Code Generation for Streaming: an Access/Execute Mechanism†

MANUEL E. BENITEZ AND JACK W. DAVIDSON
Department of Computer Science
University of Virginia
Charlottesville, VA 22903, U. S. A

ABSTRACT

Access/execute architectures have several advantages over more traditional architectures. Because address generation and memory access are decoupled from operand use, memory latencies are tolerated better, there is more potential for concurrent operation, and it permits the use of specialized hardware to facilitate fast address generation. This paper describes the code generation and optimization algorithms that are used in an optimizing compiler for an architecture that contains explicit hardware support for the access/execute model of computation. Of particular interest is the novel approach that the compiler uses to detect recurrence relations in programs and to generate code for them. Because these relations are often used in problem domains that require significant computational resources, detecting and handling them can result in significant reductions in execution time. While the techniques discussed were originally targeted for one specific architecture, many of the techniques are applicable to commonly available microprocessors. The paper describes the algorithms as well as our experience with using them on a number of machines.

INTRODUCTION

A number of new architectures have appeared that have several common features. One feature shared by many new machines is the inclusion of several functional units that can operate in parallel. A common configuration is to have separate fixed-point and floating-point execution units although other specialized functional units, such as graphical processors or address-generation processors, are possible. A term that has been coined to denote this type of processor is "decoupled" to emphasize the separation of the units and their independent, yet cooperative operation [GOOD85, SMITH84]. Conceptually, the instruction stream can be viewed as being divided into two or more separate streams for the individual units. The IBM RISC System/6000 [OEHL90], Intel's i860 [INTE89], the Astronautics ZS-1 [SMIT87], PIPE [GOOD85], and WM [WULF88] are all architectures in this class.

The major advantage of such machines is the ability to exploit instruction-level parallelism by executing instructions for the individual units simultaneously. How this concurrent operation is extracted from an essentially sequential instruction stream varies from architecture to architecture, and it is beyond the scope of this paper to discuss these differences. However, regardless of the mechanisms used to fetch and issue instructions, the ability of the compiler to generate code that effectively exploits the multiple execution units is critical.

This paper discusses the algorithms used in an optimizing compiler for exploiting an execution unit specifically designed to handle "streams" [WULF90b]. Streams are structured data stored in memory with a known, fixed displacement between successive elements. Streaming can be viewed as a special case of the access/execute model of computation [GOOD85, SMIT84]. In this model, address generation is separated from consumption of the data. This allows two processors to be used concurrently and in concert to execute a task. One processor performs address generation and fetches the data, while the second processor performs all data processing calculations. With streaming, accesses to structured data are performed by dedicated hardware specially designed to take advantage of the data's storage regularity. A single 32-bit instruction contains all the information necessary to direct the hardware to read/write the entire data structure from/to memory.

The streaming concept is extremely powerful. For example, it naturally handles codes that contain recurrence relations. These relations occur frequently in solutions to problems that require significant computational resources to solve. Recurrences occur in algorithms involving solutions of partial differential equations, signal processing, imaging transformations, and graphical transformations. Furthermore, codes that contain recurrences are difficult and often impossible to vectorize [HENN90]. Consequently, the detection of recurrence relations in programs coded in a high-level programming language and the accompanying generation of efficient machine language instruction sequences can yield significant execution performance improvements. The detection of recurrence relations and the avoiding of possible hazards is non-trivial and difficult in the face of aliasing problems.

Streaming is also useful in other situations. It is useful for copying data from one area of memory to another. It can be used anytime that structured data is accessed in a regular and predictable fashion. For example, this often occurs when processing strings. Indeed, the optimizer generates stream instructions for the following Unix utilities: *cal*, *compact*, *od*, *sort*, *diff*, *nroff*, and *yacc*.

While the algorithms described were originally developed to

†This work was supported in part by the National Science Foundation under Grant CCR-8611653 and the Defense Advanced Research Agency under contract Number N00014-89-J1699.

Appeared in the
Proceedings of the Fourth International Conference on
Architectural Support for Programming Languages and Operating Systems

exploit an architecture that contains dedicated hardware support for streaming, it has been pleasant to discover that the algorithms work quite well on and provide performance enhancements for stock microprocessors such as the Motorola 68020, the Intel 386 and 486, and Motorola 88K. Section 4 contains measurements of the effectiveness of the optimizations on these machines. The algorithms would also be applicable to other popular microprocessors such as the Sun SPARC, MIPS R2000, Intel i860, and IBM RS/6000.

THE WM ARCHITECTURE

The WM architecture implements the concept of streaming. The main features of the WM architecture that are germane to the discussion of the code generation and optimization algorithms are briefly described below. Many details of the architecture, such as the various data types supported, instruction format, operating system support, and I/O, are not discussed. The interested reader can find these details in *The WM Computer Architectures: Principles of Operation* [WULF90a].

The Functional Units

The machine has four main functional units: an instruction fetch unit (IFU), an integer execution unit (IEU), a floating-point execution unit (FEU), and a vector execution unit (VEU). All four units operate concurrently. There is also a fifth unit that is dedicated to handling streaming instructions.

The Instruction Fetch Unit

The IFU is responsible for fetching and dispatching instructions to the appropriate execution units. Conceptually, the IFU fetches instructions sequentially and dispatches them to the appropriate execution unit where they are placed in first-in-first-out (FIFO) queues to await execution. Some instructions, notably branch instructions, are processed by the IFU itself. This allows the IFU to supply a steady stream of instructions to the execution units. Even though instructions are fetched and dispatched sequentially, it should be noted that depending on the relative speed of the execution units and the instruction mix, the actual execution of instructions can be out of order with respect to the original program text. Each execution unit operates at its peak speed. In this respect WM is similar to the IBM RS/6000 [OEHL90].

Control instructions, that is those that affect the program counter, and other instructions that require synchronization of the execution units (such as converting a floating-point value to an integer value) are executed by the IFU. In the case of unconditional branches, the IFU is able to update the program counter and continue fetching and dispatching instructions without delay. Thus, unconditional transfers of control have essentially zero cost. Conditional transfers of control are handled in the following way. Compare instructions are executed by the appropriate unit. These instructions generate a result that is enqueued into a condition code FIFO. There is one such FIFO for each execution unit. To execute a conditional branch instruction, the IFU dequeues a result from the appropriate condition code FIFO and performs the appropriate action. If the FIFO is empty, the IFU stalls waiting for an execution unit to enqueue the result of a compare. It is the responsibility of the compiler to ensure that exactly one instruction that generates a condition code is executed for each conditional jump executed. It is also the compiler's job to arrange the code so that the computation of the condition code occurs well before the result is needed. When this is done properly, conditional jumps, like unconditional jumps, essentially have zero cost.

The Scalar Execution Units

Both the IEU and FEU have 32 registers. For each unit, however, certain registers have special meaning. For example, for both units register 31 is defined to be zero. Writes to register 31 have no effect. Register 0 also has special meaning. For each unit, register 0 is actually two, first-in-first-out (FIFO) queues that are used to buffer data to and from memory. A load instruction only computes an address; the destination is implicitly the input FIFO. The FIFO holds the data until it is needed for some computation. Data is consumed from the input FIFO by reading register 0.

For stores, data is enqueued in the output FIFO (by writing to register 0), then when the address is computed (by a store instruction) the memory request is generated. Stores may also be performed by first generating the address followed by writing the data to be stored into register 0. The memory request to store the data is generated when both the data and the address are available. All simple load and store instructions (for both integer and floating-point data) are executed by the IEU.

The separation of address generation from consumption (access/execute), combined with the use of queues to buffer data to and from memory is described by Smith [SMITH84]. It has been used in the PIPE processor [GOOD85], and the ZS-1 [SMIT87]. The advantage is that, in concert with the compiler, it allows the processor to mask memory latency by issuing loads in advance of the data consumption. The result is a machine that is less sensitive to memory latency and cache misses. Figure 1 contains a block diagram of the architecture illustrating these features.

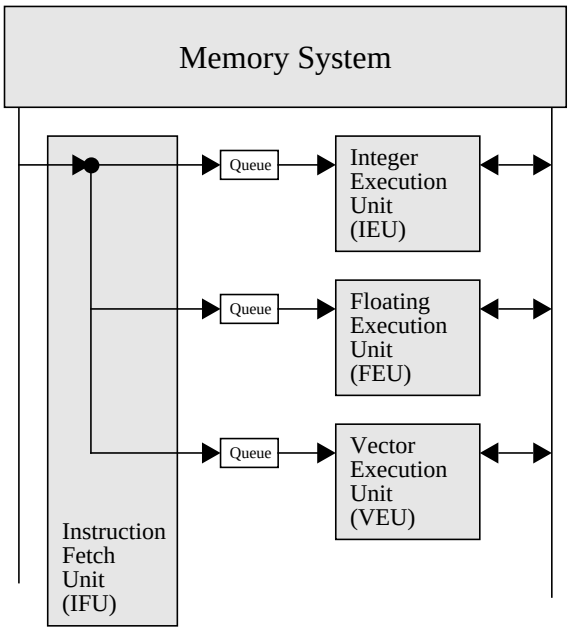

Figure 1. Block Diagram of Primary Architecture Components

The architecture has several other notable features. First, most instructions encode two operations in a single 32-bit word. Such instructions are of the form:

```
R0 := (R1 op1 R2) op2 R3
```

The two operations ***op1*** and ***op2*** are performed by a pair of pipelined arithmetic units connected as shown in Figure 2. The operation and operands enclosed in parentheses are referred to as the "inner" operator and operands, respectively. The inner operation is performed by ALU1, and the outer operator is

performed by ALU2.

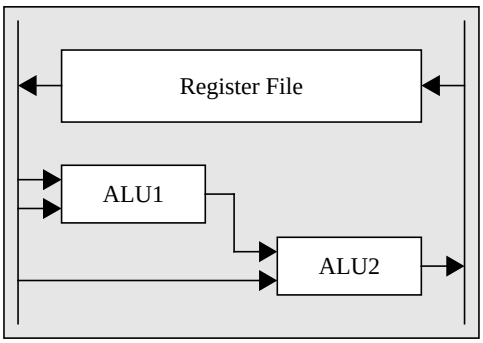

Figure 2. Pipeline Structure of Arithmetic Units

This structure induces the following data dependency rule:

> The result of an instruction is not available as an operand of the following instruction for *the same execution unit*. The value of an inner operand is specifically independent of the effect of the previous instruction.

It is interesting to note that this simple feature subsumes many of the specialized addressing modes and special operations found on many existing machines. For example, scaled addressing modes, commonly used for array operations, are coded as a single instruction using shift and add as ***op1*** and ***op2***. Auto-increment and auto-decrement addressing modes are also easily synthesized. A few machines include special instructions for performing a multiply and add [ATT88, OEHL90]. They are particularly useful in codes that perform transformations such as the fast fourier transformation and graphics transformations. This is easily and naturally handled by the two-operation, three-operand instructions.

The Vector Execution Unit

The architecture also supports vector operations. Briefly, these instructions are of the form:

```
R0 := (R1 op R2) if R3
```

Each vector register contains N components, where N is a parameter of the particular implementation. Using C-like operators and semantics to describe them, each vector instruction performs the computation:

```
forall (k = 0; k <= N; k++)
  R0[k] = R3[k] ? R1[k] op R2[k] : R0[k];
```

Conceptually the iterations of the loop are performed simultaneously by the vector execution unit (VEU).

Streaming

The instruction set contains special stream instructions (supported in hardware by specialized units) that provide fast reading and writing of structured data elements stored in memory with a known, fixed displacement between successive items. A single instruction can cause a stream of data to be read/written from/to either the IEU FIFOs, the FEU FIFOs, or the VEU.

Streaming can be viewed as an extension of the access/execute capabilities of the machine. The address generation and fetching of structured data is handled by separate, specialized units tailored for the job. These units, called stream control units (SCUs), are responsible for generating the sequence of addresses necessary to fetch/store the data and issuing the corresponding memory requests. For matrix calculations, where address generation and the fetching and storing of the array elements can be a substantial component of the code, the obvious benefit is that these tasks are now handled by a separate unit that operates concurrently with the execution unit processing the data.

In streaming mode, both register 0 and register 1 can be treated as input/output FIFOs. A stream instruction specifies four entities: the FIFO to read/write the data from/to, the base address, the count of the number of memory accesses to perform, and the stride or distance between successive elements. There are also a set of conditional jump instructions that can test whether or not a streaming operation is complete. The code below illustrates the use of streams to compute the dot product of two 32-bit floating-point vectors A and B of length N.

```
     r5 := N           -- initialize count
     r6 := A           -- load address of array A
     r7 := B           -- load address of array B
     f4 := f31         -- initialize sum to 0
     sin32f f0,r6,r5,4 -- stream A into FIFO f0
     sin32f f1,r7,r5,4 -- stream B into FIFO f1
L1:  f4 := (f0*f1)+f4  -- perform computation
     jNIf0 L1   -- jump on stream not exhausted
```

The above code is very efficient. First, the loop consists of only two instructions. The first instruction is executed by the FEU, while the second is executed by the IFU. All memory addresses are computed by the SCU. With a relatively simple hardware implementation, the code will produce the dot product in N clock cycles.

As will be shown later, streaming is useful in a variety of contexts. However, one of its most important uses is handling codes that contain recurrences. Such codes are difficult and sometimes impossible to vectorize. For these codes, streaming allows vector-like performance to be achieved. Of course, when vector code is possible, the compiler generates code that uses the vector unit. It is the compiler's responsibility to detect codes that have recurrences and to generate streaming code.

CODE GENERATION

An optimizing C compiler that supports streaming has been constructed for WM. The compiler is based on an portable optimizer that operates at the machine-level [BENI88]. Using the diagrammatic notation of Wulf [WULF75], Figure 3 shows the overall structure of the C compiler. Vertical columns within a box represent logical phases which operate serially. Columns that are divided horizontally into rows indicate that the subphases of the column may be executed in arbitrary order. The optimizer operates on register transfer lists ('RTLs'). RTLs describe the effect of machine instructions. They have the form of conventional expressions and assignments over the hardware's storage cells. For example, a WM instruction that performs a multiplication and addition would be expressed in RTL notation as:

```
r[3] = (r[4] * r[5]) + r[6];
```

Any particular RTL is machine specific, but the form of the RTL is machine independent. The optimizer uses RTLs because their machine-independent form permits it to optimize machine-specific code in a machine-independent way. Such machine-independent optimizations take the place of classical machine-specific case analysis and thus improve retargetability.

Compilers constructed using the optimizer use three pervasive strategies that lead to the generation of excellent code. The first strategy is that the front end generates naive but correct code for a

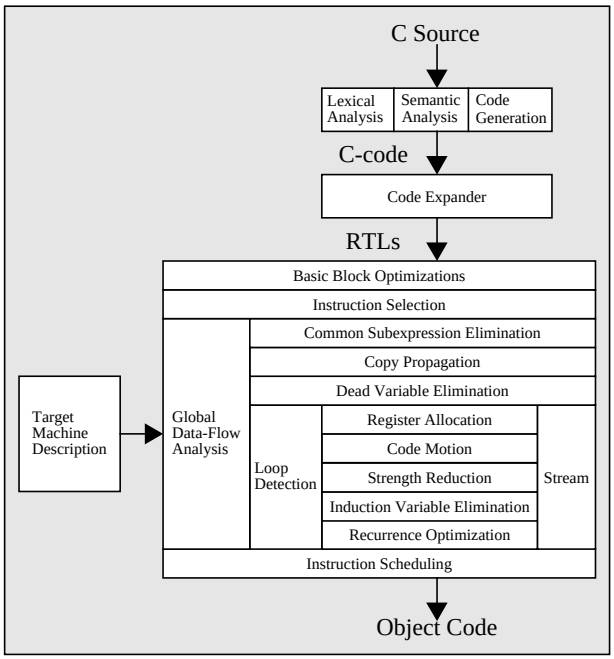

Figure 3. Schematic of the Optimizing Compiler

simple abstract machine. The code expander translates the abstract machine code into straightforward code for the target machine. The code is inefficient, but simple to produce. Both of these phases are concerned only with producing semantically correct code. Efficiency is *not* an issue. All code generation and optimization decisions are delayed until the target architecture information is available. The second strategy is that all optimizations are performed on object code (RTLs). The guiding principle is that more complete and thorough optimization is possible by operating on object code. The above two strategies are similar to the approach taken in the PL.8 compiler for the IBM 801 [AUSL82, RADI82]. The third strategy is that the optimizer uses the same representation for all phases of optimization. This allows optimization phases to be reinvoked at any time. This largely eliminates phase ordering problems common to many optimizers, and it simplifies many optimization algorithms as they need only be concerned with the transformation at hand and not the interactions with other optimizations.

The optimizer has proven itself to be quite portable, yet capable of generating very high-quality code. The C front end and optimizer have been retargeted to over ten machines. These machines include several CISC-class machines such as the VAX-11, Intel 80386, and Motorola 68020 and several RISC-class machines such as the Motorola 88K, Intergraph Clipper, and Sun SPARC. The optimizer generates very high-quality code. For example, on the Sun-3/280 using the C component of the SPEC benchmark suite, the compiler achieves a SPECratio of 4.3 (see SPEC Tables in Appendix I). The native C compiler, using the highest optimization level available, obtains a SPEC rating of 4.0.

Recurrence Detection and Optimization Algorithm

Knuth [KNUT73] defines a recurrence relation as:

> A rule which defines each element of a sequence in terms of the preceding elements.

Recurrence relations appear in the solutions to a large number of compute-bound problems. Consequently, recognizing recurrences and generating code that computes the relation more efficiently offers two obvious benefits. First, solutions can be obtained more quickly, second, and perhaps more importantly, larger problem instances can be tackled.

Because they are important, a number of benchmark programs contain recurrences. Consider the following C code:

```
for (i = 2; i < n; i++)
   x[i] = z[i]* (y[i] - x[i-1]);
```

This is the fifth Livermore loop, which is a tri-diagonal elimination below the diagonal. It contains a recurrence since `x[i]` is defined in terms of `x[i-1]`. This loop is used to describe the algorithms developed to handle recurrences and streaming.

Many of the transformations necessary to perform recurrences optimization are done by the optimizer in the course of other routine optimizations. For example, loop detection and code motion must be performed first. Figure 4 contains the resulting WM code after these optimizations are performed.

The recurrence detection algorithm builds partitions that hold information about the memory references being performed in the loop. The information is represented in a vector of the form:

`(lno, acc, iv`$^{\texttt{dir}}$`, cee, dee, roffset)`

where

`lno`	-	line number where the memory reference occurred
`acc`	-	whether it is a read or write reference (R/W)
`iv`	-	the induction variable
`dir`	-	the direction; + if the `iv` is increasing, - otherwise
`cee`	-	c in the formula iv = c*i+d†
`dee`	-	d in the formula iv = c*i+d†
`roffset`	-	dee - base offset

In the following step-by-step description of the algorithm the sections set in `Courier` font describe the application of the step to the code in Figure 4, and discuss any special conditions not illustrated by the example.

Recurrence Detection and Optimization Algorithm

Step 1. Divide the memory references that are made in the loop into partitions that reference disjoint sections of memory. For a particular memory reference, if the proper group is unknown, add the memory reference to each group. Also record whether the memory reference is a read or a write, and where the reference occurs.

```
There are three partitions:
   X = {(14,r,,,,), (16,w,,,,)}
   Y = {(13,r,,,,)}
   Z = {(10,r,,,,)}

For memory references made via pointers, it is often
the case that it is impossible to tell what regions
of memory may be accessed. In this case, the
reference will be added to each partition as it
potentially touches each.
```

† See *Compilers, Principles, Techniques and Tools* [AHO86] for a complete description of induction variable detection.

```
1.                      r31 := (2 >= r23)      -- compare n to 2
2.                      r22 := 2               -- initialize i
3.          JumpIT      L16                    -- jump if n < 2
4.          llh         r21 := _x
5.          sll         r21 := _x              -- compute address of x
6.          llh         r24 := _z
7.          sll         r24 := _z              -- compute address of z
8.          llh         r25 := _y
9.          sll         r25 := _y              -- compute address of y
10. L20:    l64f        r31 := (r22<<3) + r24  -- generate memory request for z[i]
11.                     r20 := (r22-1) << 3    -- compute offset of ith-1 array element
12.         double      f20 := f0              -- dequeue z[i]
13.         l64f        r31 := (r22<<3) + r25  -- generate memory request for y[i]
14.         l64f        r31 := (r20) + r21     -- generate memory request for x[i-1]
15.         double      f0 := (f0-f0) * f20    -- compute new x[i] and enqueue result
16.         s64f        r31 := (r22<<3) + r21  -- generate memory request to store x[i]
17.                     r22 := (r22) + 1       -- increment i
18.                     r31 := (r23) <= r22    -- compare i to n
19.         JumpIF      L20                    -- jump if i <= n
20. L16:
```

Figure 4. Unoptimized WM code for the 5th Livermore loop.

Step 2. For each memory reference in the loop, determine the induction variable, the direction of the loop (i.e., whether the induction variable is increasing or decreasing), the 'cee' value, and the 'dee' value.

The induction variable is i for all memory references in the loop, which is held in register 22. The direction of the loop is positive. The 'cee' value is 8 for each memory reference. For the X partition read, the 'dee' value is _x-8 since the reference is to x[i-1] (doubles are eight bytes). The X partition write 'dee' value is _x. For the Y and Z partition, the 'dee' values are _z and _y respectively. The partitions are now:

X = {(14,r,r22$^+$,8,_x-8,), (16,w,r22$^+$,8,_x,)}

Y = {(13,r,r22$^+$,8,_y,)}

Z = {(10,r,r22$^+$,8,_z,)}

Step 3. For each partition do:

Step a. If all references in the partition do not have the same induction variable or they do not have the same 'cee' value, mark the partition as unsafe.

For partition X, all references use the induction variable r22 (i.e., i), and have a 'cee' value of 8. Partitions Y and Z contain only one reference so they trivially satisfy the condition. Generally, a pointer reference will not have an induction variable.

Step b. Determine the constants that all 'dee' values in the partition have in common. Call this the base offset. For all references in a partition, determine the relative offset between the reference and the induction variable. If the relative offset is not evenly divisible by the 'cee' value, mark the partition as unsafe.

The base offset for the X partition is _x, for the Y partition it is _y, and for the Z partition it is _z. For partition X, the relative offset between the read memory reference and the induction variable is -8. For the write memory reference the relative offset is 0. Both are evenly divisible by the 'cee' value which is 8. For the Y and Z partition, the relative offsets are 0. The partitions are now:

X = {(14,r,r22$^+$,8,_x-8,-8), (16,w,r22$^+$,8,_x,0)}

Y = {(13,r,r22$^+$,8,_y,0)}

Z = {(10,r,r22$^+$,8,_z,0)}

Step4. For all partitions still marked safe and which contain reads and writes do:

Step a. Identify read/write pairs, that is memory references where a read fetches the value written on a previous iteration. For each read/write pair, calculate the distance between the references. This is the absolute difference of the relative offsets for the references. The maximum difference determines the number of registers needed to handle the recurrence. This distance is divided by the stride of the loop.

For the X partition, the maximum distance between read/write pairs is 8. The stride is 8, thus we need two registers to handle the recurrence. In general, you need one more register than the degree of the recurrence.

Step b. Logically, before the write, code is generated to copy the value to a register. Similarly, the subsequent reads are deleted and replace with register references. The partition is modified to reflect that the read is no longer performed in the loop.

This step is machine-dependent. For WM, the following actions are taken. At line 15, the write to FIFO f0 is modified so that the value is retained in f22. Before the write at line 16, f22 is written to FIFO f0. The load at line 14 is deleted, and the FIFO read reference in line 15 is replaced with a reference to f23. f23 holds x[i-1]. The partitions are:

X = {(16,w,r22$^+$,8,_x,0)}

Y = {(13,r,r22$^+$,8,_y,0)}

Z = {(10,r,r22$^+$,8,_z,0)}

Step c. At the top of the loop, generate code that copies the register that held the ith value into the register that holds the ith-1 value. The number of copies required is equal to the degree of

```
1.                      r31 := (2 >= r23)      -- compare n to 2
2.                      r22 := 2               -- initialize i
3.          JumpIT      L16                    -- jump if n < 2
4.          llh         r21 := _x
5.          sll         r21 := _x              -- compute address of x
6.          llh         r24 := _z
7.          sll         r24 := _z              -- compute address of z
8.          llh         r25 := _y
9.          sll         r25 := _y              -- compute address of y
9a.         l64f        r31 := (r31+8) + r21   -- generate memory request for x[1]
9b.         double      f22 := f0              -- dequeue x[1]
10. L20:    double      f23 := f22             -- copy x[i-1] to x[i]
11.         l64f        r31 := (r22<<3) + r24  -- generate memory request for z[i]
12.                     r20 := (r22-1) << 3    -- compute offset of ith-1 array element
13.         double      f20 := f0              -- dequeue z[i]
14.         l64f        r31 := (r22<<3) + r25  -- generate memory request for y[i]
15.         double      f22 := (f0-f23) * f20  -- compute new x[i]
16.         double      f0 := f22              -- enqueue x[i]
17.         s64f        r31 := (r22<<3) + r21  -- generate memory request to store x[i]
18.                     r22 := (r22) + 1       -- increment i
19.                     r31 := (r23) <= r22    -- compare i to n
20.         JumpIF      L20                    -- jump if i <= n
21. L16:
```

Figure 5. WM code for the 5th Livermore loop with recurrences optimized.

---

the recurrence.

The degree of this recurrence is 1. Thus, at the top of the loop (after line 10), register f22 is copied to f23. If the order of the recurrence is greater than 1, it is important to emit the copies in the proper order.

Step d. Build a loop pre-header (if one does not exist) to perform the initial reads.

An initial load of x[1] is emitted. It is loaded into register f22.

The resulting code is shown in Figure 5. Notice that the major difference between the code in Figure 4 and Figure 5 is that there are now only three memory references in the loop instead of four. The value of `x[i]` is retained in register f22 and becomes `x[i-1]` on the next iteration of the loop. For this loop, the number of memory references that will be executed is reduced by one quarter.

After performing the recurrence transformations, the optimizer invokes other phases to catch any optimizations that may be possible on the transformed code. For example, the copy propagate optimization phase would delete the register-to-register copy at line 10 replacing the use of register f23 at line 15 with register f22.

The algorithm applies to other machines as well. In fact, the algorithm is largely machine-independent. The routine that replaces memory references with register references is machine-specific. It consists of approximately 30 to 50 lines of C code. Figure 6 contains the assembly code the C compiler retargeted to the Motorola 68020 generated for the 5th Livermore loop. Again, it is worthwhile noting that the structure of the compiler simplifies the machine-independent implementation of the optimizations. First, the machine-independent form of the RTLs allows the optimizations to be implemented without regard to the target architecture. Second, the compiler can reinvoke other phases of the compiler at any time. In the Motorola 68020 example, the instruction selection phase determined that auto-increment addressing modes could be used to fetch the memory operands at the top of the loop.

Several C compilers on various machines were examined to see if they optimized codes containing recurrences. The C compilers were on:

1. a Sun-3 running SunOS 4.0 (Motorola 68020-based),
2. a HP9000 series 345 running HPUX 7.0 (Motorola 68030-based),
3. a Motorola 88K Delta running System V/88 (88K-based and with the Green Hills compiler Version 1.8.4),
4. a DecStation 3100 running Ultrix V2.1 Rev 14. (MIPS R3000-based)
5. a Sparcstation 1^+ running SunOS 4.1 (Sun Sparc-based)
6. an IBM RS6000/530 running AIX Version 3.1

Only the IBM C compiler optimized codes that contain recurrences. It is possible that the FORTRAN compilers on these machines optimize recurrences, although most of them use the same back end for both C and FORTRAN.

To evaluate the effect that recurrence relation detection and optimization has on execution times, the 5th Livermore loop was compiled and run on five machines. The array size was set to 100,000. For each machine, the code was generated with and without recurrence detection enabled. Table I shows the percentage improvement in execution time when recurrence optimizations are enabled. The best case improvement would occur if the time to perform the memory references dominated all other operations performed in the loop. In this case, elimination of one of the four memory references would produce a speed up of approximately 25

```
        moveq     #2,d1                     -- initialize i
        movl      a7@(n.),d0                -- load n
        cmpl      d0,d1                     -- compare i to n
        jge       L44                       -- jump if loop should not be executed
        fmoved    (8 + _x),fp0              -- load x[1]
        lea       (16 + _z),a0              -- address of z[2]
        lea       (16 + _y),a1              -- address of z[2]
        lea       (16 + _x),a2              -- address of x[2]
L48:    fmoved    a1@+,fp1                  -- load y[i]
        fsubx     fp0,fp1                   -- subtract x[i-1]
        fmoved    a0@+,fp0                  -- load z[i]
        fmulx     fp1,fp0                   -- calculate x[i]
        fmoved    fp0,a2@+                  -- store x[i]
        addql     #1,d1                     -- increment i
        cmpl      d0,d1                     -- compare to n
        jlt       L48                       -- branch if i < n
L44:
```

Figure 6. Motorola 68020 code for the 5th Livermore loop with recurrences optimized.

---

percent. The observed improvements range from 6 to 19 percent.

Machine	Percent Improvement in Execution Time
Sun 3/280	19
HP 9000/345	12
VAX 8600	6
Motorola 88100	7
WM	18

Table I. Effect of Recurrence Optimization on Execution Time

Streaming

After recurrences have been detected and optimized, the compiler attempts to exploit any possibilities for using stream operations. The algorithm makes use of the memory partition information collected in the previous algorithm.

Streaming Optimization Algorithm

Step 1. Determine the number of iterations through the loop. Call this the *loop_count*. If it is impossible to determine, set *loop_count* to ∞. If the number of iterations is determined to be three or fewer, do not use streams.

If the number of iterations is low, setting up the stream instructions would result in code that executes slower than the code without streaming. For the example, *loop_count* is *n-1*. The current partition information is:

X = {(17,w,r22+,8,_x,0)}

Y = {(14,r,r22+,8,_y,0)}

Z = {(11,r,r22+,8,_z,0)}

Step 2. For all partitions marked safe do:
For each memory reference in the partition do:

Step a. Ensure that no memory recurrences remain in the loop (e.g. subsequent reads of something written in the previous iterations). If memory recurrences still exist, do not stream.

For this example, all memory recurrences have been eliminated. Some recurrences may not be completely eliminated because there may not be enough registers to hold the intermediate results. If memory recurrences still exist, it is not possible to guarantee that the value written on one iteration will be the value read on the next iteration.

Step b. Calculate the stride which is 'cee' × loop increment.

Cee is 8 and the loop increment is 1, resulting in a stride of 8.

Step c. Determine if the memory reference is executed every time through the loop. If true, then use streaming.

This is determined by making sure that the block that contains a memory reference dominates all blocks that branch to the loop header. This condition is necessary as streaming will fetch every element of the structure being referenced. The example loop satisfies this requirement.

Step d. Determine the number of times the memory reference is executed.

If the memory reference dominates the loop exit, then it is *loop_count*, otherwise it is *loop_count* - 1. For our example, each memory reference is executed *loop_count* or *n-1* times.

Step e. If it is a read reference, use the stream-in operation, otherwise use stream-out. Allocate appropriate FIFO register. If one is not available, do not stream.

For the Livermore loop, the write reference in the X partition is allocated output FIFO f0, the read reference in the Y partition is allocated input FIFO f0, and the read reference in the Z partition is allocated input FIFO f1.

Step f. Generate code in the loop preheader to test whether the loop should be executed and to jump around the loop if it should not be executed.

This is the code at lines 1 and 11 of Figure 7.

Step g. Add appropriate stream in and stream out instructions in loop header.

These are the instructions at lines 13, 15, and 17.

Step h. Change loads and stores to use FIFO registers.

```
1.                      r31 := (r21-1) <= 0    -- check if loop should be executed
2.          llh         r20 := _x
3.          sll         r20 := _x              -- compute address of x
4.          l64f        r31 := (r31+8) + r20   -- generate memory request for x[1]
5.          llh         r22 := _z
6.          sll         r22 := _z              -- compute address of z
7.          llh         r23 := _y
8.          sll         r23 := _y              -- compute address of y
9.                      r24 := (r21) - 1       -- compute number of items to stream
10.         double      f22 := f0              -- dequeue x[1]
11.         jumpIT      L16                    -- jump if loop should not be executed
12.                     r19 := (16) + r22      -- compute address of z[2]
13.         SinD        f1,r19,r24,8           -- stream in z[2], z[3],...,z[n]
14.                     r19 := (16) + r23      -- compute address of y[2]
15.         SinD        f0,r19,r24,8           -- stream in y[2], y[3],...,y[n]
16.                     r19 := (16) + r20      -- compute address of x[2]
17.         SoutD       f0,r19,r24,8           -- stream out x[2], x[3],..,x[n]
18. L20:    double      f22 := (f0-f22) * f1   -- compute x[i]
19.         double      f0 := (f31) + f22      -- store x[i]
20.         Jnif1       L20                    -- jump if stream count not zero
21.  L16:
```

Figure 7. WM code with stream instructions.

The load instructions for *y* and *z* are deleted. These are the instructions at lines 11, 13, and 14 of Figure 5.

Step i. If *loop_count* is ∞, add stream stop instructions at all loop exits. If *loop_count* is known and finite, delete loop test and replace with a stream instruction.

The example does not use infinite streams, thus the step results in lines 19 and 20 in Figure 5 being deleted and replaced with line 20 in Figure 7.

Step j. If the value of the induction variable is dead on loop exit, delete the increment of the induction variable.

In this example, the induction variable is dead, so line 18 of Figure 5 is deleted.

Step 3. Perform strength reduction on the modified loop.

For details of this algorithm see any text on compiler construction.

With streaming and strength reduction, the loop now consists of only three instructions. No address computations are performed in the loop. These as well as the accompanying memory requests are handled by the stream control units that operate concurrently with the other execution units. The resulting code's execution rate is determined by how fast the data can be read and written. Because addresses generated by the SCUs exhibit a regular pattern, it may be possible that the interface between the SCUs and the memory unit can be optimized to take advantage of this characteristic (e.g. burst mode transfers).

It is clear that streaming is applicable to codes containing recurrences and array manipulations. However, it was somewhat of a pleasant surprise that streaming appeared in a variety of programs. A number of Unix utilities were compiled and were found to use streaming. Some examples are: *cal*, *compact*, *od*, *sort*, *diff*, *nroff*, and *yacc*. The uses included copying strings and structures, searching a decoding tree, searching a data structure for a specific item, and initializing an array.

Program	Percent Reduction in Cycles Executed
banner	5
bubblesort	18
cal	17
dhrystone	39
dot-product	43
iir	13
quicksort	1
sieve	18
whetstone	3

Table II. Execution Performance improvements by streaming.

It also appears that streaming can provide good execution performance improvements. Table II gives the percent reduction in cycles executed for programs compiled with and without streaming optimizations enabled. The programs were executed on a simulator capable of determining exact cycle counts (including memory delays). The largest improvement was exhibited by dot product (43 percent), and the smallest gain was produced by quicksort (1 percent).

SUMMARY

This paper has described an optimizer designed to exploit the access/execute capabilities of an architecture under construction. The architecture is unique in that it contains hardware support for efficiently accessing streams of data. A stream is a set of data items stored in memory with a fixed, known displacement between successive elements. Streams can be used in several contexts. One of the most important, however, is in codes that contain recurrences. Loops containing recurrences are the computational hot-spot in codes for many important problems. Furthermore, recurrences are difficult and usually impossible to vectorize.

The paper describes two complementary algorithms that are used to exploit streaming. The first algorithm detects and optimizes

recurrences. Surprisingly, this optimization is included in few available C compilers. Of six commercially available C compilers examined, only one optimizes recurrences. The algorithm is interesting because it is machine-independent, yet it is applied to machine-dependent code. For the machines examined, the optimization significantly reduces the time to execute loops containing loop-carried dependences.

The second algorithm builds on the results of the recurrence detection and optimization algorithm to handle streams. The ability of the optimizer to detect situations where streaming can profitably be used is key to exploiting the power of the architecture. For example, when streams are detected and special stream instructions are used, the cost of computing addresses and issuing memory requests is off-loaded to a separate, dedicated processor. This increases the amount of concurrency obtained when executing the loop. The algorithm appears to perform well. Besides detecting the obvious situations for using streams (like vector and array manipulations), the algorithm finds opportunities for streaming in codes where such possibilities are not immediately obvious. Furthermore, it appears that the algorithms may be applicable to generating code for vector units.

ACKNOWLEDGEMENTS

The WM architecture was designed by William A. Wulf of the University of Virginia. Charles Hitchcock of Dartmouth participated in the initial design and is responsible for a number of the machine's unique features. The WM team at the University consists of Jim Aylor, Barry Johnson, Anita Jones, Peter Kester, Sunil Pamidi, and Max Salinas. The support of the Defense Advanced Research Agency and the National Science Foundation made this work possible.

REFERENCES

[ATT88] *WE DSP32 Digital Signal Processor Information Manual*, AT&T Documentation Management Organization, 1988.

[BENI88] M. E. Benitez and J. W. Davidson, A Portable Global Optimizer and Linker, *Proceedings of the SIGPLAN Notices '88 Symposium on Programming Language Design and Implementation*, Atlanta, GA, June 1988, 329-338.

[AHO86] A. V. Aho, R. Sethi and J. D. Ullman, *Compilers Principles, Techniques, and Tools*, Addison-Wesley, Reading, MA, 1986.

[AUSL82] M. Auslander and M. Hopkins, An Overview of the PL.8 Compiler, *Proceedings of the SIGPLAN Notices '82 Symposium on Compiler Construction*, Boston, MA, June 1982, 22-31.

[GOOD85] J. R. Goodman, J. Hsieh, K. Kiou, A. R. Pleszkun, P. B. Schechter and H. C. Young, PIPE: A VLSI Decoupled Architecture, *Proceedings of the 12th International Symposium on Computer Architecture*, Boston, MA, June 1985, 20-27.

[HENN90] J. L. Hennessy and D. A. Patterson, *Computer Architecture A Quantitative Approach*, Morgan Kaufmann, San Mateo, CA, 1990.

[INTE89] *i860 64-Bit Microprocessor Hardware Reference Manual*, Intel Corporation, Santa Clara, CA, 1989.

[KNUT73] D. E. Knuth, *Fundamental Algorithms*, Addison-Wesley, Reading, MA, 1973.

[OEHL90] R. R. Oehler and R. D. Groves, IBM RISC System/ 6000 Processor Architecture, *IBM Journal of Research and Development* **34**,1 (January 1990), 23-36.

[RADI82] G. Radin, The 801 Minicomputer, *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, CA, March 1982, 39-47.

[SMIT84] J. E. Smith, Decoupled Access/Execute Architectures, *ACM Transactions on Computer Systems* **2**,4 (November 1984), 289-308.

[SMIT87] J. E. Smith, G. E. Dermer, B. D. Vanderwarn, S. D. Klinger, C. M. Roszewski, D. L. Fowler, K. R. Scidmore and J. P. Laudon, The ZS-1 Central Processor, *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, CA, October 1987, 199- 204.

[WULF75] W. Wulf, R. K. Johnsson, C. B. Weinstock, S. O. Hobbs and C. M. Geschke, *The Design of an Optimizing Compiler*, American Elsevier, New York, NY, 1975.

[WULF88] W. A. Wulf, The WM Computer Architecture, *Computer Architecture News* **16**,1 (March 1988), 70-84.

[WULF90a] W. A. Wulf, The WM Computer Architectures: Principles of Operation, TR90-02, University of Virginia, January 1990.

[WULF90b] W. A. Wulf and C. Hitchcock, The WM Family of Computer Architectures, TR90-05, University of Virginia, March 1990.

APPENDIX I - C SPEC BENCHMARK RESULTS

Configuration Information

7/90, SPEC License No. 268
Sun 3/280
1 processor Motorola 68020, 25 MHz clock speed
1 FPU Motorola 68881; 20 MHz clock speed
16 MB main memory
2 Fujitsu Eagle II disk drives, each 575MB capacity (formatted)
1 ZyLogics 451 disk controller
Operating Systems Sun OS, Version 4.0.3
All measurements run in Multi-User State
File System parameters: 8K block size, 1 K fragment size, optimized for time.
Small background load during measurements (less than 0.2)

SPEC Benchmark Release 1.0	SPEC Reference Time Elapsed Time (seconds)	System Under Test (Sample) Elapsed Time (seconds)	SPECratio
001.gcc	1482	336	4.4
008.espresso	2266	644	3.5
022.li	6206	1401	4.4
023.eqntott	1101	296	3.7
Geometric Means	2188.7	547.3	4.0

Table III. SPEC measurements for C compiler distributed with Sun OS 4.0.1

SPEC Benchmark Release 1.0	SPEC Reference Time Elapsed Time (seconds)	System Under Test (Sample) Elapsed Time (seconds)	SPECratio
001.gcc	1482	317	4.7
008.espresso	2266	611	3.7
022.li	6206	1252	5.0
023.eqntott	1101	279	4.0
Geometric Means	2188.7	510.0	4.3

Table IV. SPEC measurements for vpcc/vpo C compiler for Motorola 68020 version 7/90.