to specific hardware registers based upon these modes. The `insn` numbered 99 above will match the instruction pattern below which is provided in the description of the target machine.

```
(define_insn
  [(set (match_operand:CP 0 "register_operand")
        (match_operand:CP 1 "iua_address")))]
"* return iua_gen_load (operands, 0x0210);")
```

The `match_operand` operation, for operand 0 of the RTL operation in the `insn` informs the compiler to try to match something of mode CP that is a `register_operand`. The `match_operand` operation for operand 1 informs the compiler to try to match something of mode CP that is an `iua_address`. `Register_operand` is a boolean function that is supplied with the basic compiler and `iua_address` is a boolean function that is defined as part of the definition of the target computer. These functions are called with the appropriate part of the `insn` RTL as an argument.

The `iua_gen_load` function generates the SPARC assembly language statements that cause the CAAPP operation to be performed. This is explained in more detailed in the section on Dual-instruction Flow.

Originally, in seemed possible to modify the code that generates the tree structure for the method calls to generate the tree structures for an add instead. Logic to detect calls to the methods for operators of the IUA Class Library placed in the compiler routine responsible for generating a tree representation of a call to a class method, would be able to generate some other, more useful tree representation. These modifications would generate the tree representation for an add of two IUA Class Library planes instead of a call to the add method. To solve the problem this way required too much information that is already in an inconvenient form at this point in the compilation and does not use the same compiler routines as is used for scalar types to generate the tree generation logic for arithmetic expressions.

4 Bit-serial CAAPP ALU

C++ assumes that all integer arithmetic operations are performed using values of type `int`. Arguments to an arithmetic operation that are not of type `int` are converted/promoted to type `int` before the operation is performed. All of the computers targeted by the Gnu compiler have 32-bit or larger ALUs. This is generally true for modern SISD computers because the marginal cost of an 32-bit ALU, that can do operations in parallel by bit, is small and provides a significant speed advantage.

With a SIMD computer the cost/performance equation is different. Performance is achieved through having many processors working in parallel. Performance is related more to the number of processors that can be effectively packaged together. The size of a 32-bit ALU, when multiplied by a large number of processors, may limit the total number of processors that can be packaged together [6]. For these reasons, massively parallel SIMD computers generally have smaller ALUs (i.e., a narrower data path).

Peak efficiency on the CAAPP is achieved when it is possible to use only as many bits as are needed to represent a value because the CAAPP has a one-bit ALU. If a 7-bit value is multiplied by a 5-bit value, this requires approximately 35 bit-serial operations. If both the 7-bit value and the 5-bit value are represented using 8-bits, the multiply takes approximately 64 bit-serial operations. If both values are represented by 32-bits, the multiply takes approximately 1024 bit-serial operations.

Clearly, using 32-bits is unacceptable. But, C++ does not have types for representing just 7 and 5 bit values.

C++ can represent 8-, 16-, and 32-bit values using `char`s, `short`s, and `int`s. The automatic conversion operations that the compiler uses when the arguments are not the same type are bypassed for planes. At a small number of places in the modified compiler, the code that builds the tree representation detects a plane argument. Instead of the normal compiler code to determine the type for the node, special code determines the correct type for a plane operation from the type of the arguments.

The compiler uses special routines, specified in the machine description for the target machine (i.e., with `define_expand`), to generate the RTL for an operation. The compiler assumes that these routines need their arguments to be of the same machine-mode as the result of the operation. When generating `insn`s for arithmetic operations on planes the modified compiler uses a different routine, so that the arguments are not promoted to 32-bit `int`s and the results are not always 32-bits. This provided an acceptable compromise.

For peak practical efficiency, the abstract machine uses seven different plane modes; bit, signed byte, unsigned byte, signed half-word, unsigned half-word, signed word, and single precision IEEE floating point. A dyadic operator such as addition requires 49 different abstract machine instructions in order to utilize the different sizes and formats of both arguments[7]. Because scalars can be broadcast a bit at a time as part of an operation on the CAAPP, a distinction is made between *plane-plane*, *plane-scalar*, and *scalar-plane* argument pairs. This is more efficient than broadcasting the scalar to a plane register before the operation. This distinction adds another factor of three resulting in 147 different add instructions. As a result, the abstract machine designed for the CAAPP has approximately 3800 instructions. There is one instruction pattern defined for every instruction of the target machine.

Unfortunately, not all of the register-constant and constant-register instructions of the abstract machine are used because the Gnu C++ compiler does not distinguish seven different scalar modes.

---

[6] Performance is also affected by the ability to transfer data to and from the processors. This ability is proportional to the ratio of the number of pins available on a chip to the number of processors on the chip.

[7] Note that all constants are scalars.

The difference between signed and unsigned scalar values is not represented in the scalar modes. Because all arguments to standard Control Processor operations are first promoted to `int`s, the compiler uses the different scalar modes only to distinguish memory access types. This means that when a dyadic operation between a plane and a scalar is compiled, not all of the possible abstract machine instructions are used. Unsigned bytes are mapped to the mode QI, unsigned half-words to HI, other integer values are mapped to SI, and all scalar reals are converted to single precision floating point. The scalar modes QI, HI, SI, and SF are mapped to four of the seven possible scalar modes recognized by the abstract machine. The compiler has not yet been modified to add the additional scalar modes because this is a massive modification.

This is one of the few differences between IUA Class Library programs compiled by the modified compiler and IUA Class Library programs that use method calls. The different scalar modes are distinguished in the tree representation of the program and this tree representation is used to select the proper method to be called. As a result, the programs compiled with the modified compiler do not use the most efficient abstract machine instructions for all scalar-plane and plane-scalar cases because signed `char`s and signed `short`s have to be promoted to `int`s.

Because of the large number of instruction patterns required, a tool is used to generate them. This is possible due the *regularity* of the instruction patterns. It was possible to explore different ways of defining the target computer because this tool was available.

5 Dual-instruction Flow

To cause an operation to be performed on the CAAPP, it is necessary to instruct the Micro Controller to do this operation. This is accomplished by writing the abstract machine instruction generated by the Gnu compiler to a location (MC_LOCATION) in the Control Processor address space that is monitored by the Micro Controller.

For example, if the instruction for the abstract machine is [8]

```
AddPShortPInt 6,2
```

then the sequence necessary for doing this on the SPARC[?] is

```
sethi  %hi(MC_LOCATION),%o2
sethi  %hi(0x06020000),%o0
or     %o0,%lo(AddPShortPInt_opcode),%o0
st     %o0,[%o2 + %lo(MC_LOCATION)]
```

This code sequence places the instruction for the abstract machine (`AddPShortPInt 6,2`) in SPARC register `%o0` as a *constant*. It places the address for the Micro Controller location in another register

---

[8] Add a 32-bit integer plane to a 16-bit integer plane.

(`%o2`). The `sethi` instruction places a constant value in the high-order 22 bits of a register. The `%hi` and `%lo` designators select the top 22 bits and the bottom 13 bits, respectively, of a constant value. The `st` instruction stores the 32-bit register `%o0` into the location addressed by register `%o2` offset by a constant.

Three methods of generating these "meta" instructions were considered. The first method builds a tree representation of the SPARC operations necessary to generate and store the constant representing the CAAPP operation. Unfortunately, the constant has registers of the abstract machine specified in it. Using this method means that the compiler would have to allocate the specific abstract machine registers, as opposed to pseudo registers, prior to the optimization phases of the compiler. If actual hardware registers are used, very little optimization succeeds and data residency in hardware registers is minimal.

The second method generates `insns` for both the abstract machine instruction and the SPARC instructions. The compiler allocates pseudo registers for both the abstract machine and the SPARC for these operations. The optimization and register allocation logic of the compiler maximizes data residency in the hardware registers. Unfortunately, the optimization phases of the compiler can not see the dependence between the two sets of `insns` because there are no register dependencies. It separates the two sets, rendering the compiled code into garbage. This method also provides no way of converting the abstract machine instructions, *after optimization*, into the constants to be sent to the Micro Controller. It would be possible to provide `insns` for all the combinations of the two sets of instructions, however, the complexity of this approach makes it impractical.

The third method leaves the generation of the SPARC instructions to the post optimization phases when the final assembly language code is generated. `insns` are generated only for the abstract machine instructions. The resulting `insns` are optimized and matched with the instruction patterns for the abstract machine. When the assembly code is generated, routines are used to generate the SPARC instructions which create the constants representing the abstract machine instructions and send them to the Micro Controller.

There are two problems with this approach. A minor problem is that the SPARC instructions, for generating the abstract machine instructions, are not optimized. Consequently, the register containing the Micro Controller address may be set many times unnecessarily in a block of code. Since the abstract machine instructions generally take longer to execute than the SPARC instructions and the processing in the CAAPP is overlapped with the processing at the SPARC, this is not considered to be a great problem[9].

The more difficult problem is that the SPARC code generated in the final phase *requires* registers to accomplish its function. SPARC register 8 is required for constructing abstract machine instruction opcodes and other values. Register 9 is needed for the queue pointer required for virtualization. Register 10 is required for the constant MC_LOCATION and register 15 allows a function to be

---

[9] Use of the modified compiler greatly reduces the number of SPARC instructions required since programs for the CAAPP will generally perform most operations on the CAAPP.

called. At this point in the compilation, all the SPARC registers have been assigned. Changing the contents of any of these registers invalidates the generated code.

The Gnu compiler allows instruction patterns to specify that certain registers are modified even when this is not apparent from the operation in the `insn` (these registers are *"CLOBBERED"*). Originally, all of the instruction patterns for the abstract machine specify a small set of SPARC registers that are clobbered by each abstract machine instruction. These are the only registers used when generating assembly language instructions to send the abstract machine instructions to the Micro Controller. The instruction pattern shown below shows the clobber information added to the instruction pattern for adding an `IntPlane` and a `CharPlane` to produce an `IntPlane` result.

```
(define_insn
  [(set (match_operand:IP 0 "register_operand")
        (plus:IP (match_operand:IP 1 "register_operand")
                 (match_operand:CP 2 "register_operand")))
    (clobber (reg:SI  8))
    (clobber (reg:SI  9))
    (clobber (reg:SI 10))
    (clobber (reg:SI 15))]
  ""
"* return iua_gen_format2(operands, 0x0c29);")
```

This scheme requires that the modified compiler add the clobber RTLto every `insn` that is generated for an abstract machine instruction. However, due to the clobber RTL in the large number of abstract machine instruction patterns, compiling the compiler requires an inordinate amount of time and results in a very large executable.

The final solution omits the *clobber*s from the instruction patterns. Instead, it generates the clobber information when needed by recognizing abstract machine instruction patterns[10] during all phases of the compiler. Because it is necessary to generate the clobber information in numerous places, this is the single largest modification to the compiler.

In order to avoid producing the same Control Processor operations every time the assembly language is generated for an abstract machine instruction, the modified compiler uses a special pass that detects the first and last occurrence of `insns` for the abstract machine in a *block* and inserts special `insns` before the first and after the last. These `insns` cause setup and cleanup logic to be generated when the assembly language pass is performed. The setup logic sets the address of the Micro Controller into a SPARC register and loads the virtual queue pointer into another. The cleanup logic saves the virtual queue pointer and checks for a nearly-full-queue condition.

---

[10] The instruction pattern *attribute* facility is used for this purpose.

6 Activity

Every processor in the CAAPP executes the same instruction as every other processor in the CAAPP. Some processors, however, may be told to not *store* their result. The IUA Class Library defines that this capability affects only the assignment operator. The normal assignment statement

```
x = y;
```

replaces the value held by the variable `x` with the value in `y`. The IUA Class Library semantics for assignment of planes is analogous to the C++ statement

```
x = (active != 0) ? y : x;
```

That is, `x` is replaced by `y` only if `active` is not zero.

A variable on the CAAPP has thousands of values — one for each processor. Consider `x, y,` and `active` to be such variables; then, the above statement is performed at every processor using that processor's values for `x`, `y`, and `active`.

Activity is controlled by IUA Class Library statements that specify the activity at each processor[11]. For example if `x, y` and `z` are planes, the following code performs `assignment1` for every CAAPP processor for which `x` equals `y`, `assignment2` for every CAAPP processor for which `x` equals `y` and `z` equals zero, and `assignment3` for every CAAPP processor.

```
{Select active (x == y);
  assignment1;
  {Select active (z == 0);
    assignment2;
  }
  {Everywhere active;
    assignment3;
  }}
 assignment4;
```

The active processors for `assignment4` are not known until run-time if `assignment4` is not lexically enclosed in a block that sets the activity.

Activity is represented as a plane of bits — one bit per processor. If a BitPlane specifies the activity, then the assignment operation is affected and both the destination and the activity become inputs to the operation. `Everywhere` is treated as if there were an activity plane containing all ones and the assignment operation is performed at each processor just like normal assignment.

Originally, the modified compiler supported activity by producing a tree representation for the following pseudo-code:

---

[11] The *constructor-destructor* semantics of C++ are used to specify the activity.

```
  if (activity != NULL)
    destination = (*activity ? source : destination);
  else
    destination = source;
```

where `activity` is a pointer to an instance of a plane containing the values for each processor of the CAAPP. The code generator for the `?:` operator recognizes that a destination is a plane and generates the `insns`required. Depending on the complexity of the right-hand-side of the assignment, this often results in inefficient code that the optimization phases of the compiler have difficulty in improving.

To simplify the optimization problem, the code generation logic for the assignment operator was modified instead. This allows the right-hand-side to be represented in the tree just once. No special tree representation is required.

The code generator for assignment creates the `insns` represented by the following pseudo-code when the left-hand-side is a plane. An activity address of NULL indicates `Everywhere active`. The `set_activity` instruction affects only the operation of the abstract machine instruction that follows it.

```
  load scalar_register(a) with the address of the activity
  if scalar_register(a) == NULL goto everywhere
  load destination into plane_register(d)
  load activity into plane_register(b)
  set_activity from plane_register(b)
  move plane_register(s) to plane_register(d)
  goto join
everywhere:
  move plane_register(s) to plane_register(d)
join:
```

This code assumes that the value for the right-hand-side of the assignment operation is in `plane_register(s)` of the abstract machine.

If the compiler does no optimizations, this approach generates correct code. With optimization, the following, simpler sequence is often generated because the compiler does not recognize the dependency between the `set_active` and the `move`.

```
  load scalar_register(a) with the address of the activity
  move plane_register(s) to plane_register(d)
  if scalar_register(a) == NULL goto everywhere
  load activity into plane_register(b)
  set_activity from plane_register(b)
everywhere:
```

This is a problem because the move is no longer affected by the activity.

Defining additional instruction patterns, that show the dependence of the move upon the activity, solves the problem. The `move_using` pseudo-code operation below says that the move will be done using the activity in `plane_register(b)`.

```
  load scalar_register(a) with the address of the activity
  if scalar_register(a) == NULL goto everywhere
  load destination into plane_register(d)
  load activity into plane_register(b)
  move_using(plane_register(b)) plane_register(s) to plane_register(d)
  goto join
everywhere:
  move plane_register(s) to plane_register(d)
join:
```

During the assembly language generation phase, the `move_using insn` causes the compiler to generate the code, in the proper sequence, to set the activity and do the move.

If it is known at compile time that the activity is `Everywhere` (address of the activity plane is NULL), then the sequence

```
  move plane_register(s) to plane_register(d)
```

can be generated. Conversely, if it is known that the activity is not `Everywhere`, then the sequence

```
  load scalar_register(a) with the address of the activity
  load destination into plane_register(d)
  load activity into plane_register(b)
  move_using(plane_register(b)) plane_register(s) to plane_register(d)
```

can be generated. To allow the compiler to make this decision at compile time, the constructors and destructors for instances of the activity class are processed with special logic. The state of the activity for a *block* of user code is placed at compile time on the stack that is used for the binding information for user variables. If this information is not present when an assignment statement is processed, the full sequence is generated. Otherwise, the information stored allows one of the other simpler sequences to be selected. Because the simpler sequences do not contain branches and labels, the optimization phases of the compiler are able to do a better job.

7 Compatibility

The IUA Class Library does not support operations on planes of different shapes. Adding two planes with different shapes must generate an error. The modified compiler checks this condition at compile-time when possible or else generates code to check the condition at run-time. The attribute of shape, as used for planes is not part of the C++ language and, consequently, there are no Gnu C++ compiler facilities available for this.

The shape of a plane is defined using an initializer for the instances of the plane classes. Good programming style uses the same initializer for all the planes that have the same shape. If two planes have the same initializer, they have the same shape and no check need be generated when they are used in the same operation. The plane variable itself is used as the shape indicator in those cases where there is no initializer. For example, `x` and `y` are defined with the same initializer and have the same shape while `z` is defined using a different initializer and *may* have a different shape.

```
CharPlane x (40, 60);
CharPlane y (40, 60);
IntPlane  z (ps);
```

There is no slot in any existing compiler structure for keeping track of the shape of a plane. For plane variables, the information is available in the structure kept at run-time for these instances of the plane class. But, for expressions, the modified compiler generates no associated run-time structure because maintaining these structures would be additional overhead. The shape information is available in the initializing expression for variables at compile time. This information does not exist for expressions used as arguments to other operators.

The first attempt at a solution required that a slot to contain this shape information be added to the tree node structure for types. If the shapes are known at compile time, the compiler can compare the shapes from these slots and generate an error if necessary. If the shape is not known at compile-time, the compiler can use the information in the slot to generate code to access the shapes at run-time and perform the check.

The shape information must be propagated in the tree by various places in the compiler. If two planes are used in an operation, the tree node of the result must be made to point to the shape information of one of the plane arguments. The shape information of the plane arguments is assumed to be equivalent — otherwise either a compile-time or run-time error would have been generated.

Generating the run-time checks is straight forward. The generated tree structure is modified when the check is needed. If (X) is some tree structure for an operation that requires a check, then a compound tree structure (Check,X) is generated in place of (X)[12]. The code generated for the check compares the two shapes and generates an exception if they differ. These checks must be generated for operations such as assignment and addition.

There are three problems with this approach. First, propagating the shape information requires

---

[12] This compound structure is used to represent C++ compound expressions.

numerous scattered modifications to the compiler[13]. Second, the test and branch generated by the tree structure interferes with the optimization of the resulting code. Third, there is no way to eliminate the unneeded checks because the tree structure is not adequate for this purpose. For example, if a program consists of the following statements

```
x = y + z;
z = z + x;
```

there is no need for any checks to be generated in the second statement even if *it is not known* at compile time that `z` and `x` are compatible.

The first problem is solved by not propagating the shape information at all. Instead, each time the shape information is required, the sub-tree is searched for the leaf containing the variable declaration from which the shape information can be found. While this may seem to be very inefficient, in practice the normal sub-tree is not very deep.

The second problem is solved by defining a new type of tree node that represents the check and is converted to a new RTL operation. The `insn` containing this RTL is converted to a test and branch when the assembly language output is produced.

Because the `insn`s for a function now contain a representation of each check, the third problem is easy to solve. An additional pass to the compiler, before the optimization passes, scans the `insn`s looking for these checks. If it can be proved that an equivalent check has already been performed in a *block*, the check is deleted.

The modified compiler is able to eliminate nearly all run-time checks from most IUA Class Library programs. The cases where checks are still generated result from user code that uses different instance initializers or where there is no initializer, as is the case with function arguments and function results.

8 CAAPP Off-chip Memory References

The actual amount of CAAPP off-chip memory required for a plane is not known at compile time because of the *virtualization* of the CAAPP. The amount of memory required depends on the ratio of the size of the plane, as declared by the programmer, to the size of the physical CAAPP upon which the program is run which is not known at compile time. For this reason, the compiler cannot allocate CAAPP off-chip memory for either `static` or `automatic` plane variables. The compiler cannot allocate a Control Processor register for the CAAPP frame pointer and manipulate it using offsets as is normal for other processors because the offset values are not known at compile time. Instead, the compiler generates code that calculates the amount of CAAPP off-chip memory required at run time and dynamically allocates the space.

---

[13] This would not be a problem if being able to use new versions of the unmodified Gnu C++ compiler were not a requirement.

```
(insn 1744 1737 1739
  (set (reg:IP 76)
       (mem:IP
         (plus:SI (reg:SI 30)
                  (const_int -224)))
       ))
```

Figure 8: CAAPP Off-chip Memory Load `insn`

```
(insn 1744 1737 1739
  (set (mem:UP
         (reg:SI 14))
       (reg:UP 76)
))
```

Figure 9: CAAPP Off-chip Memory Store `insn`

Two cases occur. Either the user declares the plane instance with a source language declaration or the compiler creates the plane as a temporary (e.g., value parameter to a function). In the first case, the plane is instantiated when the source block in which it is called is entered. The amount of space needed is calculated at that time from the initializer by the `constructor` function for that plane class. In the second case, the compiler assumes that the plane will be given a value before it is referenced. The space needed is calculated from the current *virtual* CAAPP size when this value is stored. To make this work, the modified compiler generates code to *zero-out* the Control Processor memory used for instances of temporary planes at the beginning of the function in which they are created. This flags these instances as needing space to be allocated. If such a plane is referenced before it is given a value, a hardware fault is generated.

With minor, additional modification, the compiler generates `insn`s to load and store planes from abstract machine registers. These `insn`s look identical to the `insn`s used to load and store scalar values except that the mode is different (e.g., mode IP instead of SI) as is shown in Figure **??**. The address generated is the address of a plane instance and is not a valid address value for the CAAPP off-chip memory. However, the plane instance may be used to obtain the proper CAAPP address. The `insn` in Figure **??** is converted to the assembly language below that accomplishes this task.

```
   ld     [%fp-224],%o7                    Load pointer to CAAPP address
   lduh   [%o7],%o0                        Load (unsigned short) off-chip memory ad-
   sll    %o0,16,%o0                       dress
   sethi  %hi(MC_LOCATION),%o2             Align CAAPP address
   or     %o0,StoreInt12_opcode,%o0
   st     %o0,[%o2 + %lo(MC_LOCATION)]     Create instruction - LoadInt [%o0],%p12
                                           Send to Micro Controller
```

Because the space for planes is allocated at run time, this allocation must be performed before the data transfer is performed. The `insn` in Figure ?? is converted to the assembly language below that accomplishes this task.

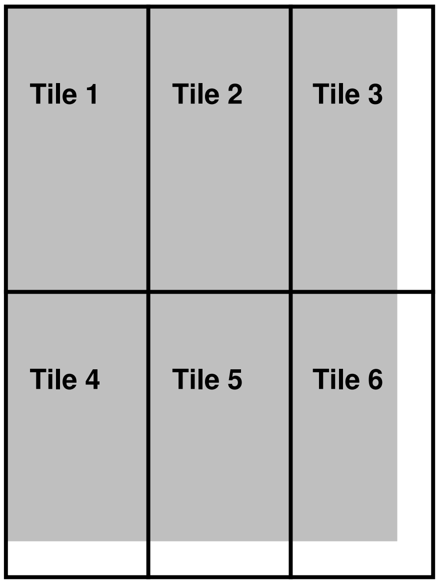
```
   ld     [%o6],%o7                        Load pointer to CAAPP address
   cmp    %o7,0
   bne,a  label                            Branch if plane is instantiated
   lduh   [%o7],%o0                        Load CAAPP off-chip memory address
   call   _IuaCl_StackUse_UShortPlane,0    Allocate and return CAAPP address
   mov    %o6,%o0                          Stack address of plane instance
   ld     [%o0],%o7
   lduh   [%o7],%o0
label:
   sll    %o0,16,%o0                       Align CAAPP address
   sethi  %hi(MC_LOCATION),%o2
   or     %o0,StoreInt12_opcode,%o0        Create instruction - StoreInt  %p12,[%o0]
   st     %o0,[%o2 + %lo(MC_LOCATION)]     Send to Micro Controller
```

Since the SPARC uses a delay slot with branching, the instruction following the `call` is actually performed as the `call` is performed. The first `lduh` instruction, in the delay slot for the `bne` branch instruction, is executed only if the branch is taken.

9 Virtualization

Technology limits the size of CAAPP that can be built and is currently limited to a maximum of 128 × 128 processing elements. Researchers routinely process digital images that approach 1024 × 1024 pixels. To allow researchers to use the CAAPP for their work on these large images, the IUA Class Library is designed to support programs that are independent of the actual size of the CAAPP on which the IUA Class Library program is run. The basic concept used to accomplish this is called *virtualization* and the technique used to implement this concept is called *tiling.*

Virtualization is accomplished by assuming that there is a virtual processing element for each



Each tile is the size of the physical CAAPP which is 64 x 32 in this example. If the plane is declared to be of size 120 x 90, then the plane is implemented using six tiles.

▨ Plane as defined.

Figure 10: Tiling a *Plane*.

element of a plane. There are two ways that the virtual processing elements can be distributed to the physical processing elements of the CAAPP. Virtual processing elements that are adjacent can be mapped to the same physical processing element; or, the virtual processing elements that are adjacent can be distributed to adjacent physical processing elements as shown in Figure **??**. The CAAPP's Coterie network[**?**] is much more efficiently virtualized using the second method so that method is used.

In Figure **??** an image is shown partitioned into tiles. Each tile is the size of the CAAPP. An integral number of tiles is always used. In this example, one physical processing element of the CAAPP is used for six elements of the plane. When two planes are added together, each CAAPP processing element must perform six additions. Regardless of the size of the plane, it can be partitioned into tiles[14]. The number of tiles determines how many times operations are performed at each processing element.

Two methods of accomplishing virtualization using tiles are available. Either each operation can be repeated for each tile as it occurs, or a sequence of operations can be repeated for each tile. The second method requires that operations be saved in a queue and the entries in this queue be repeated for each tile. Regardless of the method chosen, there must be one set of abstract machine registers for each tile. If the first method is used, there must be a low latency, high bandwidth

---

[14] The amount of memory available to each processing element results in a maximum number of tiles that can be used.

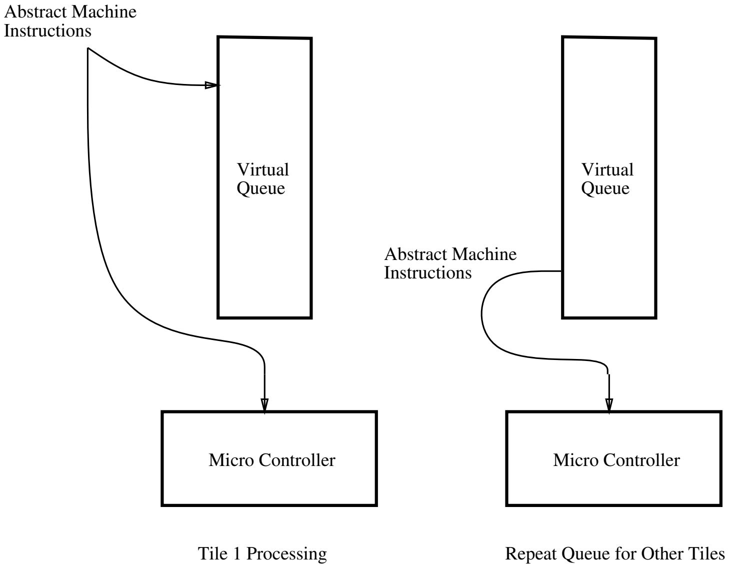


Figure 11: Queuing the Tile Operations.

path to the memory containing these register sets because each register set will be accessed for each operation. If the second method is used, there must be a low latency, high bandwidth path to just one register set. However, there must be a switch to another register set every time the queue is repeated for another tile.

The second method was chosen because the CAAPP on-chip memory is too small to contain more than one register set. Each time the queue is repeated, one register set is transferred off-chip and another register set is transferred on-chip. Obviously, the more entries that are in the queue before it must be repeated, the better the on-chip memory utilization will be.

As shown in Figure **??** each abstract machine instruction is sent directly to the Micro Controller. At the same time the instruction is written to the queue so that it can be repeated for the other tiles. The instructions for placing the abstract machine instruction in the queue are generated by the same logic that generates the instructions to send the abstract machine instructions to the Micro Controller (see Section **??**). Below is the SPARC code sequence for an abstract machine instruction (MultPShortPUChar) that is placed on the queue and sent to the Micro Controller.

```
sethi %hi(_IuaVm_queue_ptr),%o0
ld    [%o0+%lo(_IuaVm_queue_ptr)],%o1  Load pointer into queue
sethi %hi(MC_LOCATION),%o7
sethi %hi(0x090c0000),%o0              Align abstract machine register info
or    %o0,%lo(MulPShortPUChar_op),%o0  Add abstract machine opcode
st    %o0,[%o1]                        Place instruction in the queue
st    %o0,[%o7 + %lo(MC_LOCATION)]     Send instruction to Micro Controller
add   %o1,4,%o1                        Increment queue pointer
sethi %hi(_IuaVm_queue_ptr),%o0
st    o1,[%o0+%lo(_IuaVm_queue_ptr)]   save new queue pointer
```

Operations are saved on the queue (called the *virtual queue*) until one of three conditions is encountered:

1. the virtual queue is full,
2. virtual operations — the processing in each tile needs to be synchronized with the other tiles, or
3. the size of the virtual CAAPP changes.

When one of these conditions is true, the queue is flushed. The routine that flushes the queue informs the Micro Controller of the tile number for which the queue is being sent. The entries in the queue are sent, *unmodified*, to the Micro Controller for each tile. The Micro Controller uses the tile number to adjust the addresses used in CAAPP off-chip transfers.

The modified compiler assumes that the virtualization process is transparent to most phases of the compiler so that virtualization does not affect how optimizations are performed. Therefore, the state of the abstract machine registers is preserved for every tile so that queue flush operations may be performed without changing the state of the abstract machine as understood by most phases of the compiler.

Virtual Queue Full

The first case is handled by compiling checks at appropriate places in the Control Processor program that determine if the queue is full enough to be emptied. If the queue is over 75% full, the queue is flushed. The pass that sets up the SPARC registers for the dual-instruction flow (see Section **??**) inserts the check at the end of a sequence of abstract machine instructions.

Virtual Operations

The IUA Class Library defines methods for communication that require synchronization of the tiles. There are methods for:

Mesh communication: A plane is shifted across the CAAPP grid.

```
resultplane = plane.East()  // east neighbor
resultplane = plane.West()  // West neighbor
resultplane = plane.North() // North neighbor
resultplane = plane.South() // South neighbor
```

Generalized routing: The value from each CAAPP processing element is sent to the addressed CAAPP processing element.

```
resultplane = plane.Route(address_plane) // Send plane to
sumplane    = plane.RoutePlus(address_plane) // send and combine
```

Region Operations: The plane is segmented into regions. Communication within each region, in parallel across regions, allows powerful algorithms to be implemented[?].

```
// select min value in each region (region defined simarly to activity)
resultbitplane = plane.RegionSelectMin(select_plane)
// select max value in each region
resultbitplane = plane.RegionSelectMax(select_plane)
// broadcast sender's value to all other PEs in region
resultplane = plane.RegionBroadcast(select_plane)
// select min value in plane
resultbitplane = plane.SelectMin(select_plane)
// select max value in plane
resultbitplane = plane.SelectMax(select_plane)
```

Global Feedback: The Control Processor may make decisions based upon feedback from the CAAPP.

```
scalar = plane.Count() // count of non-zero
scalar = plane.Any()   // are any non-zero?
scalar = plane.Access(select_plane) // Or of selected values
```

The compiler recognizes these operations and generates inline instructions to accomplish them. There is an example of `Count` in Appendix **??**

Virtual operations require that all tiles be at the same point in the processing. The usual reason for this is that information must be communicated between the tiles. If the mesh communication network is used to transfer data from CAAPP processing element to CAAPP processing element, data must also be transferred between processing elements at the edges of adjacent tiles.

These are all cases where the abstract machine instruction to be performed must be aware of the way virtualization is achieved and how the information for the tiles is structured. These operations are called *virtual instructions* for this reason. The compiler generates the following items in the code sequence for every virtual instruction:

Save registers instruction: The compiler generates the instruction to save the register set. This instruction is placed on the virtual queue so that it is performed for each tile. For efficiency, as part of generating this instruction, the compiler specifies which registers will be needed for the virtual operation, and which registers will be needed *after* the instruction, so that only those registers will be saved. The `insn`for the save is inserted by a special pass that occurs after all registers have been allocated. This pass scans the `insn`s to find the virtual instructions generated by the RTL generation phase of the compiler. It inserts the `insn` for a save instruction just before the virtual instruction and the `insn` for a restore instruction just after. When the assembly language for the save instruction `insn` is generated, a call to the routine that processes the virtual queue is inserted just after the register save instruction.

Virtual instruction: All virtual operations are invoked through method calls on instances of the plane classes. The C++ front end of the compiler generates special tree nodes for these operations instead of the function-call tree nodes that are normally generated for class method calls. The generation of `insn`s for the virtual instruction is similar to that for other operations such as add or multiply. The virtual instructions access their data from the register sets saved in CAAPP off-chip memory.

Restore registers instruction: The restore instruction restores the registers that will be used after the virtual instruction. This may be a different set of registers because some registers may be *born* and others *die* as a result of executing the virtual instruction. For both the save and restore, if the register set is empty, the save and/or restore is not generated. This restore instruction is placed on the virtual queue so that it is performed for each tile.

The abstract machine instructions actually sent to the Micro Controller looks like the following sequence:

```
Restore Tile 1 registers
        Tile 1 operations
Save    Tile 1 registers
Restore Tile 2 registers
        Tile 2 operations
Save    Tile 2 registers
...
...
Restore Tile n registers
        Tile n operations
Save    Tile n registers

Virtual Instruction

Restore Tile 1 registers
        Tile 1 operations
Save    Tile 1 registers
...
```

The instructions for tile 1 are always sent directly to the Micro Controller. The instructions for all other tiles are always placed on the virtual queue first.

Virtual CAAPP Size

The third case is similar to the *compatibility* problem discussed in Section **??**. Not only must the shape of two planes be compared for an operation, that shape must be compared to the *shape* of the virtual CAAPP. If there is a difference between the shapes of the next operation and the previous operation, the tiles are synchronized before the size of the *virtual* CAAPP is changed[15]. Then, a new *virtual* CAAPP is defined and operations proceed with this new definition. The compiler generates a check when a new *block* is entered[16], when the compiler can detect when the size will change, and when the compiler can't determine if the size will change. The check consists of a call to a function that compares the new size to the current size. If the size changes, this routine causes the virtual queue to be flushed and then changes the *virtual* CAAPP size. This routine is passed the address of the plane instance from which it obtains the new size.

---

[15] This approach requires results in less state information that must be saved for each tile

[16] More complex analysis using data flow could reduce these instances of the checks.

Unlike the compatibility checks where the information required is lexically together, the *virtual* CAAPP size check depends on the flow of the program and must be determined for *every* abstract machine instruction. Entry to a *block* requires a check because the block is entered via a branch (that's what makes it a block). The branch may come from some point in the process where the virtual CAAPP size is different and there is no easy way at compile time to determine this.

As all passes of the compiler are capable of generating `insn`s for abstract machine instructions, the process of adding `insn`s for checks can be accomplished only after all other `insn`s have been generated. The information needed by the compiler is available when the `insn` for an abstract machine operation (e.g. add) is generated. Consequently, this information *must be kept with the* `insn` .

At first, this information was placed in a new field in the `insn` structure. This approach failed because some parts of the compiler (e.g., reload) generate new abstract machine `insn`s in a way that does not allow easy access to the existing `insn` which contains the size information. A special pass added to propagate the size information was also unable to find it in all cases.

To solve the problem, the modified compiler adds a new field to the RTL structure (a significant change to the compiler) to hold this size information. Critical places in the compiler propagate this information when RTL is copied. A final pass completes the propagation and generates the needed checks.

Currently, the modified compiler generates checks in approximately 30% of the possible places (i.e., for 3 of ten abstract machine `insns`). At first, all but one of the cases where the check must be generated was properly processed. The one case resulted because the IUA Class Library allows a plane to be initialized to a scalar. To accomplish this operation, two abstract machine instructions are generated. The first broadcasts the scalar value to an abstract machine register. The second instruction saves the abstract machine register to the CAAPP off-chip memory where the plane is kept. The `insn`s for both instructions contain the needed size information. However, the size information is the address of a plane instance *which may not have been instantiated yet* when the code from the first `insn` is executed. (Because of the need to dynamically allocate addresses in off-chip memory for planes some planes are instantiated when they are first set to a value – see Section **??**). A new abstract machine instruction that broadcasts a scalar directly into CAAPP off-chip memory solves this problem by combining the two `insn`s into one. Thus, if the plane has not yet been instantiated, its size must be the current size. It the plane has been instantiated then its size is available and can be used for the check.

10 Conclusions

The modified compiler successfully produces code for programs written using the IUA Class Library. It satisfies the requirements for efficiency and virtualization. The object modules produced are independent of the size of the CAAPP processor. Images many times larger than the physical machine may be used without requiring programmers to make any changes to their programs.

Appendix **??** shows timings from portions of the DARPA IU benchmark[?][?] comparing the code generated by the modified compiler with code that uses the standard Gnu C++ compiler and implementation of the IUA Class Library. These times were produced using the functional simulator for the Micro Controller and CAAPP[?]. This simulator provides timings that should be very close to those achieved with the actual hardware. The simulator models the concurrent processing between the Micro Controller, CAAPP processors, and data transfers on and off the CAAPP chips[17]. The Control Processor also runs concurrently. However, as the Control Processor is not simulated, it is not known what the relative times are for it. The actual elapsed time for the benchmark will be the maximum of the time spent in the CAAPP, Micro Controller, and Control Processor time.

The modified compiler produces better times for the intensity image portion of the benchmark. However, the times generated by the compiled code for the much simpler, depth image portion are significantly worse. The reason for this is that the depth image processing spends most of its time in the small nested loops. Because the inner loops are very small and the standard implementation uses run-time dynamic register allocation which is able to eliminate all loads and stores of abstract machine registers from these loops, most off-chip memory transfers are eliminated. In contrast, the modified compiler is unable to eliminate any loads or stores or find any common sub-expressions because the loops are so simple.

What is not shown is that the modified compiler is able to greatly reduce the overhead in the Control Processor associated with sending the instructions to the Micro Controller. This reduction is assumed to be well over 90%. Since the total elapsed time for any CAAPP program depends on the maximum of the times from the three asynchronous processors, reducing the Control Processor time could be very important.

Future work will verify these results on actual IUA hardware. The compiler will be kept up-to-date with the latest Gnu C++ release so that any bug fixes and performance improvements, that are produced by the many people who are working on the standard Gnu C++ compiler, may be utilized.

Some of the problems encountered in doing this work resulted from the structure of the IUA Class Library definition. For example, the IUA Class Library allows three different forms of initializers for plane instances which made it more difficult to determine if two planes were compatible or if the virtual machine size might change. Future definitions may consolidate the three cases into one. Another problem results from planes that are function arguments or results and have no initializers that can be used for these checks. In may be advantageous to add optional statements that can be used for specifying or asserting that these instances are compatible with other instances.

Many difficulties were due to the small number (4) of IntPlanes that could be kept in the

---

[17] There are rate buffering mechanisms in hardware between the Micro Controller & CAAPP and the CP & Micro Controller that allow them to proceed independently.

abstract machine registers at one time. This causes trouble with various phases of the Gnu compiler – primarily the reload phase. Future versions of the CAAPP chip may have more on-chip memory so that more abstract machine registers can be defined.

The existence of a C++ compiler, for which the source code was available, allowed this project to succeed. Without it, it would have taken too long to produce the basic C++ compiler. The cost of using an existing compiler is that some of the most efficient structures and algorithms could not be used due to the sheer magnitude of the modifications to the Gnu compiler that would have been required. The solutions to these problems could have been designed into the basic structures of a compiler built from scratch.

11 Acknowledgments

I would like to thank all of the people who have contributed to the Gnu C++ compiler. I would especially like to thank Richard Stallman of the Free Software Foundation for developing and nurturing the idea of gcc, Michael Tiemann of Cygnus, Inc. for doing the C++ front end to gcc, Jim Wilson of Cygnus & Richard Kenner of NYU for patiently answering my questions, and Heidi Roskuski of Amerinex AI for changing the Micro Controller microcode for the abstract machine when I needed it.

This work was supported in part by DARPA Project#DAAL02-91-C-0051. This work is in partial fulfillment of CLIN 0001.

References

[1] J. Burrill, *The Class Library for the IUA Tutorial*, Amerinex Artificial Intelligence, Inc. 409 Main Street Amherst, Ma. 01002

[2] J. Burrill, *A C++ Class Library for the IUA*, Amerinex Artificial Intelligence, Inc. 409 Main Street Amherst, Ma. 01002

[3] J. Burrill, H. Roskuski, *IUA Abstract Machine Instruction Reference manual*, Amerinex Articial Intelligence, Inc. 409 Main Street Amherst, Ma. 01002

[4] M.C. Herbordt, C.C. Weems, M.J. Scudder, *Non-Uniform Region Processing on SIMD Arrays Using the Coterie Network*, Machine Vision and Applications, 5 (2), 1991.

[5] D.B. Shu, J.G. Nash, *The Gated Interconnection Network for Dynamic Programming*, in Concurrent Computations, (S.K. Tewksbury, B.W. Dickinson, and S.C. Schwartz, eds.), Plenum, New York, 1988, pp. 645-658.

[6] R. Stallman, *Using and Porting Gnu CC*, Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

[7] *The SPARC$^{\text{TM}}$ Architecture manual*, Sun Microsystems, Inc., 2550 Garcia Avenue, Mountain View, CA 94043.

[8] C. Weems, *Image Processing on a Content Addressable Array Parallel Processor*, Technical Report 84-14, Department of Computer and Information Science, University of Massachusetts.

[9] C. Weems, J. Burrill, *The Image Understanding Architecture and its Programming Environment* Parallel Architectures and Algorithms for Image Understanding, (V.K. Prasanna Kumar, ed.), 1991, Academic Press, 1250 Sixth Avenue, San Diego, CA 92101.

[10] C. Weems, S. Levitan, et al, *The Image Understanding Architecture*, International Journal of Computer Vision, Vol. 2, No. 3, January, 1989, pp. 251-282, Kluwer Academic Publishers, Hingham, MA.

[11] C. Weems, et al, *An integrated image-understanding benchmark: recognition of a 2 1/2 D "mobile"*, Proceedings of the 1988 DARPA Image-Understanding Workshop, 1988, pp. 111-126.

[12] C. Weems, et al, *Preliminary Results from the DARPA Integrated Image-Understanding Benchmark Exercise* Parallel Architectures and Algorithms for Image Understanding, (V.K. Prasanna Kumar, ed.), 1991, Academic Press, 1250 Sixth Avenue, San Diego, CA 92101.

A Example Assembler Output

Below is a fragment of the assembly language output for a SPARC processor generated for a program compiled with the modified Gnu C++ compiler. General notes:

- Assembly language comments begin with "!".
- Registers of the abstract machine are specified as %p*n* where n is the register number. There are 16 8-bit registers in the abstract machine. Multiple consecutive registers are used to hold IntPlane and ShortPlane values.
- The address of the stack frame of the current function is in register %fp.
- This code was compiled for use with the CAAPP simulator. Consequently, it calls a function (_IuaMc_pFIFOl) to send an instruction word to the Micro Controller instead of directly writing it directly into memory.

The assembly language code below was generated for the following C++ code. The assembly language code for only part of the first line is included.

```
BitPlane on_hull (ps);
BitPlane right_angle_p (ps);
ShortPlane p21_row (ps);
ShortPlane p21_col (ps);
ShortPlane p32_row (ps);
ShortPlane p32_col (ps);
IntPlane norm_val (ps);
IntPlane val1 (ps);
IntPlane val2 (ps);
...
val1 = p21_col * p32_col + p21_row * p32_row;
val2 = 512 * val1 * val1;
right_angle_p = on_hull & (val2 < norm_val);

if (debug_benchmark)
 {
  printf ("%d right angles\n", right_angle_p.Count ());
  intensity_image.DisplayOv (right_angle_p,1);
 }
```

```
 call   _IuaCl_TileCheck
 add    %fp,-64,%o0
```

This is a check to see if the virtual machine size should be changed. The address of the plane that determines the new size is loaded into SPARC register `%o0` and the function is called. Note that the SPARC processor uses a *delay slot* with transfer of control instructions.

```
 sethi  %hi(_IuaVm_queue_ptr),%o1
 ld     [%o1+%lo(_IuaVm_queue_ptr)],%o1
 !Setup
```

Set up the registers needed for sending instructions to the Micro Controller. The virtual queue pointer is placed in SPARC register `%o1`. This code was compiled for use with the CAAPP simulator so the MC_ADDRESS is not needed.

```
 sethi  %hi(0x00080c9b),%o0
 or     %o0,%lo(0x00080c9b),%o0
 call   _IuaVm_pFIF01
 st     %o0,[%o1 + 0]
 !MulPShortPShort     %p0,%p0,%p8
```

Multiply the ShortPlane in `%p0` by the ShortPlane in `%p8` producing an IntPlane in `%p0`. Note the the abstract machine register numbers (0x0008) are combined with the abstract machine instruction opcode (0xc9b) and then sent to the Micro Controller. The instruction word is saved on the virtual queue at offset 0 from the current queue pointer value.

```
 sethi  %hi(0x00040c2d),%o0
 or     %o0,%lo(0x00040c2d),%o0
 call   _IuaVm_pFIF01
 st     %o0,[%o1 + 4]
 !AddPIntPInt     %p0,%p0,%p4
```

The result is then added with `%p4`. Note the the virtual queue offset is now 4.

```
 ld     [%fp+-184],%o7
 cmp    %o7,0
 bne,a  iua9
 lduh   [%o7],%o0
 call   _IuaCl_StackUse_IntPlane
 add    %fp,-184,%o0
 ld     [%o0],%o7
 lduh   [%o7],%o0
iua9:
```

The contents of `%p0` are saved in the off-chip memory (`val1`). First a check is performed to see if the memory has been allocated. If it has not been allocated, `_IuaCl_StackUse_IntPlane` is called to do the allocation. This routine is given the address of the IntPlane class instance. The address will be placed in the instance. The branch and label are generated in the pass that produces the assembly language output.

```
sll    %o0,16,%o0
or     %o0,0x2d0,%o0
call   _IuaVm_pFIFOl
st     %o0,[%o1 + 8]
!StoreInt       %p0,[%fp+-184]
```

Actually perform the store operation. The address of the plane in off-chip memory is combined with the opcode for the store operation.

```
sethi   %hi(0x0400006d),%o0
or      %o0,%lo(0x0400006d),%o0
call    _IuaVm_pFIFOl
st      %o0,[%o1 + 12]
!MovIntInt      %p4,%p0
```

```
sethi   %hi(0x040000d5),%o0
or      %o0,%lo(0x040000d5),%o0
call    _IuaVm_pFIFOl
st      %o0,[%o1 + 16]
mov     %lo(0x00000009),%o0
st      %o0,[%o1 + 20]
call    _IuaVm_pFIFOl
mov     %o0,%o0
!LShiftIntC     %p4,%p4,9
```

This is a multiply by 512 performed by shifting left 9 bits in place. Note that this abstract machine instruction requires two words to be sent to the Micro Controller. The second word is the scalar value specifying the shift amount.

```
 sethi  %hi(0x04000cad),%o0
 or     %o0,%lo(0x04000cad),%o0
 call   _IuaVm_pFIFOl
 st     %o0,[%o1 + 24]
 !MulPIntPInt     %p4,%p4,%p0
 ld     [%fp+-192],%o7
 cmp    %o7,0
 bne,a  iua10
 lduh   [%o7],%o0
 call   _IuaCl_StackUse_IntPlane
 add    %fp,-192,%o0
 ld     [%o0],%o7
 lduh   [%o7],%o0
iua10:
 sll    %o0,16,%o0
 or     %o0,0x2d4,%o0
 call   _IuaVm_pFIFOl
 st     %o0,[%o1 + 28]
 !StoreInt     %p4,[%fp+-192]
```

Do another multiply and store. Note that the virtual queue offset is now at 28.

```
 sethi  %hi(0x040c0f2d),%o0
 or     %o0,%lo(0x040c0f2d),%o0
 call   _IuaVm_pFIFOl
 st     %o0,[%o1 + 32]
 !LtPIntPInt     %p4,%p4,%p12
```

Compare register `%p4` and register `%p12` placing a BitPlane result in register `%p4`.

```
 call   _IuaVm_FlushCheck_v
 add   %o1,36,%o0
 mov   %o0,%o1
 !Cleanup
 ld    [%fp-276],%o4
 ld    [%fp-276],%o0
 call  _IuaCl_TileCheck
 mov   %o0,%o0
```

Because a check is required to see if the virtual machine size will change, the virtual queue pointer must be updated and saved. This is accomplished by calling `_IuaVm_FlushCheck_v` which will flush the queue if it is full enough. The check is then made for a virtual machine size change. If the virtual machine size changes, the queue will be flushed unless it was just flushed. The size is obtained from the plane whose address is `%fp-276`. Whenever the queue is flushed, all registers are saved and restored.

```
 sethi  %hi(_IuaVm_queue_ptr),%o1
 ld     [%o1+%lo(_IuaVm_queue_ptr)],%o1
 !Setup
 ld     [%o4],%o0
 lduh   [%o0],%o0
 sll    %o0,16,%o0
 or     %o0,0x200,%o0
 call   _IuaVm_pFIFOl
 st     %o0,[%o1 + 0]
 !LoadBit     %p0,[%o4]
```

Set up the virtual queue pointer again and then load a BitPlane into register `%p0`.

```
 sethi %hi(0x04000d40),%o0
 or    %o0,%lo(0x04000d40),%o0
 call  _IuaVm_pFIFOl
 st    %o0,[%o1 + 4]
 !AndPBitPBit     %p4,%p4,%p0
 ld    [%fp+-64],%o7
 cmp   %o7,0
 bne,a iua11
 lduh  [%o7],%o0
 call  _IuaCl_StackUse_BitPlane
 add   %fp,-64,%o0
 ld    [%o0],%o7
 lduh  [%o7],%o0
iua11:
 sll   %o0,16,%o0
 or    %o0,0x284,%o0
 call  _IuaVm_pFIFOl
 st    %o0,[%o1 + 8]
 !StoreBit     %p4,[%fp+-64]
```

And the BitPlanes in registers `%p4` and `%p0` and store the result.

```
call  _IuaVm_FlushCheck_v
add   %o1,12,%o0
mov   %o0,%o1
!Cleanup
sethi %hi(_debug_benchmark),%o0
ld    [%o0+%lo(_debug_benchmark)],%o0
cmp   %o0,0
be    L182
st    %o0,[%fp-1292]
```

Another cleanup must be done at this point because of the test and possible branch. The branch is taken if `_debug_benchmark` is zero.

```
sethi %hi(_IuaVm_queue_ptr),%o1
ld    [%o1+%lo(_IuaVm_queue_ptr)],%o1
!Setup
```

```
sethi %hi(0x100000),%o0
or    %o0,0x35,%o0
call  _IuaVm_pFIFOl
st    %o0,[%o1 + 0]
!Save 0x10
call  _IuaVm_FlushComm_v
add   %o1,4,%o0
mov   %o0,%o1
call  _IuaVm_Count_v
mov   0x4,%o0
mov   %o0,%o4
!Count     %o4,%p4
```

The Count operation returns a count of the one bits in a BitPlane. All tiles must be synchronized before this operation can be performed. The synchronization is accomplished by calling `_IuaVm_FlushComm_v` which iterates the virtual queue for each tile. Its argument is the current queue pointer. For this to work properly, the register set must be saved for the current tile and restored for the next tile. As this can be very expensive, only those registers that will be used in the future are saved and restored. The opcode for Save (0x35) is combined with a bit mask (0x0010) specifying which registers will be saved and sent to the Micro Controller *and the virtual queue*. The Count operation is performed on register `%p4` which is not needed afterwards. Consequently, no restore is required in this case as no other register value is needed. The Count is accomplished by _IuaVm_Count_v which looks at the register values (i.e, `%p4`) from each tile register set and returns the count in SPARC register `%o0`.

```
 st    %o4,[%fp-1300]
 sethi %hi(LC19),%o0
 or    %o0,%lo(LC19),%o0
 ld    [%fp-1300],%o1
 call  _printf,0
 nop
 add   %fp,-64,%o4
 st    %o4,[%fp-1308]
 ld    [%fp-268],%o0
 ld    [%fp-1308],%o1
 mov   1,%o2
 call  _DisplayOv__C5PlaneR8BitPlaneUc,0
 nop
L182:
```

The printf function is called to print out the count. Then, the DisplayOV method is called to display the BitPlane from which the count was obtained as an overlay on top of the display of a UCharPlane.

B Timing Results

The tables below show the times for the intensity image and depth image processing portion of the DARPA IU benchmark using different CAAPP and image sizes. The CAAPP size is given by the number of tiles into which the image must be segmented. A larger CAAPP requires fewer tiles. The image size is square with either 64 or 128 rows. Also shown is a comparison between the version compiled by the modified Gnu C++ compiler (MGC), where the compiler does the register allocation, and the standard implementation (DRA) using run-time dynamic register allocation. All times are reported in clock cycles. The cycle time of the CAAPP is the same as the cycle time of the Micro Controller (10 megahertz).

The times are linear in number of tiles and follow the formula $T = C + n \times \left(K + (n > 1) \times \tilde{K}\right)$ where n is the number of tiles and K is the time per tile. The term $\tilde{K}$ depends on the number of tile switches required which is a function of the user program. Thus, $\tilde{K}$ is very high for the intensity image processing and very low for the depth image processing[18]. If there is only one tile, very little multiple-tile overhead is accrued.

For the depth image processing, the times reported are the same for the cases with the same number of tiles. This is not true for the intensity image processing because the larger image returns different feedback that affects the algorithm.

---

[18] It is frequent use of the operations which communicate, such as mesh and coterie, that make $\tilde{K}$ so large.

1	2	3	4	5	6	7	8	9	10
	Image Size	Tiles	pFIFO entries	Mc Inst.	iFIFO entries	CAAPP Inst.	CAAPP Idles	CAAPP Idles	CAAPP Time
DRA	64	1 × 1	42,744	432,354	300,276	275,571	8,307	308,946	592,824
MGC	64	1 × 1	34,016	387,621	272,959	250,802	5,208	282,346	538,356
DRA	128	1 × 1	43,229	434,701	301,329	276,256	7,724	313,200	597,180
MGC	128	1 × 1	34,268	388,672	273,250	250,971	5,131	283,760	539,862
DRA	64	1 × 2	101,780	1,976,700	1,108,289	1,002,183	172,250	1,342,005	2,516,438
MGC	64	1 × 2	74,588	1,854,824	1,054,333	961,177	180,205	1,221,270	2,362,652
DRA	128	2 × 1	102,621	1,934,129	1,064,991	958,695	171,279	1,350,262	2,480,236
MGC	128	2 × 1	75,139	1,801,921	1,003,271	910,311	175,864	1,226,720	2,312,893
DRA	128	2 × 2	203,463	3,830,448	2,169,659	1,953,625	297,215	2,753,030	5,003,870
MGC	128	2 × 2	148,499	3,564,202	2,046,219	1,856,857	303,358	2,507,008	4,667,223
DRA	128	2 × 4	405,147	8,196,521	4,812,659	4,379,525	681,732	5,472,174	10,533,431
MGC	128	2 × 4	297,169	8,073,183	4,962,659	4,581,029	691,073	5,028,500	10,300,602

Intensity image processing portion of the DARPA IU benchmark – March 26, 1993

1	2	3	4	5	6	7	8	9	10
	Image Size	Tiles	pFIFO entries	Mc Inst.	iFIFO entries	CAAPP Inst.	CAAPP Idles	CAAPP Idles	CAAPP Time
DRA	64	1 × 1	4,953	56,332	40,353	37,778	374	68,153	106,305
MGC	64	1 × 1	5,205	64,878	48,658	44,418	2,135	94,223	140,776
DRA	128	1 × 1	4,953	56,332	40,353	37,778	374	68,153	106,305
MGC	128	1 × 1	5,205	64,878	48,658	44,418	2,135	94,223	140,776
DRA	64	1 × 2	9,587	114,285	80,491	75,714	1,147	131,098	207,959
MGC	64	1 × 2	10,555	137,529	100,543	90,292	825	228,350	319,467
DRA	128	2 × 1	9,587	113,897	80,107	75,346	1,147	130,410	206,903
MGC	128	2 × 1	10,555	137,141	100,159	89,924	825	227,662	318,411
DRA	128	2 × 2	19,133	229,387	162,117	152,516	2,011	264,272	418,799
MGC	128	2 × 2	21,069	275,875	202,221	181,672	839	458,802	641,313
DRA	128	2 × 4	38,226	457,347	323,257	304,136	6,481	525,108	835,725
MGC	128	2 × 4	42,097	550,323	403,465	362,448	867	914,202	1,277,517

Depth image processing portion of the DARPA IU benchmark – March 26, 1993

Key

1. Using dynamic register allocation (DRA) or modified Gnu C++ compiler (MGC).
2. Size of the image being processed. Since the program is independent of both the physical CAAPP size and the problem size, it is possible to try various combinations of both. While the same image file is used, only the selected part is processed.
3. Number of Tiles = Tile rows × tile columns.

Image Size	CAAPP Size	Number of Tiles
64	64 × 32	2
64	64 × 64	1
64	64 × 128	1
64	128 × 128	1
128	64 × 32	8
128	64 × 64	4
128	64 × 128	2
128	128 × 128	1

4. Number of instruction words sent to the Micro Controller. This is less than twice the number of abstract machine instructions executed.
5. Number of Micro Controller instructions executed.
6. Number of entries placed on the iFIFO. The iFIFO is a rate-buffer, implemented in hardware as a circular buffer, between the Micro Controller and the CAAPP. The value in this column is larger than the value in column 7 because of synchronizing instructions which are not sent on to the CAAPP.
7. Number of CAAPP instructions executed. This does not include idles.
8. Number of CAAPP idle (nop) instructions executed because the iFIFO was empty.
9. Number of CAAPP idle (nop) instructions executed when the iFIFO was not empty. These idles are due to waits for off-chip data transfer to complete and synchronization between the Micro Controller and the iFIFO. The idles due to data transfers are over 90% of this total.
10. This is the total time required for the CAAPP to complete the benchmark. It is the sum of columns 7, 8, and 9.

C Compiler Switches

Several new switch options are available for controlling the modified Gnu C++ compiler.

-mfor-interpreter Use the function IuaMc_pFIFO1 () to send instruction words to the Micro Controller. This switch is required when using the CAAPP simulator.

-mall-compatible Do not generate any compatibility checks between different planes used in an operation. This switch can be set after a program is debugged. Its use will improve Control Processor performance. Its use may improve CAAPP performance because more compiler optimizations may be successful. The compiler is unable to eliminate all compatibility checks because of lack of initializers on planes used as function arguments.

-mno-size-checks Do not generate any checks to see if the size of the *virtual* CAAPP changes. This switch may be set ONLY if all planes used in a program are the same size. Its use will lower Control Processor overhead. The compiler is unable to determine if all planes used in a program are the same size because of lack of initializers on planes used as function arguments and the possibility of separate compilation.

-fiua-dead-stores Eliminate data transfers from abstract machine registers to CAAPP off-chip memory that are not *needed*. This switch is *unsafe* in any program that uses aliases for any plane reference. Its use will improve CAAPP performance.

-mno-virtualization Generate the assembly language output to force all planes to be the size of the physical CAAPP upon which the program is executed. This switch is provided for the purposes of evaluation of the cost of *virtualization* and should not be used for other purposes. Its use will result in incorrect compiled code.

-miua-stats Print out statistics about the compilation of each function.

Contents

List of Figures