The PowerPC 604 RISC Microprocessor

The PowerPC 604 RISC microprocessor uses out-of-order and speculative execution techniques to extract instruction-level parallelism. Its nonblocking execution pipelines, fast branch misprediction recovery, and decoupled memory queues support speculative execution.

The 604 microprocessor is the third member of the PowerPC family being developed jointly by Apple, IBM, and Motorola. Developed for use in desktop personal computers, workstations, and servers, this 32-bit implementation works with the software and bus in the PowerPC 601 and 603 microprocessors. While keeping the system interface compatible with the 601 microprocessor, we improved upon it by incorporating a phase-locked loop and an IEEE-Std 1149.1 boundary-scan (JTAG) interface on chip. In addition, an advanced machine organization delivers one and a half to two times the 601's integer performance.

Performance strategy
Processor performance depends on three factors: the number of instructions in a task, the number of cycles a processor takes to complete the task, and the processor's frequency. Our architecture, which we optimized to produce compact code while adhering to the reduced instruction set computer (RISC) philosophy, addresses the first factor. The high instruction execution rate and clock frequency addresses the other two factors. The 604 provides deep pipelines, multiple execution units, register renaming, branch prediction, speculative execution, and serialization.

Six-stage superscalar pipeline. As shown in Figure 1, this deep pipeline enables the 604 to achieve its 100-MHz design. The stages are:

- **Fetch.** This stage translates an instruction fetch address and accesses the cache for up to four instructions.
- **Decode.** Instruction decoding determines needed resources, such as source operands and an execution unit.
- **Dispatch.** When the resources are available, dispatch sends instructions to a reservation station in the appropriate execution unit. A reservation station permits an instruction to be dispatched before all of its operands are available. As they become available, the reservation station forwards operands to the execution units. Dispatch can send up to four instructions in program order (in-order dispatch) to four of six execution units: two single-cycle integers, a multicycle integer, a load/store, a floating point, and a branch.
- **Issue/execute.** In each execution unit, this stage issues one instruction from its reservation station and executes it to produce results. The instructions can execute out of program order (out-of-order execution) across the six execution units as well as within an execution unit that has an out-of-order issue reservation station. Table 1 lists the latency and throughput of the execution stages.
Branch instructions

- Fetch
- Predict
- Dispatch
- Validate
- Complete

Integer instructions

- Fetch
- Decode
- Dispatch
- Execute
- Complete
- Write back

Load/store instructions

- Fetch
- Decode
- Dispatch
- Addr
- Calc
- Cache
- Align
- Complete
- Write back

Floating-point instructions

- Fetch
- Decode
- Dispatch
- Multiply
- Add
- Rnd/norm
- Complete
- Write back

Table 1. Pipeline description.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Latency</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>Most integer</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Integer multiply (32x32)</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>Integer multiply (others)</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Integer divide</td>
<td>20</td>
<td>19</td>
</tr>
<tr>
<td>Integer load</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Floating-point load</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Store</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Floating-point multiply-add</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Single-precision floating-point divide</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>Double-precision floating-point divide</td>
<td>31</td>
<td>31</td>
</tr>
</tbody>
</table>

Although some designs use even deeper pipelines to achieve higher clock frequencies than the 604 does, we felt that such a design point does not suit today's personal computers. It relies too heavily on one of, or a combination of, a very large on-chip cache, a wide data bus, or a fast memory system to deliver its performance. It would be less than competitive in today's cost-sensitive personal computer market.

Precise interrupts and register renaming. Most programmers expect a pipelined processor to behave as a non-pipelined processor, in which one instruction goes through the fetch to write-back stages before the next one begins. A processor meets that expectation if it supports precise interrupts, in which it stops at the first instruction that should not be processed. When it stops (to process an interrupt), the processor's state reflects the results of executing all instructions prior to the interrupt-causing instruction and none of the later instructions, including the interrupt-causing instruction. This is not a trivial problem to solve in multiple, out-of-order execution pipelines. An earlier instruction executing after a later instruction can change the processor's state to make later instruction processing illegal. Sohi gives a general overview of the design issues and solutions.

The 604 uses a variant of the reorder buffer described by Smith and Pleszkun to implement precise interrupts. The 16-entry reorder buffer keeps track of instruction order as well as the state of the instructions. The dispatch stage assigns each instruction a reorder buffer entry as it is dispatched. When the instruction finishes execution, the execution unit records the instruction's execution status in the assigned reorder buffer entry. Since the reorder buffer is managed as a first-in-first-out queue, its examining order matches the instruction flow sequence. To enforce in-order completion, all prior instructions in the reorder buffer must complete before an instruction can be considered for completion. The reorder buffer examines four entries every cycle to allow completion of up to four instructions per cycle.

Unlike Smith and Pleszkun's reorder buffer, the 604's reorder buffer does not store instruction results. Temporary buffers hold them until the instructions that generated them complete. At that time, the write-back stage copies the results to the architectural registers. The 604 renames registers to achieve this, instead of writing results directly to specified registers, they are written to rename buffers and later copied to specified registers. Since instructions can execute out of order, their results can also be produced and written out of order into the rename buffers. The results are, however, copied from the buffers to the specified registers in program order. Register renaming minimizes architectural resource dependencies, namely the output-dependency (or write-after-write hazard) and antidependency (or write-after-read hazard), that would otherwise limit opportunities for out-of-order execution.

Figure 2 (next page) depicts the format of a rename buffer entry. The 604 contains a 12-entry rename buffer for the general-purpose registers (GPRs) that are used for 32-bit integer operations. The 604 allocates a GPR rename buffer entry upon dispatch of an instruction that modifies a GPR. The dispatch stage writes a destination register number of the
instruction to the Reg num field, sets a Rename valid bit, and clears the Result valid bit. When the instruction executes, the execution unit writes its result to the Result field and sets the Result valid bit. After the instruction completes, the write-back stage copies its result from the rename buffer entry to the GPR specified by the Reg num field, freeing the entry for reallocation. For a load-with-update instruction that modifies two GPRs, one for load data and another for address, the 604 allocates two rename buffer entries.

Register renaming complicates the process of locating the source operands for an instruction since they can also reside in rename buffers. In dispatching an integer instruction, the dispatch stage searches its source operands simultaneously from the GPR file and its rename buffer. If a source operand has not been renamed, the processor uses the value read from the GPR file. If a rename exists (indicated by an entry with the Rename valid set and its Reg num field matching the source register number), the Result in the rename buffer is used. It is, however, possible that the result is not yet valid because the instruction that produces the GPR has not yet executed. The dispatch stage still dispatches the instruction since the operand will be supplied by the reservation station when the result is produced. The dispatched instruction contains the rename buffer entry identifier in place of the operand. The GPR file and its rename buffer can use eight read ports for source operands to support dispatching of four integer instructions each cycle.

The 604 also uses a rename buffer for floating-point registers (FPRs) and one more for the condition register (CR). The FPR rename buffer has eight 90-bit-wide entries to hold a double-precision result with its data type and exception status. The FPR file and its rename buffer access three read ports for dispatching one floating-point instruction per cycle. In addition to compare instructions, most integer and floating-point instructions can also generate negative, positive, zero, and overflow condition results. One of the eight fields in the 32-bit CR stores these 4-bit condition results. The 604 treats each field as a 4-bit register and applies register renaming using an eight-entry CR rename buffer.

Branch prediction and speculative execution. Because today’s application software contain a high percentage of branch instructions, correctly predicting the outcome of these instructions is crucial to keeping the multiple instruction pipelines flowing and for achieving two to three times the execution rate of scalar processors. The 604 uses dynamic branch prediction in the fetch, decode, and dispatch stages to predict as well as correct branch instructions early.

The 604’s speculative execution strategy complements its branch prediction mechanisms. The strategy is to fetch and execute beyond two unresolved branch instructions. The results of these speculatively executed instructions reside in rename buffers and in other temporary registers. If the prediction is correct, the write-back stage copies the results of speculatively executed instructions to the specified registers after the instructions complete.

Upon detection of a branch misprediction, the 604 takes quick action to recover in one cycle. It selectively cancels the instructions that belong in the mispredicted path from the reservation stations, execution units, and memory queues. It also discards their results from the temporary buffers. In addition, the processor resumes its previous state to start executing from the correct path even before the mispredicted branch and its earlier instructions have completed. Since the 604 detects a branch misprediction many cycles before the branch instruction completes, its fast recovery scheme helps to maintain performance of those applications with high data cache miss rates and whose branches are difficult to predict.

Serialization. A serialization mechanism delays execution of certain instructions that would otherwise be expensive to execute speculatively in the 604’s multiple-pipeline, out-of-order execution design. This mechanism delays infrequently used instructions until they can safely execute while permitting later instructions to execute. Some examples are the move to and from special-purpose register instructions, the extended arithmetic instructions that read the carry bit, and the instructions that directly operate on the CR, which the PowerPC architecture provides for calculating complex branch conditions. This mechanism also controls store instructions since it is difficult to undo stores.

The dispatch stage sends a serialized instruction to the proper execution unit with an indication that it should not be executed. When all prior instructions have completed and updated their results to the architectural state, the completion stage allows the serialized instruction to execute. Once the serialized instruction is dispatched, dispatch continues to dispatch the following instructions so they can execute before the serialized instruction. When the serialized instruction is completed, the later instructions also complete upon finishing execution. This minimizes the penalty of serialized instructions.

Machine organization

Figure 3 shows the fetch address generation logic. The fetch stage selects an address from the addresses generated in the different pipeline stages each cycle. Since an address generated in a later stage belongs to an earlier instruction, its selection precedes an address from an earlier stage.

The completion stage detects exception conditions and generates an exception handler address. This stage also
updates the program counter (PC) with the target address of a taken branch instruction, or advances it by the number of instructions being completed. The branch execute stage may correct the instruction fetch with the branch target address if the branch is mispredicted as not taken and with the sequential address if the branch is mispredicted as taken. The dispatch and decode stages may change the fetch address with either the target or sequential address of a branch instruction being predicted. There are two copies of the target and sequential address registers in the decode, dispatch, and execute stages, since there can be up to two branch instructions in each stage. The completion stage also has two target registers to handle up to two finished branch instructions.

If the fetch address hits in the branch target address cache (BTAC), the target address becomes the fetch address. Otherwise, the instruction fetch continues sequentially. The 64-entry, fully associative BTAC holds the target addresses of the branches that are predicted to be taken. If a branch is predicted as not taken for its next encounter, the branch execute stage removes it from the BTAC. The BTAC is accessed with the fetch address, and not with a branch instruction address, providing a zero-cycle fetch penalty for taken branches. Although there may be multiple branch instructions in the four instructions being fetched, the BTAC provides the target address of the first-predicted taken branch instruction.

**Instruction decode and dispatch.** The pipeline decodes four instructions every cycle to determine exception conditions, as well as the resources needed by the instructions. The resources include the execution unit, source operands, and destination registers. Decoding the instructions before the dispatch stage simplifies the dispatch logic without using predecoded bits in the instruction cache. Predicting branch instructions in the first two entries of the decode buffer minimizes the performance penalty of adding the decode stage.

When the decode stage detects an unconditional branch that was not in the BTAC, it corrects the instruction fetch to the target address of the branch. It also predicts conditional branches with the execution history found in the branch history table (BHT). Each entry of the 512-entry BHT denotes one of the four history states: strong not taken, weak not taken, weak taken, and strong taken. The table updates the history state with the actual outcome of the branch that is mapped to the entry. To simplify the design, each entry in the BHT maps to every 512th instruction address. This allows the BHT to be accessed with the fetch address and to return the four entries mapped to the four instructions starting with the fetch address.

Not all conditional branch instructions use the BHT. The architecture provides a count register (CTR) value as a branch condition to support loops in programs. Only the conditional branch instructions that do not depend on the CTR value use, as well as update, the BHT. Those that do depend on the CTR are predicted, based on the value of the shadow CTR. The shadow CTR has a future state of the CTR that is

---

**Figure 3. Instruction fetch address generation logic.**
updated by speculatively executed move-to-CTR or branch-and-decrement instructions. This prediction scheme is very effective on branches that control loop iteration.

The dispatch buffer sends up to four instructions to four of the six execution units each cycle. As space allows, more instructions advance from the decode stage into the four-entry dispatch buffer. The 604 places only a few restrictions on dispatch to enable a high-speed implementation. They are

1) **One instruction per execution unit.** Since each execution unit can start only one instruction per cycle and an instruction can bypass the reservation station if the execution unit is available, dispatching one instruction per unit simplifies the logic without imposing an undue performance penalty. Two identical single-cycle integer units handle the more frequent instructions.

2) **Resources available.** Each instruction needs a reorder buffer entry, a reservation station entry in the appropriate execution unit, and reorder buffer entries to hold its results. Available resources depend on the state of the instructions previously dispatched as well as those currently being dispatched.

3) **Stop dispatch after branch.** Instructions following a branch instruction are not dispatched in the same cycle as the branch is dispatched. This restriction simplifies saving the processor state, which allows immediate canceling of speculatively executed instructions that follow predicted branches.

4) **In-order dispatch.** Dispatching instructions in order results in only a small cost in performance and greatly simplifies resource allocation and dispatch logic. Out-of-order execution is introduced with six independent execution pipelines and out-of-order issue reservation stations to achieve performance comparable to an out-of-order dispatch design.

**Reservation stations and result forwarding.** A two-entry reservation station on every execution unit allows instructions to be dispatched before obtaining all of their operands. Without a reservation station, an instruction cannot be dispatched until all of its source operands become available, either in the register file or in its rename buffer. Without reservation stations, the 604's in-order dispatch design would be more complex, since it would have to detect data dependencies and would frequently stall. The reservation stations in the three integer units can issue instructions out of order to allow a later instruction to bypass an earlier stalled instruction. The branch, load/store, and floating-point unit reservation stations may only issue instructions in order.

Each execution unit provides one result bus for each type of result it produces. For instance, the multicycle integer unit has one result bus for the GPR and another for the CR data types. Figure 4 shows the four GPR result buses and the reservation stations and GPR rename buffer that are connected to them. Each GPR reservation station entry monitors all four GPR result buses for any mispredictions for any A or B operands, denoted as A op and B op in the figure. When an execution unit returns a result and the associated GPR rename buffer entry identifier, the reservation station compares the identifier against those in its entries. When a match is found, it forwards the result to the waiting instruction. For returning the update address of a load-with-update instruction while executing one load instruction per cycle, the load/store unit shares the result bus of the less frequently used multicycle integer unit.

It is interesting to note why the 604 uses a reorder buffer, rename buffers, and reservation stations to provide the same functions that a DRIS (deferred-scheduling, register-renaming instruction shelf) in the Metaword architecture provides. A DRIS entry consists of instruction status fields that a reorder buffer entry would have, source operand fields that a reservation station entry would have, and destination fields that a rename buffer entry would have. (The 604's reservation station entry uses a separate source operand to store either an immediate or a copy of the source operand. Although the DRIS figure in the Metaword article shows only the ID field to indicate the DRIS entry with the source operand, it is likely that another field is needed to store an immediate operand.)

Two disadvantages of the DRIS had we used it in the 604 design are

- **Scheduling overhead.** The DRIS instruction scheduling is more complicated than the 604's dedicated reservation stations since the next instruction for an execution unit must be the first "ready" instruction of the "right" type in all DRIS entries.

- **Single result type.** DRIS supports renaming of only one register type, whereas the 604 needs three. Say that more than one DRIS is used, as described in Popescu et al. to support separate integer and floating-point registers. One of them would have to house all instructions to provide precise interrupts while not being able to provide register renaming. An alternative is to design one DRIS to accommodate the largest register type.

**Execution units.** The branch execution unit can hold two branch instructions in its reservation station and two more finished branch instructions. It serves to validate branch predictions made in earlier stages, and also verifies that the predicted target matches the actual target address. If a misprediction is detected, the branch execution unit redirects the fetch to the correct address and starts the branch misprediction recovery.

The 604 has a three-stage complex integer unit (CFX) to execute integer multiply, divide, and all move to and from special-purpose register instructions. The CFX can sustain one multiply instruction per cycle for 32x16-bit and those 32x32-bit multiplies whose B operand is representable as a
17-bit signed integer. It can sustain one multiply per two cycles for larger 32x32-bit operands. The GFX also uses the multiply pipeline stages to execute a divide instruction in 20 cycles. The 604's two simple integer units execute all other integer instructions in one cycle.

The three-stage floating-point unit can sustain one double-precision multiply-add per cycle, one single-precision divide every 18 cycles, or one double-precision divide every 31 cycles. It complies fully with the IEEE-Std 754 floating-point arithmetic standard. The 604 provides hardware support for denormalization, exceptions, and three graphics instructions. It also provides a non-IEEE mode for graphics support. The non-IEEE mode converts a denormalized result to zero to avoid prenormalization in subsequent operations.

**Instruction completion and write back.** After an instruction executes, the execution unit copies results to its rename buffer entries and the execution status to its reorder buffer entry. Among other things, the execution status indicates whether the instruction finished execution without an exception. Of the four reorder buffer entries examined every cycle, up to four instructions that finished without an exception complete in program order.

Other than the in-order completion necessary to support precise exceptions, the 604 imposes only a few additional restrictions on instruction completion. They are:

1) **Stop before a store instruction.** Since a store data operand is read from the register file in the completion stage, a store instruction cannot complete if its store operand is still in the rename buffer. Stopping completion before a store instruction allows the store operand to be written to the register file, even if it is produced by an instruction currently being completed.

2) **Stop after a taken branch instruction.** Since a taken branch instruction sets the program counter to its target address, it is speed critical to advance the program counter from the new target address by the number of instructions completed after the taken branch in one cycle. Stopping completion after a taken branch instruction avoids this logic altogether.

To minimize effects of long execution latency on in-order completion, the completion stage overlaps with the last execution cycle for those instructions with multicycle execution stage. These include the multiply, divide, store, load miss, and execution serialized instructions. A store instruction completes as soon as it is translated without an exception. Similarly, a load instruction that misses in the data cache completes upon translation without an exception. Since the reservation stations can forward the load data when it is available to the dependent instructions, the load miss can safely be completed.

Most superscalar designs impose additional restrictions due to a limited number of ports to register files. For instance, four write ports would be required to complete up to four instructions if each one can update one register. The 604 GPR file would require eight write ports to complete four load-with-update instructions per cycle. The 604 avoids this problem by decoupling instruction completion from register file updates using the write-back stage. Instructions complete without regard to the number of registers they update. The completion stage updates their results if ports are available; if not, the write-back stage updates them. The rename buffer entries function as temporary buffers for those instructions that are not completed and as write-back buffers for those that are. All three GPR, FPR, and CR rename buffers contain two read ports for write back. Correspondingly, the three register files have two write ports for write back.

**Memory operations**

High-speed superscalar processors require a greater memory bandwidth to sustain their performance. The 604 meets the increased demand with large on-chip caches, non-blocking memory operations, and a high-bandwidth system interface. The 604 takes advantage of the weakly ordered memory model, to which the PowerPC architecture subscribes, to offer efficient memory operations. Although loads and stores that hit in the data cache can bypass earlier loads and stores, program order memory access can be enforced with instructions provided for this purpose.

**Load/store unit.** Figure 5 (next page) shows a block diagram of the load/store unit and the memory queues. This unit has a two-cycle execution stage. It calculates the memory address and translates that address with a 128-entry, two-

---

**Figure 4. GPR result buses, reservation stations, and rename buffer.**
way set-associative translation lookaside buffer (TLB) in the first cycle. The second cycle processes loads making a speculative cache access and aligns bytes when the access hits in the cache. The pipelined execution stage executes one load or store instruction per cycle.

In the first half cycle, the load/store unit calculates a load instruction's memory address, denoted as EA in the figure. It translates the real address, denoted as RA in the figure, and the data cache access begins in the second half cycle. If the access hits in the cache, the unit aligns the data and forwards it to the rename buffer and the execution units in the second cycle. If the access misses, the unit places the instruction and its real address in the four-entry load queue. When a load miss completes, it accesses the cache a second time. If the load is still a miss, the unit moves it to the load miss register while reloading the missing cache line. This permits a second load miss to access the cache and to initiate the second cache line reload before the first is brought in.

The unit calculates the memory address of a store and translates it in the first cycle. It does not write the data to the cache, however, until after the store instruction completes. The unit places the instruction and its real address in the six-entry store queue. Since the data cache is not accessed in the second cycle, it is available for an earlier store from the store queue (if necessary) or load miss from the load queue (if necessary).

When a store instruction completes, the load/store unit marks it completed in the store queue so that instruction completion can continue without waiting for storage to the cache or memory. If the store hits in the cache, the unit writes it to the cache and removes it from the store queue. If the store is a miss, the unit will bypass it in the store queue to allow later stores to take place while cache reloading proceeds. Multiple store misses can be bypassed in the store queue.

Figure 6 shows the store queue structure. Four pointers identify the state of the store instructions in the circular store queue. When a store has finished execution (or successful translation), the load/store unit places it in the finished state. When it completes, the finish pointer advances to place it in the committed state. When it is committed to cache or memory, the completion pointer advances to place it in the committed state. If the store hits in the cache, advancing the commit pointer removes it from the queue. If the store is a miss, the commit pointer does not advance until the missing cache line is reloaded and the store is written to the cache. While the cache line is being reloaded, the next store indicated by the completion pointer can access the cache. If this store hits in the cache, the unit removes it from the queue. If it misses, another cache line reload begins.

**Caches.** The 604 provides separate instruction and data caches to allow simultaneous instruction and data accesses every cycle. Both 16-Kbyte caches provide byte parity protection and a four-way set-associative organization with 32-byte lines. They are indexed with physical addresses, have physical tags, and make use of the least recently used replacement algorithm.

The instruction cache provides a 16-byte interface to the fetch unit to support the four-instruction dispatch design. This nonblocking cache allows subsequent instructions to be fetched while a prior cache line is being reloaded. This design is particularly beneficial if the missing cache line belongs in a mispredicted path, since it allows the correct instructions to be accessed immediately after a branch misprediction recovery. The instruction cache also provides streaming, a mechanism to forward instructions as they are received from off-chip cache or memory.

The instruction cache does not maintain coherency; instead, the architecture provides a set of instructions for software to manage coherency. In particular, the instruction cache block invalidate (ICBI) instruction causes all copies of the addressed cache line to be invalidated in the system. The ICBI generates an invalidation request, to which all coherent caches must comply.

The data cache contains a 64-bit data interface to the load/store unit for data access, MMU for tablewalking, and bus interface unit (BIU) for cache line reloading and snooping. (The architecture specifies an algorithm to traverse page table entries that define the virtual-to-physical memory mappings. "Tablewalk" refers to a hardware implementation of the algorithm.) The data cache's two copy-back buffers support nonblocking cache operations. A copy-back buffer holds a dirty (modified) cache line that is being replaced or that hits on a snoop request. The data cache moves an entire cache line into a copy-back buffer in one cycle to minimize the
cycles the cache is unavailable.

The MESI (modified, exclusive, shared, invalid) coherency protocol defined in the *PowerPC 60x Processor Interface Specification* keeps the data cache coherent. A duplicated data tag array supports two-cycle snoop response with minimum performance impact to the normal cache operations.

To further support nonblocking cache operations, we've extended the MESI coherency protocol with one more state. As illustrated in a simplified state diagram in Figure 7, the protocol assigns the new state, Allocated, to the block selected to hold the missing cache line. All necessary information for this particular miss, including the address and set number, remains in the memory queue and later completes the cache line reload. The cache line becomes either shared or exclusive, based on the coherency response.

The software can individually disable, invalidate, or lock both instruction and data caches. While a cache is disabled, all accesses bypass it and directly access the off-chip cache or memory. While a cache is locked, it is accessible but its contents cannot be replaced. All cache misses, in this case, are accessed from off chip as cache-inhibited accesses. Coherency is, however, maintained even when a cache is locked. The data cache supports the cache touch instructions, which initiate reloading of the specified cache line if it is not in the cache. These instructions can effectively shorten cache miss rates and latency.

**Bus interface unit and memory queues.** The 604's BIU implements the *PowerPC 60X Processor Interface Specification* to ensure bus compatibility with the 601 and 603 microprocessors. A split transaction mode allows the address bus to operate independently of the data bus, freeing the data bus during memory wait states. To support the split transaction mode, the BIU uses the address and data buses only during what are known as address tenure and data tenure cycles. The BIU also provides a pipelined mode, in which up to three address tenures can be outstanding before data for the first address is received. If permitted, the BIU will complete one or more write transactions between the address and data tenures of a read transaction. Byte parity protects its 32-bit address and 64-bit data buses.

Figure 8 (next page) shows the address and data queues that implement split and pipelined transaction modes. Four types of memory queues support the four types of operations: line fill, write, copy back, and cache control. For a line-fill operation, the line-fill address from either the instruction or data cache remains in the memory address queue until the address can be sent out in an address tenure. After the address tenure, the address transfers to the line-fill address queue. This releases the address bus for other transactions in the split transaction mode. As each double word for the cache line returns, it moves to the line-fill buffer and also forwards to the load/store unit.

During a write operation the address stays in the memory address queue, and the data in the write buffer, until both the address and data can move out in a write transaction. The size of a write transaction can vary from 1 to 8 bytes to handle nondouble-word aligned writes. Similarly during a copy-back operation, the address remains in the copy-back address queue, and the data in the copy-back buffer, until both the address and data transfer in a 32-byte burst write transaction. For a cache control instruction or a store to a shared cache line, the address stays in the cache control address queue until an address-only transaction broadcasts the cache control command. Since all address queues in the 604 are considered part of the coherent memory system, the BIU checks them against data cache and snoop addresses to ensure data consistency and maintain MESI coherency protocol.

**System support features**

The 604 provides several features to support robust system design such as instruction and data address breakpoints and single-step and branch instruction tracing facilities for software debugging. It also provides performance-monitoring functions for the system to profile software performance.
Table 2. The 604 physical characteristics.

<table>
<thead>
<tr>
<th>Item</th>
<th>Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Technology</td>
<td>0.5-μm CMOS, 4 metal layers</td>
</tr>
<tr>
<td>Die size</td>
<td>196 mm², 12.4x15.8 mm</td>
</tr>
<tr>
<td>Transistor count</td>
<td>3.6 million</td>
</tr>
<tr>
<td>Cache size</td>
<td>16-Kbyte I-cache and D-cache</td>
</tr>
<tr>
<td>Voltage</td>
<td>3.3V, 5V I/O tolerant</td>
</tr>
<tr>
<td>Power dissipation</td>
<td>Less than 10W at 100 MHz</td>
</tr>
<tr>
<td>Signal I/Os</td>
<td>CMOS/TTL compatible</td>
</tr>
<tr>
<td>Package</td>
<td>304-pin CQFP</td>
</tr>
</tbody>
</table>

without additional hardware. These functions can determine many key performance parameters, such as instruction execution rate, branch prediction rate, cache hit rates, and average cache miss latency.

The 604 design follows the level-sensitive scan design methodology to provide high test coverage. As required by LSSD rules, every storage element, except in arrays, connects to a scan chain that starts with a chip input pin and ends on a chip output pin. During test mode, storage elements in a scan chain behave as a shift register that can also capture inputs to exercise a sequential digital network in a combinational manner. The 604's common on-chip processor (COP) provides many functions to control and observe the storage elements. Some of the functions useful for chip and system debugging include setting instruction or data address breakpoints, single stepping, running Xcycles, and reading and writing system memory locations as well as any storage element within the processor. The COP functions are implemented as an extension to the IEEE-1149.1 specification, and are controlled entirely through that interface.

System designers can configure the 604 processor operating frequency as one, one-and-a-half, two, or three times the system bus frequency. The on-chip phase-locked loop generates the necessary processor clocks from the bus clock. The 604 also provides a nap mode, which clocks only external interrupt detection logic and the phase-locked loop. It enters nap mode under software control and exits from the mode upon detecting an interrupt. The 604 can still service snoop requests if the system asserts the RUN pin to run the clocks while in the nap mode. We estimate nap mode power consumption at less than 0.4 watts.

Table 2 lists some of key physical characteristics of the 604. Figure 9 shows the 604 die photo.

**References**


S. Peter Song is a senior engineer in the Systems Technology and Architecture Division of IBM. He led the definition of the 604 microarchitecture and later designed the speculative execution, completion, and exception control logic. Song holds BS, MS, and PhD degrees in electrical and computer engineering from the University of Texas at Austin. He is a member of the IEEE Computer Society, Eta Kappa Nu, and Tau Beta Pi.

Marvin Denman is a principal staff engineer in the RISC Microprocessor Division of Motorola, Inc. He has contributed to the definition of the 604 microarchitecture and later designed the fetch and branch-processing logic. Denman holds a BS degree in computer science from Texas A&M University and an MS in electrical engineering from the University of Texas at Austin. He is a member of the IEEE Computer Society.

Figure 9. Die photo of the PowerPC 604.

Joe Chang's biography, photograph, and address appear on p. 51 of this issue.

Direct questions concerning this article to S. Peter Song, Somerset Design Center, 11400 Burnet Road, M/S 973, Austin, TX 78758; spsong@ibmoto.com.

Reader Interest Survey
Indicate your interest in this article by circling the appropriate number on the Reader Service Card.

Low 154       Medium 155       High 156
Thus in all these cases the Romans did what all wise princes ought to do; namely, not only to look to all present troubles, but also to those in the future, against which they provided with the utmost prudence.

—NICCOLO MACHIAVELLI, The Prince

The Alpha AXP™ architecture grew out of a small task force chartered in 1988 to explore ways to preserve the VAX/VMS™ customer base through the 1990s. This group eventually came to the conclusion that a new RISC architecture would be needed before the turn of the century, primarily because 32-bit architectures will run out of address bits. Once we made the decision to pursue a new architecture, we shaped it to do much more than just observe the OpenVMS VAX™ customer base.

This article discusses the architecture from a number of points of view. First, we draw the distinction between architecture and implementation, then state the overriding architectural goals. A number of key architectural decisions are derived directly from these goals. The key decisions are then discussed, to distinguish the Alpha AXP architecture from others. The remaining sections discuss the architecture in more detail, from data and instruction formats through the detailed instruction set, and relate the designed-in future growth of the architecture. A short glossary is included for some of the key technical terms used in this article.

From the beginning of the Alpha AXP design, we distinguished the architecture from the implementations, following the distinction made by the IBM System/360 architects [1]:

COMPUTER ARCHITECTURE is defined as the attributes and behavior of a computer as seen by a machine language programmer. This definition includes the instruction set, instruction formats, operation codes, addressing modes, and all registers and memory locations that may be directly manipulated by a machine language programmer.

IMPLEMENTATION is defined as the actual hardware structure, logic design, and data-path organization of a particular embodiment of the architecture.

Thus, the architecture is a document [8] that describes the behavior of all possible implementations; an implementation is typically a single computer chip. The architecture and software written to the architecture are intended to last several decades, while individual implementations will have much shorter lifetimes. The architecture must therefore carefully describe the behavior that a machine language programmer sees, but must not describe the means by which a par-
ticular implementation achieves that behavior.

A similar approach has been used with good success in specifying the PDP-11™ and VAX™ families of computers. An alternate approach is to design and build a fast RISC chip, then wait to see if it is successful in the marketplace. If so, successor implementations are forced to reproduce accidents of the initial design, or to introduce slight software incompatibilities. This approach works, but with varying success.

Architectural Goals

When we started the detailed design of the Alpha AXP architecture, we had a small list of goals:

- High performance
- Longevity
- Run OpenVMS AXP™ and Unix™ (DEC OSF/1 AXP™) operating systems
- Easy migration from VAX and MIPS™ architecture customer bases

The key design decisions of the architecture were driven directly from these goals.

In considering performance and longevity, we set a 15–25 year design horizon, and tried to avoid any design elements that we thought could become limitations during this time. In current architectures, a primary limitation is the 32-bit memory address. We thus adopted a full 64-bit architecture, with a minimal number of 32-bit operations for backward compatibility.

We also considered how implementation performance should scale over 25 years. During the past 25 years, computers have become about 1,000 times faster. So we focused our design decisions on allowing Alpha AXP system implementations to become 1,000 times faster over the coming 25 years. We thought it would be reasonable for raw clock rates to improve by a factor of 10 over that time, and that two more factors of 10 would have to come from other design dimensions.

If the clock cannot be made faster, then more work must be done per clock tick. We therefore designed the Alpha AXP architecture to encourage multiple instruction issue implementations that will eventually sustain about 10 new instructions starting every clock cycle. This focus on aggressive multiple instruction issue distinguishes the Alpha AXP architecture from many other RISC architectures.

The remaining factor of 10 will come from multiple processors—a single system containing perhaps 10 processors and sharing memory. We therefore designed a multiprocessor memory model and matching instructions from the very beginning. This early focus on multiple processors also distinguishes the Alpha AXP architecture from many other RISC architectures, which try to add the proper primitives later.

To run OpenVMS AXP and Unix—and now Microsoft Windows NT™—operating systems we adopted an idea from a previous Digital Equipment Corporation RISC design called PRISM [5]. We placed the underpinnings for interrupt delivery and return, exceptions, context switching, memory management, and error handling in a set of privileged software subroutines called PALcode. These subroutines have controlled entry points, run with interrupts turned off, and have access to real hardware (implementation) registers. By having different sets of PALcode for different operating systems, neither the hardware nor the operating system is burdened with a bad interface match, and the architecture itself is not biased toward a particular computing style.

To run existing VAX [2] and MIPS [7] binary images, we adopted the idea of binary translation, as described in a companion article [9]. The combination of PALcode and binary translation gave us the luxury of designing the Alpha AXP architecture from a "clean sheet of paper"—other than the fundamental integer and floating-point data-types, there are no specific VAX or MIPS features carried directly into the Alpha AXP instruction-set architecture for compatibility reasons.

Key Design Decisions

RISC

The Alpha AXP architecture is a traditional RISC load-store architecture—all data is moved between registers and memory without computation, and all computation is done between values in registers. Little-endian byte addressing and VAX/IEEE floating-point are carried over from the VAX and MIPS architecture customer bases.¹ We assumed that most implementations would pipeline instructions (i.e., they would start execution of a second, third, etc., instruction before the execution of a first instruction completes). We assumed the implementation latency of many operations would be important. Latency is the number of cycles a program must wait to use the result of a preceding instruction. We assumed the vast majority of memory operands would be aligned. An aligned operand of size 2N bytes has a byte address with N low-order zeros. Other memory operands are termed unaligned.

Full 64-bit Design

The Alpha AXP architecture uses a linear 64-bit virtual address space. Registers, addresses, integers, floating-point numbers, and character strings, are all operated on as full 64-bit quantities. There is no address segmentation.

Register File

In choosing the register file design, we considered both a single combined register file and split integer and floating-point register files. We chose a split register file to support aggressive multiple issue. A combined file is somewhat more flexible, especially for programs that are heavily skewed toward integer-only or floating-point-only computation. A combined file also makes it easier to pass a mixture of integer and floating-point subroutine parameters in registers. However, split files allow graceful two-chip implementations,

¹The little-endian bias is very slight—both big and little-endian Alpha AXP systems and software are in fact being built.
smaller integer-only implementations, and need fewer read/write ports per file to sustain a given amount of multiple instruction issue.

We also considered whether each file should contain 32 or 64 registers. We chose 32, largely because (i) 32 in each is enough to support at least 8-way multiple issue, and (ii) two valuable instruction bits are better used to make a 16-bit displacement field in memory-format instructions. More registers might seem better, but excess registers consume chip area and access time, save/restore speed across subroutines and context switches, and instruction bits that might be put to better use. Compilers can deliver substantial performance gains when given 32 registers instead of 16, but there is no clear evidence of similar gains when going to 64 registers. Demand for registers will likely grow slowly over time, but there are a number of implementation techniques such a short latency pipelines and register renaming that should keep up.

Multiple Instruction Issue

Most of the design is driven by the need for aggressive multiple instruction issue. We wanted to avoid any mechanism that would hinder such implementations. We therefore tried to avoid all special or hidden processor resources. Thus, the Alpha AXP architecture has no condition codes, no global exception enables, no multiplier-quotient or string registers, no branch delay slots, no suppressed instructions or skips, no precise arithmetic exceptions, and no single-byte writes to memory. All of these features, found in some other RISC architectures, have the effect of hindering multiple instruction issue, or hindering pipelining of multiple instances of the same instruction (e.g., a dedicated string register makes it difficult to have three unrelated string operations in the pipeline at once).

Here is a simple example of the cost of special or hidden processor resources. Consider a dual-issue implementation with a four-cycle-deep pipeline. At the beginning of each cycle, up to six prior instructions are partially executed and two more are about to be issued. Six prior instructions can have six pending writes to result registers, plus six sets of side effects on special or hidden processor resources. The next two instructions can specify a total of four operand registers, two more result registers and two more sets of side effects on special or hidden resources.

The decision to issue zero, one, or two of the next instructions involves 36 simple comparisons of pairs of register numbers and 12 complex comparisons of sets of side effects. The number of such comparisons increases as a function of the issue width, the pipeline depth, and the number of special or hidden processor resources. The complexity of these comparisons can limit the clock rate. Since the register-number comparisons are unavoidable, we tried to limit special or hidden processor resources.

The Alpha AXP architecture has no branch delay slots. The branch delay slots found in some other RISC architectures require exactly one following instruction to be executed after a conditional branch. This was perhaps a good idea for overlapping branch latency in a single-issue chip with a 1-cycle instruction cache circa 1988, but it does not scale well to a 1995 4-way issue chip with a 2-cycle instruction cache. Instead of one instruction, up to 8 instructions would be needed in the delay slot. Branch delay slots also introduce a restart problem if the instruction in the delay slot faults—one restart program counter is needed for the delay slot and another one for the actual branch target.

There are no suppressed instructions in the Alpha AXP architecture, whereby the execution of one instruction conditionally suppresses a following one. Suppressed (or skipped) instructions are found in some other RISC architectures. The suppression bit(s) represent a nonreplicated hidden state, so it is difficult to multi-issue more than one potential suppressor. If an interrupt is taken between a suppressor and suppressor, or if the suppressor takes a restartable exception (e.g., page fault), the correct version of the suppression state must be saved and restored. There are also definitional problems with this approach: For example, are exceptions ever reported for actually-suppressed instructions? What happens if the suppressed instruction suppresses a third instruction?

The Alpha AXP architecture has no byte load or store instructions, and no implicit unaligned accesses. There are also no partial-register writes. The byte load/store instructions and unaligned accesses found in some other RISC architectures can be a performance bottleneck because they require an extra byte shifter in the speed-critical load and store paths, and they force a hard choice in fast cache design. The partial-register writes found in some other RISC architectures can also be a performance bottleneck because they require masking and shifting in the fundamental operation of accessing a register.

Our previous experience included a MIPS implementation project that found the shifter for the Load-left/Load-right instructions to be a direct cycle-time bottleneck. Also, the VAX 8700 implementation (circa 1986) had removed the byte shifter in the load/store hardware in favor of a faster microcycle, with 2 cycles for a byte load and 6 cycles for an unaligned 32-bit access. This achieved a net performance gain. Prior experience encouraged us to avoid byte load/store.

An additional problem with byte stores is that an implementer may easily choose only 2 of the 3 features:

- Fast writeback cache
- Single-bit error correction (ECC)
- Byte stores

Byte stores are straightforward in simple byte-parity writethrough cache implementations. But a byte store to a fast ECC writeback cache involves:

1. Reading an entire cache word
2. Checking the ECC bits and correcting any single-bit error

Except for the expensive design of 4 or 5 ECC bits for every 8 bits of data.
3. Modifying the byte
4. Calculating the new ECC bits
5. Writing the entire cache word

This read-modify-write sequence requires hidden sequencer hardware and hidden state to hold the cache word temporarily. The sequencer tends to slow down ordinary full-cacheword stores. The need for byte stores tends to ripple throughout the memory subsystem design, making each piece a little more complicated and a little slower. With the nonreplicated hidden state, it is difficult to issue another byte store until the first one finishes. Finally, the existence of a byte store instruction has led to programs and library routines for other RISCs with single-byte move and compare loops. By processing 8 bytes at a time, string manipulation on Alpha AXP implementations is up to 8 times faster.

Instead of including byte load store, we followed the RISC philosophy of exposing hidden computation as a sequence of many simple, fast instructions. In the Alpha AXP architecture, a byte load is done as an explicit load/modify/store sequence; a byte store as an explicit load/modify/store sequence. We tuned the instruction set to keep these sequences short. The instructions in these sequences can be intermixed, scheduled, and multi-issued with other computation, just like the rest of the instructions in the architecture.

The Alpha AXP architecture has no precise arithmetic exceptions. Reporting an arithmetic exception (i.e., overflow, underflow) precisely means that instructions subsequent to the one causing the exception must not be executed. This is straightforward in a slow implementation running a single instruction to completion before starting the next one. It becomes substantially more difficult to do quickly in a situation such as pipelined 4-way issue implementation. There are standard techniques available for delivering precise exceptions while running quickly (checking exponents, suppressing register writes, exception silos and backout), but these techniques consume substantial design time and can cost some performance. They appear not to scale well with wider multiple issue or faster clocks.

Exceptional cases are just that—exceptional, or rare, events. Based partly on customer requests, we decided to emphasize the performance of normal operations, at the expense of exceptional cases. Rather than an implicit exception ordering between every pair of instructions, we adopted the Cray-1 model of arithmetic exceptions [4]—you hear about it eventually—plus an explicit Trap Barrier (TRAPB) instruction that can be used to make exception reporting as precise as desired. We also documented a code-generation design that needs one trap barrier per branch (at most) to give precise reporting. Using TRAPB instructions in the first Alpha AXP implementation costs 3% to 25% performance in real floating-point programs and under 1% in integer programs, but gives perhaps a 10% cycle-time improvement.

In contrast to arithmetic exceptions, memory management exceptions, such as page faults, are reported precisely. This is not as much of a burden on implementers as precise arithmetic exceptions would be, and lack of precise memory management faults would be a severe burden on software writers.

Shared-Memory Multiprocessing

The Alpha AXP architecture’s shared-memory multiprocessing model is an integral part of the design, not the add-on found in some other RISC architectures.

The underlying primitive for multiprocessor-safe updating of a shared memory location is a sequence of RISC instructions: Load-locked, in-register modify, Store-conditional, test. If this sequence completes with no interrupts, no exceptions, and no interfering write from another processor, then the Store-conditional stores the modified result and the test indicates success—an atomic update was in fact performed.

If anything goes wrong, the Store-conditional does not store a result, and the test indicates failure. The program must then retry the sequence until it succeeds. We chose this primitive sequence (quite similar to the MIPS R4000 chip design [7]) because it can be implemented in a way that scales up with processor performance: in the absence of an interfering write, the entire sequence can be done in an on-chip writeback cache, and hundreds of chips can do noninterfering sequences simultaneously. The sequence can also be used to achieve byte granularity of writes in shared memory [8].

The Alpha AXP architecture has no strict multiprocessor read/write ordering, whereby the sequence of reads and writes issued by one processor in a multiprocessor configuration is delivered to all other processors in exactly the order issued. Strict order is simple, but has a problem similar to that of byte stores. An implementer may easily choose only 2 of the 3 features:

- Pipelined writes
- Bus retry
- Strict read/write ordering

If one processor starts a write to location A, starts a write to location B, discovers that the write of A failed (i.e., bus parity error), and retries it successfully, then a second processor will observe the writes out of order: B, then A.

Before Alpha AXP implementations, many VAX implementations avoided techniques such as pipelined writes to main memory, multibank caches, write buffer bypassing, routing networks, and crossbar memory interconnects, to preserve strict read/write ordering.

The Alpha AXP architecture’s shared memory model instead specifies no implicit ordering between the reads and writes issued on one processor, as viewed by a different processor. This programming model is an enabling technology for wide variety of high-performance implement-
This section describes the fundamentals, quite similar to the IBM System/370 serialization design [5].

**Processor State**
- Data Representation and Processor State

This section describes the fundamental Alpha AXP data types and their representation in memory and registers. It also describes the complete hardware register state for each processor, and outlines the additional state maintained by operating-system-specific PALcode routines. The Alpha AXP architecture differs from other RISC architecture by carefully specifying a canonical form for 32-bit data in 64-bit registers. A canonical form is a standardized choice of data representation for redundantly encoded values. Since 32-bit operations assume canonical operands and give canonical results, very few explicit conversions between 32- and 64-bit representations are needed.

The fundamental unit of data in the Alpha AXP architecture is a 64-bit quadword. As shown in Figure 1, quadwords may reside in memory or registers. For backward compatibility, 32-bit longwords may also be stored in memory.

There are three fundamental datatypes: integer, IEEE floating-point [6], and VAX floating-point [2], each available in 32-bit and 64-bit forms. VAX floating-point values in memory have 16-bit words swapped, for compatibility with VAX (and PDP-11) formats. The VAX floating-point load and store instructions do word swapping to give a common register order. The 32-bit Load instructions expand values to 64-bit canonical form, and the 32-bit store

---

**Glossary of Terms Used in this Article**

(Note: Items are grouped by related topics. Groups are separated by “———”)
instructions contract 64-bit values back to 32. All register-to-register operations are thus done on full 64-bit values in a common integer or floating-point format. No partial-register reads or writes are done.

The canonical form of a 32-bit value in a 64-bit integer register has the most significant 33 bits all equal to bit(31). In essence, bit(31) is kept as a "fat bit." This allows signed integer values to be used directly in 64-bit arithmetic and branches. This canonical form is maintained as a closed system (even for 32-bit data considered to be "unsigned") by using a combination of 64-bit operates, 32-bit add/sub/mul, and two-instruction sequences for shifts.

The canonical form of a 32-bit value in a 64-bit floating-point register has the 8-bit exponent field expanded to 11 bits, and the 23-bit mantissa field expanded to 52 bits. Except for IEEE denormals, this allows single-precision floating-point values to be used directly in double-precision arithmetic and branches. This canonical form is maintained as a closed system by using single-precision instructions.

Bytes and words (16-bit quantities) are not fundamental data types. They may be transferred between memory and registers with short sequences of instructions, and manipulated in registers using normal arithmetic and the byte manipulation instructions described in the section entitled "Operate Instructions."

The hardware processor state, shown in Figure 2, includes 32 integer registers R0..R31 of 64 bits each; R31 is always zero. There are also 32 floating-point registers F0..F31 of 64 bits each; F31 is always zero. Writes to R31 and F31 are ignored.

There is a 64-bit Program Counter (PC) containing a longword-aligned virtual byte address (i.e., the low 2 bits of the PC are always zero). The VAX architecture keeps the PC in general register 15, where it is directly used for PC-relative memory addressing. This is much less appropriate in the Alpha AXP architecture, because code and data pages will usually be separated by 64KB or more to allow separate memory protection, but the 16-bit displacement in load/store instructions cannot span more than 64KB.

There is also a lock flag and locked physical address for the load-locked/store-conditional sequence, and a floating-point control register containing the IEEE dynamic rounding mode.

Hardware implementations may optionally include a pair of state registers for memory prefetching (FETCH/FETCH.M instructions), and an optional Interrupt Flag for use only by translated VAX/OpenVMS programs that reproduce CISC instruction atomity using a sequence of RISC instructions [8, 9].

In addition to the preceding hardware state, the Privileged Architecture Library routines for the various operating systems implement additional state. This state may be maintained by hardware or (PALcode) software, at the option of the implementer, and it varies from one operating system to another. Typical PALcode state includes a Processor Status (PS) word, Kernel and User stack pointers, a Process Control Block base for context switching, a Process-unique value for threads, and a processor number for multi-processor dispatching. Additional PALcode states may include a floating-point enable bit, interrupt priority level, and translation lookaside buffers for mapping instruction-stream and data-stream virtual addresses. All this state is soft, in the sense that it is defined only in relationship to the PALcode routines for a specific operating system.

In a multiprocessor implementation, all of the preceding state is replicated for each processor.

Memory Access

Alpha AXP memory is byte addressed, using the lowest-numbered byte of a datum. Only aligned long-words or quadwords may be accessed: an aligned long-word is a four-byte datum whose address is a multiple of four; an aligned quad-word is an eight-byte datum whose address is a multiple of eight. Normal load or store instructions that specify an unaligned address take a precise data alignment trap to PALcode (which may do the access using two aligned accesses or report a fatal error, depending on the operating system design).

Alpha AXP implementations are bi-endian, allowing data to be accessed using either a little-endian view (byte 0 is the low byte of an integer), or a big-endian view (byte 0 is the high byte of an integer). As discussed later in the section Load/Store Instructions, there is a one-instruction bias in the sequences for big-endian byte manipulation.

Virtual addresses are a full 64 bits; implementations may restrict addresses to have some number of identical high-order bits, but must always distinguish at least 43 bits. Virtual addresses are mapped in an operating system-specific way to physical addresses, using fixed-size pages. Memory protection is done on a per-page basis. Address-mapping errors (i.e., protection, page faults) take precise traps to PALcode. Each page may also be marked to provide a fault on any read, write, or instruction-fetch.

Virtual addresses may be further qualified by address space numbers (ASNs), to allow multiple disjoint addresses spaces. The choice of disjoint or common mapping across all processes is done on a per-page basis. The virtual- to physical-address mapping is done on a per-page basis. Each implementation may have a page size of 8KB, 16KB, 32KB, or 64KB. The 64KB upper bound allows a linker to allocate blocks of memory with differing protection or ASN properties far enough apart to work on all implementations.
virtual- to physical-address mapping can be many to one (i.e., synonyms are allowed).

In a multiprocessor implementation, shared main memory locations have the same physical address on all processors. Per-processor unshared locations are also allowed.

Memory has longword granularity: two processors may simultaneously access adjacent longwords without mutual interference. The load-locked/store-conditional sequence discussed previously can be used to achieve multiprocessor byte
granularity.

Input/output is memory mapped: some physical memory addresses may refer to I/O device registers whose access triggers side effects (such as the transfer of data). Side effects on reads are discouraged.

**Instruction Formats**

There are four fundamental instruction formats, shown in Figure 3: Operate, Memory, Branch, and CALL_PAL. All instructions are 32 bits wide and reside in memory at aligned longword addresses. Each instruction contains a 6-bit opcode field and zero to three 5-bit register-number fields, Ra, Rb, and Rc. The remaining bits contain function (opcode extension), literal, or displacement fields. To minimize register file ports in fast implementations, Rb is never written, and Rc is never read.

All the operate instructions are 3-operand register-to-register, calculating Rc = Ra OP Rb. In integer operates, the opcode and a 7-bit function field specify the exact operation. Integer operates may have an 8-bit zero-extended literal instead of Rb. In floating-point operates, the opcode and an 11-bit function field specify the exact operation. There are no floating-point literals.

Memory format instructions are used for loads, stores, and a few miscellaneous operations. Loads andstores are 2-operand instructions, specifying a register Ra and a base-displacement virtual byte address. The effective address calculation sign extends the 16-bit displacement to 64 bits, and adds the 64-bit Rb base register (ignoring overflow). The resulting virtual byte address is mapped to a physical address. The miscellaneous instructions make other uses of the Ra, Rb, and displacement fields.

Branch format instructions specify a single register Ra and a signed PC-relative longword displacement. The branch target calculation shifts the 21-bit displacement left by 2 bits to make it a longword (not byte) displacement, then sign extends it and adds it to the updated PC. Conditional branch instructions test register Ra, and unconditional branches write the updated PC to Ra for subroutine linkage. The large longword displacement allows a range of +/-4MB, substantially reducing the need for branches around or to other branches.

The CALL_PAL instruction has only a 6-bit opcode and 26-bit function field. The function field is a small integer specifying one of a few dozen Privileged Architecture Library subroutines to invoke.

**Operate instructions**

There are five groups of register-to-register operate instructions: integer arithmetic, logical, byte-manipulation, floating-point, and miscellaneous. All instructions operate on 64-bit quadwords unless otherwise specified.

The INTEGER ARITHMETIC INSTRUCTIONS are add, subtract, multiply, and compare. Add, subtract, and multiply have variants that enable arithmetic overflow traps. They also have longword variants that check for 32-bit overflow (instead of 64) and force the high 33 bits of the result to all equal bit(31). Add and subtract also have scaled variants that shift the first operand left by 2 or 3 bits (with no overflow checking), to speed up simple subscripted address arithmetic. The UMULH instruction (from PRISM) gives the high 64 bits of an unsigned 128-bit product, and may be used for dividing by a constant. There is no integer divide instruction; a software subroutine is used to divide by a non-constant. The compares are signed or unsigned and write a Boolean result (0 to 1) to the target register.

The LOGICAL INSTRUCTIONS are AND, OR, and XOR, with the second operand optionally complemented (ANDNOT, ORNOT, XORNOT). The shifts are shift left logical, shift right logical, and shift right arithmetic. The 6-bit shift count is given by Rb or a literal. The conditional move instructions test Ra (same tests as the branching instructions), and conditionally move Rb to Rc. These can be used to eliminate branches in short sequences such as MIN(a,b).

The BYTE-MANIPULATION INSTRUCTIONS are used with the Load and Store Unaligned instructions to manipulate short unaligned strings of bytes. Long strings should be manipulated in groups of eight (aligned quadwords) whenever possible. The byte-manipulation instructions are fundamentally masked shifts. They differ from normal shifts by having a byte count (.7) instead of a bit count (0..63), and by zeroing some bytes of the result, based on the data size given in the function field.

The extract (EXTxx) instructions
Figure 4. Load/store instructions

Example 1: Load Byte (unsigned, little-endian)

<table>
<thead>
<tr>
<th>Byte</th>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

LDQ_U R2,0(R1)
EXTBL R2,R1,R2

Example 2: Load Byte (signed, big-endian)

<table>
<thead>
<tr>
<th>Byte</th>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>-2</td>
</tr>
</tbody>
</table>

LDQ_U R2,0(R1)
SUBQ R31,R1,R3
EXTQH R2,R3,R2

Example 3: Store Byte (little-endian)

<table>
<thead>
<tr>
<th>Old</th>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

LDQ_U R2,0(R1)
INSBL R0,R1,R3
MSKBL R2,R1,R2
OR R2,R3,R2
STQ_U R2,0(R1)

Example 4: Explicit Load Quadword (unaligned, little-endian)

<table>
<thead>
<tr>
<th>Low part</th>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

LDQ_U R2,0(R1)
LDQ_U R3,7(R1)
EXTQL R2,R1,R2
EXTQH R3,R1,R3
OR R2,R3,R2

Example 5: Multiprocessor Test-and-Set

<table>
<thead>
<tr>
<th>Flag</th>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

LDQ_L R2,0(R1)
BNE R2,flag_set
OR R2,#1,R2
STQ_C R2,0(R1)
BEQ R2,contention

COMMUNICATIONS OF THE ACM/February 1993/Vol.36, No.2
extract part of a 1-, 2-, 4-, or 8-byte field from a quadword and place the resulting bytes in a field of zeros. A single EXTxl instruction can perform byte or word loads, pulling the datum out of a quadword and placing it in the low end of a register with high-order zeros. A pair of EXTxl/ EXTxH instructions can perform unaligned loads, pulling the two parts of an unaligned datum out of two quadwords and placing the parts in result registers, where they are ready for combining into the full datum by a simple OR.

The insert (INSxx) and mask (MSKxx) instructions position new data and zero out old data in registers\(^8\) for storing bytes, words, and unaligned data.

The compare-byte instruction allows character-string search and compare to be done eight bytes at a time. The ZAP instructions allow zeroing of arbitrary patterns of bytes in a register. These instructions allow very fast implementations of the C language string routines, among other uses.

The FLOATING-POINT ARITHMETIC INSTRUCTIONS are add, subtract, multiply, divide, compare, and convert. The first four have variants for IEEE and VAX, single and double. They also have variants that enable combinations of arithmetic traps, and that specify the rounding mode. The single-precision instructions write canonical 64-bit results, but do exponent checking and rounding to single-precision range. The compares write a Boolean result (0 or non-zero) to the target register. The converts transfer between single and double, floating-point and integer, and two forms of VAX double (D-float and G-float). A combination of hardware and software provides full IEEE arithmetic. Operations on VAX reserved operands, dirty zeros, IEEE denormals, infinities, and not-a-numbers are done in software.

There are also a few floating-point instructions that move data without applying any interpretation to it. These include a complete set of conditional move instructions, similar to the integer conditional moves. The MISCELLANEOUS INSTRUCTIONS include memory prefetching instructions to help decrease memory latency, a Read Cycle Counter instruction for performance measurement, a Trap Barrier instruction for forcing precise arithmetic traps, and Memory Barrier instructions for forcing multiprocessor read/write ordering.

Load/Store instructions

The load and store instructions only move data. They never apply an interpretation to the data, and therefore never take any data-dependent traps. This design allows moving completely arbitrary bit patterns in and out of registers, and allows completely transparent saving/restoring of registers.

The integer Load and Store Quadword Unaligned (LDQ_U, STQ_U) instructions ignore the low three bits of the byte address and always transfer an aligned quadword. These are used with the in-register byte manipulation instructions to operate on byte, word, and unaligned data via short sequences of RISC instructions.

Example 1 in Figure 4 shows a 2-instruction sequence for loading a byte into the low end of a register, using little-endian byte numbering. Example 2 shows a similar sequence for loading a byte into the high end of a register, using big-endian byte numbering. Example 3 shows a sequence for storing a byte (the first 2 and last 2 instructions may dual-issue on the first Alpha AXP implementation). Example 4 shows a sequence for an explicit unaligned load quadword (no data alignment trap).

The integer load-locked and store-conditional (LDQ_L, LDLL_L, STQ_C, STLCL_C) instructions are included in the architecture to facilitate atomic updates of multiprocessor shared data. As described previously, they can be used in short sequences of RISC instructions to do atomic read-modify-writes. Example 5 shows a sequence for doing a multiprocessor test-and-set. Note that changing the LDQ_U/STQ_U in Example 3 to AND/LDQ_L/STQ_C/BEQ gives a multiprocessor-safe byte store sequence.

There are two related load address instructions. LDA calculates the effective address and writes it into Rc. LDAH first shifts the displacement left 16 bits, then calculates the effective address and writes into Rc. LDAH is included to give a simple way of creating most 32-bit constants in a pair of instructions.\(^7\) 64-bit constants are loaded with LDQ instructions.

Branching instructions

The branch instructions include conditional branches, unconditional branches, and calculated jumps. In addition to the previously described conditional moves, the architecture contains hints to improve branching performance.

The integer conditional branches test register Ra for an opcode-specified condition (\(>0 \geq 0 =0 <0\) even odd) and either branch to the target address or fall through to the updated PC address. The floating-point conditional branches are the same, except they do not include even/odd tests. Arbitrary testing (and faulting on VAX or IEEE nonfinite values) can be done by sequences of Compare instructions and Branch instructions. Logical or arithmetic instructions can combine Compare results without using branches.

Unconditional branches write the updated PC to Ra for subroutine linkage and branch to the target address. Ra = R31 may be used if no linkage is needed.

Calculated jumps write the updated PC to Ra and jump to the target address in Rb. Calculated jumps are used for subroutine call, return, CASE (SWITCH) statements, and coroutine linkage.

The architecture specifies three kinds of branching hints in instructions. The hints need not be correct,
but to the extent that they are, implementa-
tions may perform faster.

The first form of hint is an archi-
tected static branch prediction rule: forward conditional branches are predicted not-taken, and back-
ward ones taken. To the extent that compilers and hardware implemen-
tors follow this rule, programs can run more quickly with little hardware
cost. This hint does not preclude
doing dynamic branch prediction in an
implementation, but it may re-
duce the need to do so.

The second form describes com-
puted jump targets. Otherwise-
unused instruction bits are defined to
give the low bits of the most likely
target, using the same target calcula-
tion as unconditional branches. The
14 bits provided are enough to spec-
ify the instruction offset within a
page, which is often enough to start a
fastest-level instruction cache fetch
many cycles before the actual target
value is known.

The third form describes subro-
tine and coroutine returns. By mark-
ing each branch and jump as 'call',
'return', or 'neither', the architecture
provides on implementation enough
information to maintain a small stack
of likely subroutine return addresses. This implementation stack can be
used to prefetch subroutine returns
quickly.

The Conditional Move instruc-
tions (see Operate Instructions sec-
tion) and the branching hints elimi-
nate some branches and speed up the
remaining ones without compromis-
ing multiple instruction issue.

Supervision
The underpinnings of an operating
system are all done in PALcode sub-
routines, and are a flexible part of
the architecture. All asynchronous
events, such as interrupts, excep-
tions, and machine errors, are medi-
ated by PALcode routines. PALcode
establishes the initial state of the
machine before execution of the first
software instruction. PALcode rou-
tines mediate all accesses to physical
hardware resources, including physi-
cal main memory and memory-
mapped I/O device registers.

This design allows implementers
to craft a set of PALcode routines that
closely match an operating sys-
tem design, not only for traditional
operating systems, but also for spe-
cialized environments such as real-
time or highly secure computing. As
new computing paradigms are adopted
and new operating systems are created, the Alpha AXP architec-
ture may well prove flexible enough
to accommodate them efficiently.

Future Changes
The Alpha AXP architecture will
surely change during its lifetime. In
addition to the PALcode flexibility
discussed previously, there is explicit
performance flexibility and instruc-
tion-set flexibility in the architecture.

Architecture fields that are too
small can limit performance. The
Alpha AXP architecture therefore
has many fields deliberately sized for
later expansion.

While initial implementations use
only 43 bits of virtual address, they
check the remaining 21 bits, so that
software can run unmodified on later implementations that use (up to) all 64 bits. Similarly, while initial
implementations use only 34 bits of
physical address, the architected
Page Table Entry (PTE) formats and
page-size choices allow growth to 48
bits. By expanding into a 16-bit PTE
field that is not currently used by
mapping hardware, a further 16 bits
of physical address growth can be
achieved, if ever needed.

While initial implementations use
only 6KB pages, the design accom-
modates limited growth to 64KB
pages. Beyond that, page table gran-
ularity hints allow groups of 8, 64, or
512 pages to be treated as a single
large page, thus effectively extend-
ing the page size range by a factor of
over 1,000. Each architected PTE
format also has one bit reserved for
future expansion.

Several other soft PALcode regis-
ters, such as the PS or ASN, that
need only a few bits today are allo-
cated a full 64 bits for future expan-
sion.

Exception processing can limit
performance. PALcode routines del-
iver exceptions to an operating sys-
tem, so the design can be gradually
improved.8 Some currently specified
software exceptions (such as IEEE
denormal arithmetic) could be moved into PALcode or hardware.

There are a number of areas of
instruction-set flexibility designed
into the architecture. Four of the
6-bit opcodes are nominally reserved
for adding integer and floating-point
aligned octaword (128-bit) load/store
instructions.9 Nine more 6-bit op-
code remain for other expansion.
Within each opcode, the function
field contains room for further ex-
ansion.10

Within the IEEE floating-point
function field, code points are nomi-
really reserved for double-extended
precision (128-bit) arithmetic. Within
the Memory Barrier instruction
group, three code points were re-
erved for subset barriers.11

Not all changes involve growth.
There are subsetting rules defined
for removing one or both floating-
point datatypes (IEEE and VAX). If
both are removed, the floating-point
registers can also be removed. The
AMO/Vxx PALcode routines and
RS/RC instructions are defined as
optional, and may be deleted when
the transition of translated VAX
code is completed. Other unneeded
PALcode routines could also be re-
move eventually.

Summary
The goals that shaped the Alpha
AXP architecture design have largely
been realized. For high perfor-

ance, the first implementation (the
DECchip 21064) is listed in the Octo-
ber 1992 Guinness Book of Records
as the world's fastest single-chip mi-
croprocessor. It is too early to mea-
sure longevity, but the fact that we
had designed-in flexibility in places
that changed during development is
at least encouraging. OpenVMS

8Many commercially successful architectures
have grown to double-width memory imple-
mentations in mid-life: the IBM 709 series from
36 to 72 bits; the IBM System/360 series from
32 to 64 bits; the Digital PDP-11 series from 16
to 32 bits; and the Digital VAX series from 32 to
64 bits. This trend is likely to continue.

9For example, the scaled add/subtract func-
tions were added between prototype chip and
product chip. The fact that the function fields
are not fully policed is a mistake.

11One of these has already been defined as a
write-write barrier.
AXP, Unix (OSF/1), and Windows NT operating systems all run on Alpha AXP implementations today. Programs from the VAX and MIPS architecture customer bases transport easily to Alpha AXP implementations, and run quickly.

Many of the ideas in the Alpha AXP design are now being adopted by other architectures in the industry.

Acknowledgments
Hundreds of people have worked on the Alpha AXP architecture, hardware, and software. Many Alpha AXP architectural ideas came from the PRISM design [3] most notably the PALCode idea. The architecture work was done in the rich environment of dozens and later hundreds of bright, thoughtful, and outspoken professional peers. Ellen Bambouta, Dileep Bhandarkar, Richard Brunner, Wayne Cardoza, Dave Cutler, Daniel Dobberpuhl, Robert Giggio, Henry Greib, Richard Grove, Robert Halstead, Jr., Michael Harvey, Nancy Kronenberg, Raymond Lanza, Stephen Morris, William Joyce, Charles Nylander, Dave Orbitis, Mary Payne, Audrey Reith, Robert Supnik, Benjamin Thomas, Catharine van Ingen, and Rich Witek all contributed directly to the written specification. Rich Witek is coarchitect, and is the other half of the term "we" used in this article.

References

AlphaAXP, OpenVMS AXP, PDP-11, VAX, and DEC OSF/1 AXP are trademarks of Digital Equipment Corporation. Unix is a trademark of Unix Systems Laboratories, Inc.

CR Categories and Subject Descriptors: B.7.1 [Input/Output and Data Communications]: Integrated Circuits—Types and design styles; C.1.2 [Computer Systems Organization]: Processor Architectures—Multiple Data Stream Architectures (Multiprocessors)

General Terms: Design

Additional Key Words and Phrases: Alpha AXP chip

About the Author:
RICHARD L. SITES is a senior consultant engineer in the semiconductor engineering group at Digital Equipment Corporation, where he is working on binary translators and the Alpha AXP architecture. He has worked at IBM, Hewlett-Packard, and Burroughs and holds a number of patents on computer hardware and software inventions. Author's Present Address: Digital Equipment Corporation, 153 Taylor Street, TAY2-2/F14, Littleton, MA 01460; email: sites@tallas.enet.dec.com

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© ACM 0002-0782/93/0200-035 $1.50

Our Planet’s Rain Forests Are Burning.

Less rain forest means less rain. Less oxygen for us all to breathe. And an even greater threat of global warming. When these rain forests go, gone forever are the rare plants, animals and cultures that call them home.

Rain Forest Rescue.
The National Arbor Day Foundation

You can help stop this senseless destruction. Right now, you can join The National Arbor Day Foundation, the world’s largest tree-planting environmental organization, and support Rain Forest Rescue to help stop further burning. You’d better act now.

Call Rain Forest Rescue.
1-800-255-5500
Mit dem P6 ist Intel unterwegs, neue Welten zu entdecken, die nie ein x86-Prozessor zuvor betreten hat: man spricht von höchster Performance, neuester Technologie und modernsten Architekturmerkmalen. So weit entfernt vom Stand der irdischen Technik sind diese aber nicht. Vielmehr trifft man auf manches, das man schon aus anderen Prozessorschmieden kennt.


Dann soll der P6 zunächst in einer 133-MHz-Version auf den Markt kommen, die gut 200 SPECint erreicht, knapp doppelt so viel wie der derzeit schnellste Pentium. Auf Applikations-Level verspricht Intel eine Leistungskennzahl von gut 50 Prozent.


Um die Anzahl der parallel zu verarbeitenden µOps noch weiter zu erhöhen, bedient sich Intel einer weiteren bei RISC-CPU's altbekannten Technik: An Verzweigungen wartet der Prozessor nicht, bis klar ist, welchen Weg das Programm einschlagen wird, sondern führt einen Zweig, den er für wahrscheinlich hält, auf guten Glück schon mal aus (Speculative Execution — gibt's schon beim Cyrix).

Um die Anzahl der parallel zu verarbeitenden µOps noch weiter zu erhöhen, bedient sich Intel einer weiteren bei RISC-CPU's altbekannten Technik: An Verzweigungen wartet der Prozessor nicht, bis klar ist, welchen Weg das Programm einschlagen wird, sondern führt einen Zweig, den er für wahrscheinlich hält, auf gut Glück schon mal aus (Speculative Execution — gibt's schon bei- spielsweise bei den MIPS-CPU's).

Extra-Cache


**Neue Befehle**


Wie der Pentium ist auch der P6 ein reinrassiger 32-Bit-Prozessor. Eine Erweiterung der internen Register von 32 auf 64 Bit fand nicht statt. Der Datenbus ist nur deshalb 64 Bit breit ausgeführt, damit der P6 im Haupteinsatz bleibt. Der Adreßbus des P6 wurde von 32 (Pentium) auf 36 Bit erweitert. Damit kann der P6 bis zu 64 GByte Hauptspeicher adressieren – das dürfte für die nächsten Jahre mehr als ausreichend sein.


Der P6 wird – wie alle neueren Intel-Prozessoren – den System-Management-Mode (SMM) unterstützen. Darüber hinaus kennt er alle Funktionen aus dem bisher strengsten unter Verschlüsselung gehaltenen Ap...


**Neues Businterface**

Als Verbindung zur Außwelt verwendet der P6 ein 64 Bit breites Bus-Interface, das mit maximal 66 MHz arbeitet. Das Teilungsverhältnis zwischen externem und internem Takt wird über Sense-Pins bestimmt und beträgt wahlweise 1:2, 1:3 oder 1:4. Trotz nominal gleicher externer Frequenz ist die Bandbreite des P6-Busses erheblich höher als die des Pentium. Wenn der Pentium Daten vom Bus liest, sendet er zunächst die Adresse und wartet dann, bis die Daten bereitstehen. Der P6 geht raffinierter vor: Nachdem er die Adresse angelegt hat, gibt er den Bus frei und füllt interne Operationen durch. Erst wenn die angeforderten Daten bereitstehen, setzt er den Bus wieder frei und führt interne Operationen durch. Erst wenn die angeforderten Daten bereitstehen, setzt er den Bus wieder frei und führt interne Operationen durch. Erst wenn die angeforderten Daten bereitstehen, setzt er den Bus wieder frei und führt interne Operationen durch.


Wie der P54C besitzt auch der P6 einen integrierten APIC (Advanced Programmable In-
In medias res

Beim P6 gibt es einige interne Details, die eine genauere Betrachtung verdienen. Hierzu gehört vor allem die Generierung von Befehlsfolgen, die auch der Programmerer im Hinterkopf haben sollte, wenn er optimale P6-Code schreiben will.


Im Reorder Buffer befindet sich auch die Register Alias Tabelle (RAT), die 40 General-Register zur Verfügung stellt. Diese dienen als Renaming-Register für die µOps. Daneben stehen noch das Original-Registerset (RRF, Real Register File) zur Verfügung. Der ROB verknüpft pro Tak drei µOps mit Registern aus dem Real Register File oder der Register Alias Table. Die so komplettrierten µOps landen in der 20 Einträge großen Reservation Station. Alle bisher durchgeführten Operationen erfolgen in order, also innerhalb der Reihenfolge, die der Programmiervorgabe vorgesehen hat.

Out-of-Order

Im Reservation-Station werden die µOps so lange, bis ihre Operationen komplett sind. Ist dies der Fall, werden sie als 'ready' markiert und ausgeführt, sobald eine geeignete Rechenseinheit frei ist. Im Regelfall stehen mehr µOps zur Verarbeitung bereit, als Ausführungseinheiten verfügbar sind. Welche Befehle bevorzugt bearbeitet werden, bestimmt ein komplexes Regelwerk, über dessen Details Intel bisher nur wenig bekanntgegeben hat. Sicher ist lediglich, daß die µOps zuerst bearbeitet werden, die bereits lange auf die Ausführung warteten. Eine weitere Regel besagt, daß die Operationen bevorzugt abgearbeitet werden, die noch zur Komplettierung eines in nächster Zukunft ausgegebenen x86-Befehls fehlen.

Die Verarbeitungsgeschwindigkeit der einzelnen Ausführungseinheiten variiert stark. Die In-teger-Unit verarbeitet die meisten Befehle in einem Tak, Multiplikationen benötigen aber vier Takte (Pentium: 7–14 Takte) und Divisionen zwischen 12 und 36 Takte (Pentium: 42–84 Takte). Bei der Floating-Point-Unit fallen die Unterschiede zum Pentium nicht so stark ins Gewicht. Eine Addition benötigt, wie beim Pentium, drei Takte, die Multiplication wurde zwei Takte langsamer (fünf statt drei Takte). Für die Division braucht die P6 zwischen 18 und 38 Takte (Pentium: 39 Takte) und für das Wurzelziehen zwischen 29 und 69 Takte (Pentium: 70–140 Takte). Der beliebte FXCH-Befehl, der die beiden Operanden auf dem FPU-Stack vertauscht, benötigt quasi überhaupt keinen Takt mehr, da er via Register-Renaming erledigt wird. Seine 'Bearbeitungszeit' geht also in der Queue unter.


Wenn eine µOp ausgeführt ist, landet sie entweder im MOB oder im ROB. Ihr Resultat wird dort in das Register-File geschrieben, anschließend ist der von ihr belegte Platz frei für neue µOps. Der ROB kann bis zu drei µOps pro Tak ab-


MP-Support

Der Split-Transaction-Bus bietet ideale Voraussetzungen für ein MP-System. Jede CPU belegt den Bus nur so lange wie unbedingt nötig, und die Bandbreite von 528 MByte/s dürfte für vier Prozessoren durchaus ausreichend sein. Neben den vier CPUs steht Intels MP-Konzept noch vier weitere Devices vor, die um die Herrschaft auf dem Systembus ringen. Dies sind zwei Memory-Kontroller und zwei I/O-Bridges. Letztere sind besonders interessant, lassen sich doch mit zwei solchen Devices Rechner aufbauen, die über bis zu acht PCI-Slots verfügen. Dies prädestiniert MP-Systeme mit dem P6 als High-

Beim P54C (Pentium-90 und -100) schrängt der gemeinsam genutzte L2-Cache die Anwendungsmöglichkeiten auf Desktop-Applikationen ein. Transaktionsorientierte Anwendungen wie etwa SQL-Server können so nur geringen Gewinn aus der zweiten CPU ziehen. Da dieser Flaschenhals bei P6-basierenden MP-Systemen wegfallt, dürfte die Akzeptanz hier deutlich höher sein. Einziger Haken ist der enorme Platzbedarf des P6 auf dem Motherboard. Vier Prozessoren lassen sich auf dem
Hydra Intel?

Mit dem P6 zementiert Intel seinen Führungsanspruch im x86-Markt für die nächsten Jahre. Die unumstrittene Konkurrenz schickt sich gerade erst an, zum Pentium aufzuschließen. Wenn AMDs K86 und der M1 Ende des Jahres in Stückzahlen auf den Markt kommen, heißt es bei Intel nur 'Bin schon da!'.


Der P6 zielt aber nicht nur gegen die Emporkömmlinge im x86-Lager. Mit seiner starken Ausrichtung auf MP-Systeme willt er auch im Revier der klassischen RISC-Prozessoren. Diese hatten bisher mit dem High-End-Server-Markt eine feste Bastion, die für Renommee und vor allem für sichere Gewinne sorgte. Sollte es Intel tatsächlich gelingen, diese Festung mit dem P6 zu knacken, wird die Luft für die Mitbewerber dünn. Magere Finanzen, das bekommt DEC derzeit schmerzhaft zu spüren, sind keine gute Basis, um High-End-CPU's weiterzuentwickeln.


Der bunte und vielfältige PC-Markt könnte an den Folgen dieser Entwicklung zusammenbrechen – schließlich betätigt sich Intel nicht mehr nur als CPU- und Chipsatz-Hersteller, sondern fertigt auch noch selbst PC-Platinen. Ein P6-Board, das zu 90 Prozent aus Intel-Komponenten besteht, kann nun mal am billigsten von Intel selbst hergestellt werden. Schließlich dürfte sich Intel also mit dem P6 neben dem CPU- und auch noch das Board-Monopol?

Orion, wie der neue Pentium-Chipsatz Triton, neben normalen Page Mode DRAMS auch schnelle EDO-DRAMS unterstützt.

Preise & Perspektiven

Wer sich um die Sicherheit seiner Daten im DRAM Sorgen macht, wird mit dem Orion optimal bedient. Statt der relativ nutzlosen Parity-Prüfung unterstützt der Chipsatz die Option, das DRAM mit ECC (Ein-Bit Fehlerkorrektur) zu operieren. Dies bedeutet bei einer Busbreite von 64 Bit keinerlei zusätzliche Kosten gegenüber Parity-DRAMs – in beiden Fällen werden acht zusätzliche Bits benötigt.


Lieferengpässe wie in der Einführungsphase des Pentium dürften uns beim P6 erspart bleiben. Der Chip wird in einem 0,6-Micron-BiCMOS-Prozeß hergestellt, den Intel auch bei den Pentiums mit 90, 100 und 120 MHz verwendet. Anfangs waren die Kosten für den Intel 0,4-Micron-Fab ihren Betrieb aufgenommen haben, sollte dann die zweite Welle mit 160 und 200 MHz Taktfrequenz rollen. Irgendwann danach steht sicher auch ein P6 mit 512 KByte großem L2-Cache ins Haus.


Fazit


Literatur

Compilers for vector or multiprocessor computers must have certain optimization features to successfully generate parallel code.

DAVID A. PADUA and MICHAEL J. WOLFE

Supercomputers\(^1\) use parallelism to provide users with increased computational power. Most supercomputers are programmed in some higher level language, commonly Fortran: all supercomputer vendors provide Fortran compilers that detect parallelism and generate parallel code to take advantage of the architecture of their machines\(^2\) [25, 46, 53].

This article discusses some of the common (and not so common) features that compilers for vector or multiprocessor computers must have in order to successfully generate parallel code. The many examples given throughout are related to the generic types of machines to which they apply. Where appropriate, we also relate these parallel compiler optimizations to those used in standard compilers.

\(^{1}\) Some of the supercomputers available today are the FX Series from Alliant Computer Corporation, the Cyber 205 from Control Data Corporation, the Convex C-1 and Convex Computer Corporation, the Cray 2 and the Cray X-MP [15] from Cray Research, the Facom VP from Fujitsu [30], the S-810 Vector Processor from Hitachi [40], the SX System from NEC Corporation, and the SCS-411 from Scientific Computer Systems. Alliant’s FX and Convex’s C-1 are usually classified as minisupercomputers.

\(^{2}\) Besides the vendor-supplied compilers, there are a number of experimental and third-party source-to-source restructurers. Among them are the University of Illinois’s ParalleX [32], KAM’s KAP [17, 18], Rice University’s PFC [5], and Pacific Sierra’s VAST [11].

This work was supported in part by the National Science Foundation under Grant US NSF-DCR-84-00016 and US NSF-DCR-84-01100, the U.S. Department of Energy under Grant US DOE-DE-FG02-85ER25001, and the IBM Donation.

Fortran is currently the programming language of choice for supercomputers, largely because vendors presently provide optimizing, vectorizing, and concurrentizing\(^3\) compilers only for Fortran. In addition, Fortran has historically been the most frequently used numerical programming language, and much investment has been made in its programs. The examples in the text are written in Fortran, using some of the new features described in the latest Fortran-8x proposal [7], such as the end do statement and array assignments. The literature distinguishes two types of concurrent loops: doall and doacross [16, 43]. The latter imposes a partial execution order across iterations in the sense that some of the iterations are forced to wait for the execution of some of the instructions from previous iterations. Doall loops do not impose any partial ordering across iterations, even though there may be critical regions in the loop bodies.\(^4\) Despite our use of Fortran, none of the features or transformations explained here are peculiar to this language, and they can be successfully applied to many others.

\(^{3}\) The term parallel is often used to describe the translation of serial code for parallel computers. We prefer the more specific concurrent over parallel, since parallel may mean vector, concurrent, or lock-step multiprocessor computation. Also, Alliant Computer Corporation, the first commercial vendor to supply a compiler that automatically translates code for multiple processors, uses the term concurrent.

\(^{4}\) The DOALL statement was originally defined in the Burroughs’s FMP Fortran [12, 30].
We will discuss how data-dependence testing of some form is required in any compiler that wishes to detect parallelism in serial code, and explain the different types of parallel code that can be generated. Vector code is appropriate for computers with vector instructions sets, while concurrent constructs are used in multiprocessor environments. Also covered are ways to improve the computation of the data-dependence graph. A (very incomplete) catalog of transformations and restructuring tricks that compilers use to optimize code for parallel computers is given, followed by a discussion of how these compilers communicate with programmers.

DATA DEPENDENCE

Data-dependence testing [5, 8, 10, 52] is required for any form of automatic parallelism detection. Data-dependence relations are used to determine when two operations, statements, or two iterations of a loop can be executed in parallel. For instance, in the code

\[ S_1: A = B + C \]
\[ S_2: D = A + 2 \]
\[ S_3: E = A \cdot 3 \]

statements \( S_1 \) and \( S_3 \) cannot be executed at the same time since \( S_1 \) uses the value of \( A \) that is computed by \( S_3 \). This is called true dependence or flow dependence since the data value flows from \( S_1 \) to \( S_3 \), and is denoted \( S_1 \rightarrow S_3 \). \( S_3 \) also depends on \( S_1 \) (denoted \( S_1 \leftarrow S_3 \)): thus \( S_3 \) must be executed before both \( S_1 \) and \( S_3 \). The data-dependence relations are often depicted in a data-dependence graph, with arcs representing the relations, as follows:

\[ S_1 \rightarrow S_2 \]
\[ S_2 \rightarrow S_3 \]

Notice that \( S_2 \) and \( S_3 \) are not connected by data-dependence arcs and so may be executed in parallel if two processors are available.

Two other kinds of data dependence are important. In the program segment

\[ S_1: A = B + C \]
\[ S_2: B = D / 2 \]

\( S_1 \) uses the value of \( B \) before \( S_2 \) assigns a new value to \( B \). Since \( S_1 \) is to use the "old" value of \( B \), it must be executed before \( S_2 \); this is called antidependence, as the relation is from the use to the assignment, and is denoted \( S_1 \leftarrow S_2 \). The third kind of dependence is shown in the program segment below:

\[ S_1: A = B + C \]
\[ S_2: D = A + 2 \]
\[ S_3: A = E + F \]

Here \( S_1 \) assigns a new value to \( A \) after \( S_2 \), which has already given a value to \( A \). If \( S_2 \), is executed after \( S_1 \), then \( A \) will contain the wrong value after this program segment. Thus, \( S_1 \) must precede \( S_2 \); this is called output dependence and is denoted \( S_1 \rightarrow S_3 \).

The flow of control must also be taken into account when building data-dependence relations. For instance, in the program segment

\[ S_1: A = B + C \]
\[ \text{if } (X >= 0) \text{ then} \]
\[ S_2: A = 0 \]
\[ \text{else} \]
\[ S_3: D = A \]
\[ \text{end if} \]

the relations \( S_1 \rightarrow S_2 \) and \( S_2 \rightarrow S_3 \) hold, but \( S_2 \rightarrow S_3 \) does not hold even though \( S_2 \) assigns a value to \( A \) and \( S_3 \) uses \( A \), and \( S_1 \) appears after \( S_2 \) in the program. Since \( S_2 \) and \( S_3 \) are on different branches of the same if statement, the value of \( A \) used in \( S_3 \) will never come from \( S_2 \).

Since the actual execution flow of a program is not known until run time, a data-dependence relation does not always imply data communication or memory conflict. For instance, in this program segment

\[ S_1: A = B + C \]
\[ C: \text{if } (X >= 0) \text{ then} \]
\[ S_2: A = A + 2 \]
\[ \text{end if} \]
\[ S_3: D = A \cdot 2.1 \]

the data-dependence relations \( S_1 \rightarrow S_2 \) and \( S_2 \rightarrow S_3 \) will both be computed by the compiler, even though \( S_3 \) will in fact take the value of \( A \) from only one of \( S_1 \) or \( S_2 \), depending on the value of \( X \).

Many compilers also use the concept of dependence from an if statement to the statements under control of the if. This is called control dependence and is denoted \( \delta' \). In the program segment above, for example, the control-dependence relation \( C, \delta' S_2 \) holds. Control-dependence relations are often added to the data-dependence graph in order to find which statements in the program can be reordered, or when looking for cycles in the graph.

Data Dependence in Loops

Inside loops we are interested in data-dependence relations between statements and also dependence...
relations between instances of statements. We distinguish between different instances of execution of a statement by superscripting the statement label with the loop iterations. For instance, in the loop

\[
\begin{align*}
do & \ I = 1,3 \\
S_1: & \quad A(I) = B(I) \\
do & \ J = 1,2 \\
S_2: & \quad C(I, J) = A(I) + B(J) \\
& \quad \text{end do} \\
& \quad \text{end do}
\end{align*}
\]

statement \(S_1\) is executed three times: \(S_1', S_1'', S_1'''\) and \(S_2\) is executed six times: \(S_2', S_2'', S_2''', S_2''''', S_2''''''\) (we put the outer loop iteration number first).

We can also draw the iterations as points with Cartesian coordinates, as below:

\[
\begin{align*}
S_1' \quad \rightarrow \quad S_1'' \quad \rightarrow \quad S_1''' \\
\downarrow \\
S_2' \quad \rightarrow \quad S_2'' \quad \rightarrow \quad S_2'''
\end{align*}
\]

This diagram illustrates the iteration space; the dotted arrows show the order in which the instances of the statements are executed.

To find data-dependence relations in loops, the arrays and subscripts are examined. In the loop

\[
\begin{align*}
do & \ I = 2, N \\
S_1: & \quad A(I) = B(I) + C(I) \\
S_2: & \quad D(I) = A(I-1) \\
& \quad \text{end do}
\end{align*}
\]

the relation \(S_1 \Rightarrow S_2\) holds, since, for any iteration \(i\), \(S_1\) will assign \(A(I)\) and \(S_2\) will use \(A(I)\) on the same iteration of the loop. Since the dependence stays in the same iteration of the loop, we say \(S_1 \Rightarrow S_2\).

In the following similar loop

\[
\begin{align*}
do & \ I = 2, N \\
S_1: & \quad A(I) = B(I) + C(I) \\
S_2: & \quad D(I) = A(I-1) \\
& \quad \text{end do}
\end{align*}
\]

the relation \(S_1 \Rightarrow S_2\) still holds, but for any iteration \(i\), \(S_1\) will use an element of \(A\) that was assigned on the previous iteration of the loop by \(S_1''\) (except \(S_2\) which uses an "old" value of \(A(I)\)). Since the dependence flows from iteration \(i-1\) to iteration \(i\), we say that \(S_1 \Rightarrow S_2\) (the relation is \(\Rightarrow\) because \(i - 1 < i\)).

A third similar example is shown below:

\[
\begin{align*}
do & \ I = 2, N \\
S_1: & \quad A(I) = B(I) + C(I) \\
S_2: & \quad D(I) = A(I+1) \\
& \quad \text{end do}
\end{align*}
\]

In this loop, for any iteration \(i\), \(S_1\) will use an element of \(A\) that will be reassigned by \(S_1''\). Since \(S_2\) should use an "old" value of \(A\), the antidependence relation \(S_2 \Rightarrow S_1\) holds. This relation flows from iteration \(i\) to iteration \(i+1\), so we say \(S_2 \Rightarrow S_1\) (since \(i < i + 1\)).

The \(=\) or \(<\) used as the subscript of \(\Rightarrow\) is called the data-dependence direction, since it gives the direction of the dependence relation in the iteration space. In nested loops, there is a direction for each loop; these comprise a data-dependence direction vector. For instance, in the loop

\[
\begin{align*}
do & \ I = 1, N \\
do & \ J = 2, N \\
S_1: & \quad A(I, J) = A(I, J-1) + B(I, J) \\
S_2: & \quad C(I, J) = A(I, J) + D(I+1, J) \\
S_3: & \quad D(I, J) = 0.1 \\
& \quad \text{end do} \\
& \quad \text{end do}
\end{align*}
\]

the following dependence relations hold:

\[
\begin{align*}
S_1 & \Rightarrow S_2 \Rightarrow S_3 \\
S_1 & \Rightarrow S_2 \Rightarrow S_3 \\
S_1 & \Rightarrow S_2 \Rightarrow S_3 \\
\end{align*}
\]

Since most array subscripts are simple, simple tests are usually sufficient. Some compilers (e.g., Cray Fortran [CFT] and Control Data's Fortran 200 compilers) use restricted tests that allow only certain array subscript expressions, such as \(A(I)\) or \(A(I*C+k)\) (where \(c\) and \(k\) are constants). Dependence is assumed for any array reference that does not conform to the restrictions. These tests work well for new codes that are written with a particular computer in mind, since programmers will know what loops they want executed in parallel and will help the compiler by keeping their loops simple.

Sophisticated methods have been developed to handle more general cases (see [5, 8, 10, 52]). For instance, in order for data flow-dependence to be caused by the two array references to \(A\) here

\[
\begin{align*}
do & \ I = L, U \\
S_1: & \quad A(C*I+J) = \ldots \\
S_2: & \quad \ldots = A(d*I+k) \\
& \quad \text{end do}
\end{align*}
\]
(where \(c, d, j,\) and \(k\) are integer constants), the greatest common divisor of \(c\) and \(d\) (GCD \((c,d)\)) must divide \((k-j)\). For example, no dependence would exist in the following loop

\[
\begin{align*}
d & \leftarrow L, U \\
S_1 & : A(2*1) = \ldots \\
S_2 & : \ldots = A(2*I+1)
\end{align*}
\]

since the GCD \((2,2)=2\), which does not divide \(1-0=1\).

More importantly, there must exist two values of the loop index variable \(I\), say \(x\) and \(y\), such that

\[
L \leq x \leq y \leq U \\
c*x+j = d*y+k
\]

In the sample loop below

\[
\begin{align*}
d & \leftarrow 1,10 \\
S_1 & : A(19*1+3) = \ldots \\
S_2 & : \ldots = A(2*I+21)
\end{align*}
\]

the only two values that satisfy this dependence equation are \(x = 2\) and \(y = 10\):

\[
19*2+3 = 41 = 2*10+21
\]

Solving this Diophantine equation is the subject of several of the references cited above, which also generalize this to nested loops where several loop indexes appear in a single subscript.

**CODE GENERATION**

When a good data-dependence graph has been built for a loop nest, the compiler can generate parallel code. Since many supercomputers have vector instruction sets, vectorization is important. Most vector computers can compute certain reduction operations, such as the sum of all the elements of a vector, using vector instructions. At least one supercomputer also has hardware to assist in the solution of first order linear recurrences.\(^3\)

Newer supercomputers achieve higher speeds by using multiple processors. For these machines, generation of concurrent code will utilize all processors. Concurrent loops and concurrent blocks of code are two types of parallel code that can be generated.

**Loop Vectorization**

Loops are vectorized by examining the data-dependence graphs for innermost loops. A simple graph algorithm to find cycles in the data-dependence graph identifies any trouble spots in vectorization. If there are no cycles in the graph, then the whole loop can be vectorized. For example, the following loop

\[
\begin{align*}
d & \leftarrow 1,N \\
S_1 & : A(I) = B(I) \\
S_2 & : C(I) = A(I) + B(I) \\
S_3 & : E(I) = C(I+1)
\end{align*}
\]

has the following data-dependence graph:

\[
S_1 \quad \Downarrow \quad \text{Flow dependence} \\
S_2 \quad \Uparrow \quad \text{Antidependence} \\
S_3
\]

Because the data-dependence graph has no cycles, it can be completely vectorized, although some statement reordering will be necessary. Since \(S_1 \neq S_2\), \(S_3\) must precede \(S_2\) in the vectorized loop:

\[
\begin{align*}
S_1 & : A(1:N) = B(1:N) \\
S_2 & : C(1:N) = A(1:N) + B(1:N) \\
S_3 & : E(1:N) = C(2:N+1)
\end{align*}
\]

The following loop contains a data-dependence cycle:

\[
\begin{align*}
d & \leftarrow 2,N \\
S_1 & : A(I) = B(I) \\
S_2 & : C(I) = A(I) + B(I-1) \\
S_3 & : E(I) = C(I+1) \\
S_4 & : B(I) = C(I) + 2.
\end{align*}
\]

The following is the data-dependence graph for this loop:

\[
S_1 \quad \Downarrow \quad \text{Flow dependence} \\
S_2 \quad \Uparrow \quad \text{Antidependence} \\
S_3 \\
S_4
\]

Statements \(S_1\) and \(S_2\) comprise a cycle in the data-dependence graph: when the dependence graph

\[
A \text{ first order linear recurrence is defined by the equations } a_i = a_{i-1} + x \times a_{i-2} \\
\text{ for } i \geq 2. \quad a_0 \text{ and } a_1 = 0
\]

\[a_{i} = a_{i-1} + x \times a_{i-2}
\]
contains a cycle, the strongly connected components (or maximal cycles) must be found. All the statements in a data-dependence cycle must be executed in a serial loop unless the cycle can be broken. However, other statements may still be vectorized. Thus, the previous loop may be partially vectorized as follows:

\[
\begin{align*}
S_1: & \quad A(2:N) = B(2:N) \\
S_2: & \quad E(2:N) = C(3:N+1) \\
& \quad \text{do } I = 2,N \\
S_3: & \quad C(I) = A(I) + B(I-1) \\
S_4: & \quad B(I) = C(I) + 2.
\end{align*}
\]

A sufficient (but not necessary) condition for vectorization of a loop is that no upward data-dependence relations \((S_i, b \prec S_j, \text{and } S_j \text{lexically precedes } S_i)\) appear in the loop. This will guarantee legal vectorization, but will miss loops where simple statement reordering would allow vectorization (as in the first example above).

Certain reduction operations appear frequently in programs and are recognized by vectorizing compilers. Prominent among these is the \textit{SUM} of a vector:

\[
\begin{align*}
& \quad \text{do } I = 1,N \\
S_1: & \quad A(I) = B(I) + C(I) \\
S_2: & \quad ASUM = ASUM + A(I) \\
\end{align*}
\]

Here, the \textit{SUM} function returns the sum of its argument. A special case of a \textit{SUM} is a dot product: this is important because many supercomputers have a distinct adder and multiplier that can perform a dot product \((\text{a SUM of a multiplication})\) in the same time as a \textit{SUM} alone, thus performing the multiplication with almost no time penalty.

Care must be taken by the compiler and the programmer to ensure that the correct answer will always result. Most methods to produce a \textit{SUM} using vector instructions involve accumulating partial sums, then adding the partial sums together. Since this will add the arguments in a different order than the original loop, round-off errors may accumulate differently, and some programs may produce substantially different answers. Some compilers have a switch that will disable generation of reductions in order to guarantee the same answers from the vectorized code as from the serial code.

Other common reductions are the \textit{PRODUCT} of a vector or maximum or minimum of a vector. Simple pattern recognition can be used to find these operations (as well as \textit{SUM}) in loops. More complicated patterns, such as a loop that finds the maximum of a vector and saves the index of the maximum, are also frequently found:

\[
\begin{align*}
\text{IMAX} = & 1 \\
\text{AMAX} = & A(1) \\
\text{do } I = & 2,N \\
& \text{if } (A(I) > \text{AMAX} ) \text{ then } \\
& \quad \text{AMAX} = A(I) \\
& \quad \text{IMAX} = I \\
& \text{end if }
\end{align*}
\]

Recognizing and generating vector code for such multistatement patterns enhance the overall power of a vectorizer.

At least one supercomputer has been designed with an instruction to solve first order linear recurrences. These recurrences can be recognized by pattern matching:

\[
\begin{align*}
& \quad \text{do } J = 2,N \\
& \quad A(J) = A(J-1) * B(J) + C(J)
\end{align*}
\]

Fast algorithms for solving recurrences on parallel computers have been devised that may be useful for some systems, although these suffer from the same round-off error accumulation problem mentioned earlier. Many computer systems offer a library of procedures to compute certain common forms of recurrences, some compilers recognize these recurrences and translate them into calls to the appropriate library procedure.

Loop Concurrentization

One way to use multiple processors in a computer is to partition the set of iterations of a loop and assign a different subset to each processor [16, 19, 34, 38, 42, 43]. Two important factors that determine the quality of the concurrent code are the balance of processor load, and the amount of processor idle time due to synchronization. Concurrentization, therefore, should aim at an even distribution of the iterations among processors and should try to organize the code so as to avoid synchronization or, at least, to minimize waiting.

Checking for independent iterations is done by examining the data-dependence directions. If all the data-dependence relations in a loop have an \(=\) direction for that loop, then the iterations are independent. The only time that communication between

---

*The Burroughs Scientific Processor (BSP) [30] was designed with a recurrence instruction, but the project was canceled before the first machine was delivered. Among current supercomputers, the Hitachi S-810 has a recurrence instruction.

1. In particular, the STAMLIB library for the Control Data 6600 [48], 7600, Cyber 70, Cyber 170, and Cyber 205 computers.
processors is necessary is when a data-dependence relation exists in the loop with a < direction for that loop. For instance, the loop

```
do I = 1,N
  S1: A(I,J) = B(I,J) + C(I,J)
do J = 2,N
  S2: C(I,J) = D(I,J) / 2
  S3: E(I,J) = A(I,J-1)**2 + E(I,J-1)
end do
end do
```

has the following data dependences:

- S, b. < S
- S, b = S
- S, b = S

Since all the data-dependence directions for the I loop are =, each iteration of the I loop can be executed in parallel. If N processors are available, each processor can execute one iteration of the loop. If fewer processors are available, the iterations can be folded onto the processors in one of several ways. The compiler can preschedule the iterations of the loop onto the P processors either in contiguous blocks

- processor 1 executes iterations 1, 2, ..., [N/P]
- processor 2 executes iterations [N/P]+1, ... 2[N/P]

or by assigning every Pth iteration to the same processor:

- processor 1 executes iterations 1, P+1, 2P+1, ...
- processor 2 executes iterations 2, P+2, 2P+2, ...

Alternatively, the processors can be self-scheduled [27, 47], meaning that each processor at the end of every iteration enters a critical section of code to determine what iteration of the loop it should execute next. Self-scheduling works well when the workload for each iteration is relatively large, but may vary between different iterations, perhaps due to conditional code in the loop.

Sometimes the iterations of a loop are not independent:

```
do I = 2,N
  S1: A(I) = B(I) + C(I)
  S2: C(I) = D(I) * 2.
  S3: E(I) = C(I) + A(I-1)
end do
```

The data-dependence relations for this loop are the following:

- S, b. < S
- S, b. < S
- S, b. < S

Since there is a < data-dependence direction for this loop, the iterations cannot be executed independently. If the different iterations are to be executed in parallel, the processor executing iteration i must not fetch the value for A(i-1) in statement S before the processor executing iteration i-1 has stored the value of A(i-1) in statement S'. The code inside the loop with the synchronization added would appear as follows:

```
do I = 1,N
  S1: A(I) = B(I) + C(I)
  signal ( I )
  S2: C(I) = D(I) * 2.
  if ( I > 2 ) wait ( I-1 )
  S3: E(I) = C(I) + A(I-1)
end do
```

The compiler can sometimes reorder statements to reduce the effect of the required synchronizations. Weak data-dependence tests may add some synchronizations that are not really necessary, so good dependence testing is critical for good performance.

Recognition of simple reductions applies to concurrent loops as well as vector loops. One method to generate parallel code for the loop

```
do I = 1,N
  S1: A(I) = B(I) + C(I)
  S2: D(I) = A(I) * 2.
  S3: ASUM = ASUM + A(I)
end do
```

is to add synchronization (as above) around statement S, so that each processor p (of the P available processors) would execute the following loop:

```
do I = p,N,P
  S1: A(I) = B(I) + C(I)
  S2: D(I) = A(I) * 2.
  if ( I > 1 ) wait ( I-1 )
  S3: ASUM = ASUM + A(I)
  signal ( I )
end do
```

This method is simple to implement and will always result in the same answer as the original scalar code (which, again, can be an important factor due to round-off error accumulations). A faster method is to accumulate partial sums on each processor

```
ASUMX(p) = 0
do I = p,N,P
  S1: A(I) = B(I) + C(I)
  S2: D(I) = A(I) * 2.
  if ( I > 1 ) wait ( I-1 )
  S3: ASUMX(p) = ASUMX(p) + A(I)
end do
```

December 1986 Volume 29 Number 12
and then add the partial sums into the summation variable ASUM at the end of the loop. This produces completely parallel code without synchronization, but may accumulate different round-off errors.

**High-Level Spreading**

Another approach to utilizing multiple processors is to spread independent operations over several processors. A fine-grain parallelism model would spread the computation tree of an expression evaluation over several processors. For instance, the computation tree for the expression

\[ A + (B \times 2 - C \times 1.5) - (D - 2 \times E) \]

is as follows:

- **Level 1**
- **Level 2**
- **Level 3**
- **Level 4**
- **Level 5**

In the computation tree at levels 2, 3, and 4, there are two arithmetic operations that could be performed simultaneously. With two processors, this expression could be computed in four time steps (assuming + and \( \times \) take the same time and assuming no time for variable fetching or processor communication), instead of the seven time steps necessary for a single processor.

The problem with this model is that the assumptions are invalid for current machines; variable fetching does take time, and if several processors share a common memory, they can interfere with each other. Communication between processors also takes time; thus, it is better to try to spread blocks of code that represent a relatively large workload. As with concurrent loops, it is best if the blocks of code assigned to different processors are completely independent, so that no communication costs are incurred. If the workload is spread unevenly over the processors, then some processors may finish early and be left idle while other processors are still busy.

Adjacent blocks of code, such as adjacent independent recurrence loops or procedure calls, are candidates for high-level spreading [50]. The data-dependence relations between these high-level objects are examined, and if the blocks of code are independent, then they can be spread over several processors. If there are data-dependence relations between the blocks of code, then either synchronization must be added (as in loop concurrentization) to perform spreading, or the code must be executed serially.

The potential for speedup with spreading is much lower than for loop concurrentization. Loop concurrentization may find \( N \) independent blocks of code, where \( N \) is the loop bound. Spreading, in practice, will usually find only 2–3 independent blocks of code suitable for parallel execution.

**Trade-offs between Vectorization and Concurrentization**

Some recent computer designs (e.g., the Cray X-MP, Alliant FX/8, and ETA10) have multiple processors with vector instructions. The techniques of vectorization and concurrentization must be used together to take full advantage of these computers. When only one loop exhibits any parallelism, the compiler must decide whether to generate vector code, concurrent code, or whether to split the index set into segments, all of which can be executed concurrently and in vector mode. When the loop contains many if statements that would produce sparse vector operations, concurrent execution may be more efficient.

When several nested loops exhibit parallelism, the compiler must choose which loop to execute in vector mode and which in concurrent mode. Several factors should be considered. For example, since the vector speed of some machines depends on the stride of memory accesses, the compiler may choose to execute the loop that generates stride-1 memory operations in vector mode and some other loop in concurrent mode. However, since a \( < \) data-dependence direction implies that a synchronization would be required for concurrent execution, the compiler may attempt to execute a loop with a \( < \) direction in vector mode, and to choose a loop with all \( = \) directions for concurrent execution. This trade-off is illustrated in the sample loop below:

```fortran
DO J = 1, N
  DO I = 1, N
    A(I, J+1) = B(I, J) + C(I, J)
  END DO
END DO
```

This loop can be compiled with the 1 loop in vector mode, which will generate stride-1 memory operations (assuming Fortran column-major storage)
and with the J loop in concurrent mode, as follows:

```plaintext
doacross J = 1,N
S_1: A(1:N,J+1) = B(1:N,J) + C(1:N,J)

signal ( J )
if ( J > 1 ) wait ( J-1 )
end doacross
```

This may produce the best vector execution speed, but the data-dependence relation \( S_1 \) \& \( S_2 \) requires synchronization in the concurrent loop. An alternate method to compile the loop would perform the J loop in vector mode:

```plaintext
doall I = 1,N
S_1: A(I,2:N+1) = B(I,1:N) + C(I,1:N)
end doall
```

Although this loop would have better concurrent execution speed, it would perhaps be at the expense of slower vector execution. Balancing the different methods of compiling the loop to get the best performance is a tough job for the compiler.

### IMPROVING POTENTIAL PARALLELISM

The previous sections should clarify the importance of a good data-dependence testing procedure. If unnecessary relations are added to the data-dependence graph, then the potential for parallelism discovery can be reduced dramatically. Some methods for computing a more accurate data-dependence graph are given here.

#### Induction Variable Recognition

Variables in loops whose successive values form an arithmetic progression are called induction variables; the most obvious example of an induction variable is the index variable of a loop. Induction variables are often used in array subscript expressions. Traditional optimization techniques are aimed at finding induction variables to remove them from the loop and also to optimize the array address calculation \[2\]. For data-dependence tests, the array subscripts should be known in terms of the loop index variables; therefore, discovery of induction variables is important. Most compilers will recognize that the loop

```plaintext
INC = N
do I = 1,N
12 = 2*I-1
X(INC) = Y(I) + Z(12)
INC = INC - 1
end do
```

refers to the same array addresses as the following loop:

```plaintext
do I = 1,N
X(N-I+1) = Y(I) + Z(2*I-1)
end do
```

The expression assigned to 12 is recognized as a function of the loop index variable, so 12 is easily recognized as an induction variable. The assignment to INC is a self-decrement, which qualifies INC as an induction variable. If the last values of 12 and INC are not used later in the program, the two loops above may be used interchangeably. After vectorization, both loops become

\[
X(N:1:-1) = Y(1:N) + Z(1:2*N-1:2)
\]

#### Wraparound Variable Recognition

Sometimes a variable may look like an induction variable, but does not quite qualify. The assignment to J in the loop

```plaintext
J = N
do I = 1,N
B(I) = (A(J) + A(I)) / 2.
J = I
end do
```

appears to qualify J as an induction variable, but J is used before it is assigned. In fact, the programmer used a trick to make the array A look like a cylinder. The loop takes the average of two adjacent elements of the array A; in the first iteration, the neighbor of A(1) is defined to be A(N)—the J variable accomplishes this trick. J is called a wraparound variable, since the values assigned to it are not used until the next iteration of the loop.

By peeling off one iteration of the loop, J can be treated as a normal induction variable:

```plaintext
if (N >= 1) then
    B(1) = (A(1) + A(N)) / 2.
    do I = 2,N
        B(I) = (A(I-1) + A(I)) / 2.
    end do
end if
```

The if is necessary to test the zero-trip condition of the loop. The loop may be vectorized to become

```plaintext
if (N >= 1) then
    B(1) = (A(1) + A(N)) / 2.
end if
```

#### Symbolic Data-Dependence Testing

As mentioned in the data-dependence section, the simplest data-dependence subscript tests will be sufficient for a large number of cases, but are too lim-
itted for a general-purpose powerful compiler. Even some of the more sophisticated tests have severe restrictions, such as requiring that the loop bounds be compile-time constants. To handle a large majority of cases, a compiler must be able to compute precise data-dependence relations for very general array references. For instance, in the following loop

\[
\begin{align*}
\text{do } I &= \text{ LOW, IGH} \\
S_1 &: \quad A(I) = B(I) + C(I) \\
S_2 &: \quad D(I) = A(I-1) \\
\end{align*}
\]

the data-dependence relation \( S_1 \prec S_2 \) can be computed using the simplest tests. However, in the similar loop

\[
\begin{align*}
\text{do } I &= \text{ LOW, IGH, INC} \\
S_1 &: \quad A(I) = B(I) + C(I) \\
S_2 &: \quad D(I) = A(I-INC) \\
\end{align*}
\]

the same relation \( S_1 \prec S_2 \) holds, but is more difficult to detect, since the values of LOW, IGH, and INC are all unknown to the compiler, and even the sign of INC is unknown (a negative increment would make the loop go backwards). The following is another case in which symbolic data dependence (so called because the subscript expression cannot be decomposed into compile-time constants) is needed:

\[
\begin{align*}
\text{do } I &= 1, N \\
S_1 &: \quad A(LOW+I-1) = B(I) \\
S_2 &: \quad B(I+N) = A(LOW+I) \\
\end{align*}
\]

Here, the two references to the array \( A \) can be compared by canceling out the loop-invariant value LOW. This is then the same as comparing \( A(I-1) \) to \( A(I) \), which can be handled by simpler tests. The two \( B \) array references cause no data dependence, since the section of the array referenced by \( B(I) \) is \( B(1:N) \), which does not intersect with the section of the array referenced by \( B(I+N) \), namely \( B(N+1:N+N) \).

**Global Forward Substitution**

*Global forward substitution* is a transformation that substitutes the right-hand side of an assignment statement for occurrences of the left-hand-side variable, which is especially useful in conjunction with symbolic data dependence. In programs, temporary variables are frequently used to hold commonly used subexpressions or offsets; these variables appear later in the program in array subscripts. Without some kind of global knowledge, the data-dependence tests must assume that the set of subscript values might intersect. For example, the program

\[
\begin{align*}
\text{NP1} &= N+1 \\
\text{NP2} &= N+2 \\
\vdots \\
\text{do } I &= 1, N \\
S_1 &: \quad B(I) = A(NP1) + C(I) \\
S_2 &: \quad A(I) = A(I) - 1. \\
\text{do } J &= 2, N \\
S_1 &: \quad D(J,NP1) = D(J-1,NP2)*C(J) + 1. \\
\end{align*}
\]

defines two variables, \( \text{NP1} \) and \( \text{NP2} \) in terms of \( N \). A loop, later in the program, uses \( \text{NP1} \) and \( \text{NP2} \) in array subscripts. If the compiler does not keep any information about \( \text{NP1} \), then it must assume that the assignment of \( A(I) \) might reassign \( A(\text{NP1}) \), and thus there is dependence between \( S_1 \) and \( S_2 \). However, \( \text{NP1} \) was defined to be \( N+1 \), and the assignment to \( A(I) \) will not ever reach \( A(N+1) \), so this is a false dependence. Similarly, if the compiler does not keep information about \( \text{NP1} \) and \( \text{NP2} \), it must assume that \( S_1 \) forms a recurrence. In fact, since \( \text{NP1} \) can not equal \( \text{NP2} \), the two references to the \( D \) array in \( S_1 \) are independent, and the \( J \) loop can be vectorized or concurrentized. Many compilers perform constant propagation [26, 45, 51], which is a special case of global forward substitution.

**Semantic Analysis**

Semantic analysis of the program can also help remove data-dependence relations. For instance, in the loop

\[
\begin{align*}
\text{do } I &= \text{ LOW, IGH} \\
S_1 &: \quad A(I) = B(I) + A(I+M) \\
\end{align*}
\]

\( S_1 \) can be vectorized if \( M \geq 0 \), but not if \( M < 0 \). By looking at the surrounding code, the compiler might find an if statement:

\[
\begin{align*}
\text{if ( } M > 0 ) \text{ then} \\
\text{do } I &= \text{ LOW, IGH} \\
S_1 &: \quad A(I) = B(I) + A(I+M) \\
\end{align*}
\]

Taking the if statement into account, the loop will not be executed unless \( M > 0 \); therefore, it can be vectorized. Note that since the iterations are not independent, concurrent code for this loop may still not be appropriate.
Sometimes, even if the if statement is not present, the program can be modified to achieve a result similar to the previous transformation. The original loop could be transformed into the following:

```plaintext
if ( M>=0 ) then
   do I = LOW,IGH
      A(I) = B(I) + A(I+M)
   end do
else
   do I = LOW,IGH
      A(I) = B(I) + A(I+M)
   end do
end if
```

The then part of the if statement can now be vectorized. This transformation is called two-version loops for obvious reasons.

An alternative to semantic analysis is for the compiler to request information from the user. If the user inputs M \geq 0 as an assertion that is true immediately before beginning of execution of the original loop, then the compiler can proceed to vectorize the loop. (Assertions are discussed in greater detail in the section on interaction with the programmer, p. 1198.)

Semantic analysis can also be used to generate vector code for the following loop:

```plaintext
do I = LOW,IGH
S1: A(I) = A(M)
end do
```

Since the relative values of LOW, IGH, and M are not known, the data dependence S₁, δ₁ S₁, relation must be assumed. This loop can, however, be executed with a vector instruction. The value A(M) is loop invariant, even if M falls between LOW and IGH. The value A(M) will at worst be copied to itself; it will not change. The vector code for this statement will give the same results as the serial loop, but the entire statement must be examined to prove this.

**Interprocedural Dependence Analysis**

When a procedure or function call appears in a loop, most compilers will assume that the loop must be executed serially. Analysis of the effects of the procedure or function call, including which parameters are changed and what global variables are used or changed, can allow dependence testing to decide whether or not the procedure call prevents parallel code from being generated. Studying other information about the parameters, such as values of constants, across procedure call boundaries can help the compiler optimize the code [9, 12, 14, 49].

An alternative method for handling procedure calls is to expand the procedure in-line (also called procedure integration) [35]. This makes possible the application of some transformations that simultaneously manipulate code in the calling and in the called routines. Also, dependence analysis for the subroutine body is more exact, since only the effects of the one call must be taken into account, and the overhead of the subroutine call is eliminated. In-line expansion is useful even for serial computers. However, in-line expansion should be done with care to avoid an undue increase in the time required for compilation.

**Removal of Output and Antidependences**

Output dependences and antidependences are, in some sense, false dependences. They arise not because data are being passed from one statement to another, but because the same memory location is used in more than one place. Often, these false dependences can be removed by changing variable names or copying data.

**Variable Renaming.** Renaming introduces new variable names to replace some of the occurrences of the old variables throughout the program.

In the following program segment:

```plaintext
S₁: A = B + C
S₂: D = A + 1
S₃: A = D + E
S₄: F = A - 1
```

variable A is assigned twice. This produces the output dependence S₁, δ₀ S₁, and the antidependence S₁, δ S₁. Both of these dependences disappear if a new variable replaces A in S₁ and S₄:

```plaintext
S₁: A₂ = B + C
S₂: D = A₂ + 1
S₃: A = D + E
S₄: F = A - 1
```

Renaming arrays is a difficult problem in general. Current compilers rename arrays only in very limited cases, if at all.

**Node Splitting.** Some loops contain data-dependence cycles that can be easily eliminated by copying data. The following loop:

```plaintext
do I = 1,N
S₁: A(I) = B(I) + C(I)
S₂: D(I) = (A(I) + A(I+1)) / 2.
end do
```

*Example 1: a loop surrounding a subroutine invocation and a loop in the body of the routine could be interchanged (see Note 3).*
has the following data-dependence graph:

```
    S1
   / \    \\
   /   \   \\
  S   S  \  \ \\
```

This appears to be a data-dependence cycle. However, one of the arcs in the cycle corresponds to an antidependence; if this arc were removed, the cycle would be broken. The antidependence relation can be removed from the cycle by inserting a new assignment to a compiler temporary array as follows:

```
do I = 1,N  
S,: ATEMP(I) = A(I+1)  
S,: A(I) = B(I) + C(I)    
S,: D(I) = (A(I) + ATEMP(I)) / 2.  
end do  
```

The modified loop has the following data-dependence graph:

```
    S'1
   /   \\
   /     \\
  S   S  \\
```

The data-dependence cycle has been eliminated by "splitting" the S node in the data-dependence graph into two parts; the new loop can now be vectorized:

```
S': ATEMP(1:N) = A(2:N+1)  
S,: A(1:N) = B(1:N) + C(1:N)  
S,: D(1:N) = (A(1:N) + ATEMP(1:N)) / 2.  
```

A similar technique can be used to remove output dependences in data-dependence cycles. The loop

```
do I = 1,N  
S,: A(I) = B(I) + C(I)  
S,: A(I+1) = A(I) + 2*D(I)  
end do  
```

has the following data-dependence graph:

```
    S1
   /   \\
   /     \\
  S    S  \\
```

Flow dependence

```
S    
```

Output dependence

The data dependence cycle can be broken by adding a temporary array:

```
do I = 1,N  
S,: ATEMP(I) = B(I) + C(I)  
S,: A(I+1) = ATEMP(I) + 2*D(I)    
S,: A(I) = ATEMP(I)  
end do  
```

The data-dependence graph for the modified loop is

```
    S1
   /   \\
   /     \\
  S    S  \\
```

which has no cycles, and can now be vectorized:

```
S,: ATEMP(1:N) = B(1:N) + C(1:N)  
S,: A(2:N+1) = ATEMP(1:N) + 2*D(1:N)  
S,: A(1:N) = ATEMP(1:N)  
```

In both of these cases the added cost is a copy either to or from a compiler temporary array. For machines with vector registers, however, the temporary array will be assigned to a vector register. The "extra" copy is just a vector register load or store that needs to be done anyway; proper placement of the load or store will allow vectorization of these loops without additional statements.

OPTIMIZATIONS FOR VECTOR OR CONCURRENT COMPUTERS

Many of the optimizations in this section were designed with vector or concurrent computers in mind. These optimizations are used to exploit more parallelism or to use parallelism more efficiently on the target computer.

Due to space limitations, the list of methods we discuss is incomplete. Among the absentees are the methods that deal with while loops and with recursion. Only recently in the context of developing optimizing compiler methods for Lisp has recursion been carefully considered [22, 23]. while loops, on the other hand, are hard to manipulate, and the known transformation techniques are successful only in limited cases.

Scalar Expansion

Vectorizing compilers will promote (or expand) scalars that are assigned in loops into temporary
arrays. This is clearly necessary in order to generate vector code. For instance, the loop

\[
\begin{align*}
do \ I = 1, N \\
S_1: & \quad X = A(I) + B(I) \\
S_2: & \quad C(I) = X ** 2 \\
end \ do
\end{align*}
\]

can be vectorized by first expanding \(X\) into a temporary array, \(XTEMP\).

\[
\begin{align*}
& \text{allocate} \ (XTEMP(1:N)) \\
do \ I = 1, N \\
S_1: & \quad XTEMP(I) = A(I) + B(I) \\
S_2: & \quad C(I) = XTEMP(I) ** 2 \\
end \ do \\
& \quad X = XTEMP(N) \\
& \text{free} \ (XTEMP)
\end{align*}
\]

and then generating vector code:

\[
\begin{align*}
& \text{allocate} \ (XTEMP (1:N)) \\
do \ I = 1, N \\
S_1: & \quad XTEMP (1:N) = A(1:N) + B(1:N) \\
S_2: & \quad C(1:N) = XTEMP(1:N) ** 2 \\
& \quad X = XTEMP(N) \\
& \text{free} \ (XTEMP)
\end{align*}
\]

The explicit \textit{allocate} and \textit{free} are not necessary on many computers, such as those with vector registers, since the temporary array will exist only in the registers.

When the target machine is a multiprocessor, there is another alternative. Multiprocessor languages (such as Cedar Fortran\textsuperscript{11} and Blaze [37]) allow the declaration of iteration-local variables. Thus, the loop

\[
\begin{align*}
doall \ I = 1, N \\
& \quad \text{real} \ X \\
& \quad X = A(I) + B(I) \\
& \quad C(I) = X ** 2 \\
end \ doall
\end{align*}
\]

is another valid transformation of the previous loop, as long as \(X\) is not used outside the loop in the original program. The \textit{real} \(X\) declaration in the \textit{doall} loop means that there will be a separate copy of \(X\) for each iteration of the loop. Declaring a scalar variable as iteration local has the same effect as transforming the scalar into an array.

**Loop Interchanging**

In a multiply-nested loop, the order of the loops may often be interchanged without affecting the outcome [6, 52]. For instance, the loop

\[
\begin{align*}
do \ J = 1, N \\
do \ I = 2, N \\
& \quad A(I, J) = A(I-1, J) + B(I) \quad \text{(L1)} \\
end \ do \\
end \ do
\end{align*}
\]

is equivalent to the following loop:

\[
\begin{align*}
do \ I = 2, N \\
do \ J = 1, N \\
& \quad A(I, J) = A(I-1, J) + B(I) \quad \text{(L2)} \\
end \ do \\
end \ do
\end{align*}
\]

Loop interchanging can translate either of the previous two loops to the other. Interchanging loops is not always possible. The following loop is an example of a loop that cannot be interchanged:

\[
\begin{align*}
do \ K = 2, N \\
do \ L = 1, N-5 \\
& \quad A(K, L) = A(K-1, L+5) + B(K) \\
end \ do \\
end \ do
\end{align*}
\]

Figure 1 (p. 1197) illustrates how data-dependence graphs are used to determine when loop interchanging is valid.

Loop interchanging may be used to aid in loop vectorization. For instance, L1 (above) computes a linear recurrence in its inner loop; interchanging it to create L2 allows vectorization of the \(J\) index:

\[
\begin{align*}
do \ I = 2, N \\
& \quad A(I, 1:N) = A(I-1, 1:N) + B(I) \\
end \ do
\end{align*}
\]

Loop interchanging may also be used to put a concurrent loop on the outside, leading to a better program after loop concurrentization. Thus, loop L2 could be interchanged into L1, which would then be concurrentized to become the following:

\[
\begin{align*}
doall \ J = 1, N \\
do \ I = 2, N \\
& \quad A(I, J) = A(I-1, J) + B(I) \\
end \ do \\
end \ doall
\end{align*}
\]

**Fission by Name**

Techniques have been developed to handle virtual memory systems, cache memories, and register allocation. The \textit{fission-by-name} transformation tries to break a single \textit{DO} loop into several adjacent loops. Two statements in the original loop will be in the same resulting loop if there is at least one variable or array referenced by both statements. Fission by name (originally called "distribution of name parti-
Loop Fusion
Loop fusion is a conventional compiler optimization \cite{3, 35} that transforms two adjacent loops into a single loop. The use of data-dependence tests allows fusion of more loops than is possible with standard techniques. For example, the loops
\[
\begin{align*}
\text{do } I = 2, N \\
S_1: & \quad A(I) = B(I) + C(I) \\
& \text{end do} \\
\text{do } I = 2, N \\
S_2: & \quad D(I) = A(I-1) \\
& \text{end do}
\end{align*}
\]
would not be fused by conventional compilers that do not study the array subscripts. However, the loop fusion is legal since the data-dependence relation \( S_1 \delta S_2 \) would not be violated:
\[
\begin{align*}
\text{do } I = 2, N \\
S_1: & \quad A(I) = B(I) + C(I) \\
& \text{end do} \\
S_2: & \quad D(I) = A(I-1) \\
& \text{end do}
\end{align*}
\]
A slightly modified example shows when loop fusion is not legal:
\[
\begin{align*}
\text{do } I = 2, N \\
S_1: & \quad A(I) = B(I) + C(I) \\
& \text{end do} \\
\text{do } I = 2, N \\
S_2: & \quad D(I) = A(I+1) \\
& \text{end do}
\end{align*}
\]
In the original two loops, the data-dependence relation \( S_1 \delta S_2 \) holds; the fused loop below, however, has the relation \( S_1 \delta S_2 \):
\[
\begin{align*}
\text{do } I = 2, N \\
S_1: & \quad A(I) = B(I) + C(I) \\
& \text{end do} \\
S_2: & \quad D(I) = A(I+1) \\
& \text{end do}
\end{align*}
\]
Loop fusion is used in conventional compilers to reduce the overhead of loops. Likewise, fusion helps to reduce start-up costs for doall loops. It may also increase overlapping if two doall loops require synchronization between iterations.

Since loop fusion and loop fission are dual transformations, any compiler that uses both of them should do so carefully. Loop fusion should not be used to fuse the loops just created by fission. Loop fusion can be used to combine separate loops, if they all refer to the same set of variables, with the same goal as fission by name. For instance, by fusing several loops that refer only to the arrays \( A, B, D \), the compiler would have a larger loop with the benefits of loop fusion, but still have the improved memory hierarchy performance that comes from only referring to a small set of arrays in the loop.

Strip Mining
Strip mining \cite{35} is used for memory management; it transforms a singly nested loop into a doubly nested one. The outer loop steps through the index set in blocks of some size, and the inner loop steps through each block. As an example, consider the following loop:
\[
\begin{align*}
\text{do } I = 1, N \\
A(I) & = B(I) + 1 \\
D(I) & = B(I) - 1 \\
& \text{end do}
\end{align*}
\]
After strip mining, this becomes the following:
\[
\begin{align*}
\text{do } J = 1, N, 32 \\
& \text{do } I = J, \text{MIN}(J+31, N) \\
A(I) & = B(I) + 1 \\
D(I) & = B(I) - 1 \\
& \text{end do} \\
& \text{end do}
\end{align*}
\]
Thus, the loop is excavated in chunks, just as a strip mine is excavated in shovelfuls.

The block size of the outer block loop \( 32 \) in this example is determined by some characteristic of the target machine, such as the vector register length or the cache memory size (see Figure 2, p. 1198, for an example of strip mining and fission by name). For vector machines, the inner strip loop will be vectorized; for parallel computers, the outer block loop can sometimes be concurrentized. Figure 3 (p. 1199) shows how strip mining and loop interchanging can be combined to optimize performance.

Loop Collapsing
Loop collapsing \cite{44, 52} transforms two nested loops into a single loop, which is used to increase the effective vector length for vector machines. Therefore,
\[
\begin{align*}
\text{real } A(5,5), B(5,5) \\
\text{do } I = 1, 5 \\
& \text{do } J = 1, 5 \\
A(I,J) & = B(I,J) + 2. \\
& \text{end do} \\
& \text{end do}
\end{align*}
\]
becomes as follows:
The compiler uses data dependences to determine when loop interchanging is valid. For example, the loops in (a) may not be interchanged. To see why, consider its iteration space in (b). The arrows in (b) show the data-dependence relations flowing across the iteration space. The instances of statements $s$ are executed by (a) in the order shown by the dotted arrows in (c). If the loops in (a) were interchanged, the new order of execution would be as shown in (d). In this loop ordering, $s^{1,2}$ would be executed before $s^{1,3}$, even though the $s^{1,2}$ needs a value computed by $s^{1,3}$; thus this statement ordering is invalid. As a second example, consider the loops in (e). In this case interchanging is clearly valid since no dependences are violated in the new execution order.

**FIGURE 1. How to Determine When Loop Interchanging Is Valid**

\[
\text{real } A(25), B(25) \\
do \text{ IJ } = 1, 25 \\
A(IJ) = B(IJ) + 2.
\]

A general version of loop collapsing is useful for parallel computers where only a single doall nest is supported or to improve the performance of self-scheduled loops. In general, this may require the introduction of some extra assignment statements. For instance, the loop

\[
\text{do } I=1,N \\
\text{do } J=1,M \\
A(I,J) = B(I,J) + 2.
\]
do I=1,N
A(I) = B(I) + C(I)
E(I) = F(I) + G(I)
D(I) = A(I) + B(I)
end do
(a)

do I=1,N
A(I) = B(I)
D(I) = A(I)
end do

do I=1,N
E(I) = F(I)
end do
(b)

do J=1,N,32
  do I=J,min(N,J+31)
    A(I) = B(I) + C(I)
    D(I) = A(I) + B(I)
  end do
end do

end do

end do
(c)

Fission by name and strip mining are used to improve memory performance. Assume a target computer with 32-element vector registers. Before register allocation is performed, the input loop (a) will go through the following sequence of transformations. First, fission by name is applied (b), then strip mining (c), and finally loop vectorization (d). The target program (d) is now a sequence of 32-element vector operations.

FIGURE 2. An Example of Fission by Name and Strip Mining

may be transformed into

\[
\text{do } L=1,N\times M \\
\quad I = \lfloor L/M \rfloor \times \text{mod}(L-1,M) + 1 \\
\quad J = \text{mod}(L-1,M) + 1 \\
\text{end do}
\]

regardless of the bounds of the array A.

INTERACTION WITH THE PROGRAMMER

Several user interaction strategies have been used by optimizing compilers for parallel computers. The most frequent approach is for the compiler to translate directly into object code and to provide a summary specifying what was vectorized and what was not. When something is not vectorized, the compiler gives a reason, which could be the presence of a data dependence, the need to assume a dependence because a subscript range is not known at compile time, the presence of a call to an unknown routine, and so on. If users are not satisfied with the outcome, they may resubmit the program after rewriting parts of it or after inserting directives or assertions.

For example, consider the following loop:

\[
\text{do } I=1,N \\
\quad A(K(I)) = A(K(I)) + C(I) \\
\text{end do}
\]

Most compilers will not vectorize this loop; instead they will notify the user that the compiler was forced to assume a dependence since it did not know the value of vector K at compile time. If programmers know that K is a permutation of a subset of the integers, they may order the compiler, through a directive, to vectorize the loop. This directive usu-
The translation of (a) illustrates two interesting uses of loop interchanging. Loop (a) could be vectorized in the form in which it is presented, but it would require a SUM, which may cause round-off error problems. The loops can be interchanged so that I becomes the innermost index (b). After strip mining (c), the innermost loop is vectorized (d). Notice that thanks to loop interchanging the elements of the vectors in (d) are in contiguous memory locations. Vector register assignment may now be performed, leading to (e). The block loop I may be interchanged to become the outermost loop (f). We can now illustrate a second consequence of loop interchanging. The vector register load \( vr1 \leftarrow C(L:1,J) \) and store \( C(L:1,J) \leftarrow vr1 \) in (e) are loop invariant and may be moved outside the innermost loop (g). Loop interchanging to increase the number of loop invariant register loads and stores is also a useful sequential optimization technique.

FIGURE 3. Two Examples of Loop Interchanging
ally takes the form of a comment line preceding the loop.

Another way for the user to supply information to the compiler is through assertions. This is sometimes provided as an alternative to compiler directives. For example, in the previous loop the programmer could have asserted that K is a permutation of a subset of the integers. Assertions have two advantages over directives. They are self-explanatory, and they can be tested at run time while debugging the program. Directives, on the other hand, may be a simpler way or even the only way to specify what the user wants. For example, directives may be the only way for the user to request that some part of the code be executed sequentially.

A second user interaction strategy is to produce a restructured source program with vector and/or concurrent language extensions.12 This strategy makes it possible for the programmer to learn how a program was translated without having to look at the assembly code. These restructurers, like the compilers discussed above, accept directives and assertions from the user.

Experimental interactive restructurers, such as the Blaze Restructurer at Indiana University and the R* programming environment at Rice University, are being developed. Some of these restructurers rely on the user to specify the transformations, and users may specify interactively what scalars to expand, what loops to interchange, which ones to vectorize, and so on. These tools make the process of hand-rewriting programs highly reliable and may become a useful tool for program development.

Commercial interactive vectorizers are already available.13 These interactive tools allow users to find the most time-consuming parts of their programs and rewrite them. They will also aid users in writing vectorizable code in order to get the best performance.

SUMMARY
When the Cray-1 was first delivered to Los Alamos Scientific Laboratories in 1976, there was a great deal of skepticism about whether compiler technology would ever catch up with hardware. Much research had been done for the Texas Instruments ASC and for the Illiac IV on automatic detection of parallelism, but it did not yet meet the needs of the user community. Since that time, several vector supercomputers and minisupercomputers have been announced, each with its own vectorizing compiler.

Some of these compilers are more powerful than others, but all are much better than anything hoped for in 1976.

Now a new generation of computers with multiple processors is coming. Compilers to detect parallelism suitable for spreading over many processors already exist, and more will follow. Many of the optimizations that were used for vectorization are also useful for multiprocessing. That so many of the ideas can be shared by different architectures will make it easier to write compilers for new machines as time passes.

Acknowledgments. The authors would like to thank U. Banerjee, W. L. Harrison, A. Videnbaum, and the referees for their many helpful comments about this article.

Further Reading
Optimizing compiler algorithms. See [5], [28], [29], [33], [41], and [52].

Deflection Analysis. See [8]-[10].

REFERENCES
Abstract. The CONVEX FORTRAN 5.0 compiler translates FORTRAN source programs into efficient object code for execution on the CONVEX C2 parallel supercomputer. The compiler back end consists of an optimizer and code generator that are also used by CONVEX vector Ada (vada) and vector C (vc) compilers. Generation of efficient object code for the C2 requires the compiler to automatically vectorize and parallelize looping constructs. The compiler also performs many familiar scalar optimizations at both the local and global level plus some new optimizations for masked vector code. Many of the optimizations and transformations performed are facilitated by a unique dataflow representation that removes all induction values from loops. The compiler's code generator performs global register allocation, tree-height reduction, and code scheduling to take advantage of the C2's multiple, pipelined function units and vector chaining.

THE C2 ARCHITECTURE

The CONVEX C2 supercomputer is a parallel, vector multiprocessor. It is capable of multiprocessing, tightly-coupled parallel processing, or both simultaneously. Processors are switched rapidly from one process to another, automatically by hardware, and without operating system intervention. Each processor of the C2 is an integrated vector and scalar processor that is upward compatible at the binary level with CONVEX's earlier supercomputers, the C1 and the C1-XP.

Parallel Architecture

Up to four CPUs and an I/O processor are connected by a 5x5 crossbar switch to a 5-ported global memory. Each processor is capable of issuing one memory request per 40ns clock cycle, achieving a maximum aggregate bandwidth of 1 gigabyte per second. Up to 2 gigabytes of dynamic RAM memory is interleaved up to 64 ways to more than accommodate the bandwidth and to reduce bank conflict. 1024 32-bit communication registers accessible on a separate communications bus provide the means for synchronizing processors. The communication registers are divided into eight 128 element sets, allowing one set for each of four running processes and up to four processes waiting for resources.

A process is the active image of a program. A thread is an independent stream of execution through a process's address space. The operating system initiates the execution of the first thread in a Unix process. Subsequent execution of a spawn instruction by the process copies the private state of the process into 4 of the communication registers and requests idle processors to enter into the execution of the process on newly created threads. When an idle processor sees such a request, it loads its CIR (communication index register) with the index of the communication register set, and copies into its private state the same 4 communication registers to create a new thread. A join instruction terminates the thread that executes it and returns the thread's processor to the idle state, unless the thread is the last thread in the process. The join instruction has no effect in the final thread.

The spawn and join instructions are handled entirely by hardware, requiring only a few cycles to execute. Idle processors require 40 cycles (best case) to initiate a new thread, but these are cycles that would otherwise often be wasted.

Once processors are executing in parallel, their actions can be synchronized through fetch-and-add, lock/unlock, and send/receive instructions that allow communication registers or memory locations as operands.
Processor Architecture

Each processor within the C2 is a pipelined scalar/vector processor with a direct mapped logical cache of 1024 32-bit words and a 1024 entry translation cache for virtual addressing. The register set consists of 8 32-bit address (A) registers, 8 64-bit scalar (S) registers, and 8 128-element vector (V) registers with 64-bits per element. Vector registers are arranged in four banks of two registers each. Each bank can sustain two reads and one write simultaneously without performance degradation. VL, VS, and VM registers are the number of elements of a vector register to be used in vector operations, VS is the stride in bytes used for vector memory references, and VM holds a 128-bit mask that is typically set by vector comparison instructions.

Scalar instructions operate on scalar and address registers, and vector instructions operate on the vector registers. The vector instructions generalize to vectors the scalar arithmetic, relational, boolean, and load/store instructions. There are also vector instructions for compressing, merging, and masking vectors based on a vector of addresses, and for performing the common vector reductions, such as vector sum, vector product, etc. Vectors are compressed, merged or masked under control of VM. Compression shortens a vector by selecting only elements corresponding to 1-bits in VM. Merging combines two shorter vectors into a longer vector by selecting elements from one vector or the other based on the appropriate bit of VM. Masking combines two equal length vectors under control of VM to yield an equal length result vector. Most C2 vector instructions also have operation-under-mask counterparts. In such operations, VM is used to suppress operation on elements that might otherwise cause hardware exceptions.

COMPILER GOALS

Our most important goal was the automatic generation of code to exploit the C2's architecture. Primarily, this meant that the compiler had to automatically vectorize and parallelize most loops or loop nests in a source program without requiring any modifications to the source. It also meant that we had to generate instructions that were not related to parallelization either because they were new to C2 or were C1 instructions the compiler had just never supported. These included machine instructions for intrinsic functions, for vector converts, for vector compressing and merging, and for performing vector operations under mask.

Given the hardware design, the number of processors assigned to a process varies depending on the system load and innumerable other factors. It was therefore necessary that parallel code be independent of the number of processors allocated for its execution.

Other goals included improved optimization, improved code generation, and faster compilation. These goals led to a decision to throw away the old optimizer, rewrite it from scratch, and make major modifications to the existing code generator.

Our final goal was to have a single compiler for the C1 and C2 series and a single back end for FORTRAN, C, and ADA. Common compilers and a common back end reduce maintenance costs and increase the reliability of all derivative compilers.

COMPILER STRUCTURE

The compiler consists of three main phases, a language specific front end, an optimizer, and a code generator. The optimizer and code generator together form a common language back end. The optimizer is largely machine independent. Machine dependencies are isolated in the code generator. Front ends for C, ADA, and FORTRAN have now been implemented. Each compiler has roughly the same ability to vectorize and parallelize programs except the ADA compiler front end currently does not take advantage of automatic parallelization. The structure of the compiler family is illustrated in Fig. 1.
Front ends lex, parse and analyze source code, producing a language independent list of trees, which is then fed to the optimizer. Declarative information is passed to the back end in the form of a symbol table. Subscript expansion has not been performed at this point, because it is simpler to perform the necessary dependency analysis of array references in the back end before arrays have been linearized. For example, it is easier to analyze \( A(I,J) \) than \( A(I-1+N^*(J-I)) \).

The optimizer transforms the list of trees into a control flow graph in which basic blocks are represented as DAGs, that is, as directed acyclic graphs. Local and global optimizations are performed on this structure, then the graph is further transformed to an acyclic graph. Vectorization and parallelization are performed on the acyclic graph, then the graph is transformed back to normal form. A final round of local and global transformations is then performed and the graph is passed to the code generator.

Finally, the code generator assigns registers and opcodes to nodes, schedules them, and emits object code.

**REORDERING TRANSFORMATIONS**

The most important transformations performed by the 5.0 back end are reordering transformations. Reordering transformations do not eliminate operations from a program or replace them by simpler operations, but rearrange them so they can be more efficiently executed. (Here, operation means an execution instance of an instruction.) These transformations are made possible by a sophisticated dependency analysis that determines what constraints are required on the order in which operations are executed to guarantee fidelity to the program's original intent. Reordering transformations performed by the 5.0 optimizer include:

1. vectorization,
2. partial vectorization,
3. parallelization,
4. loop distribution, and
5. loop interchange.

These transformations are orchestrated by the 5.0 compiler to attain a high degree of efficiency in generated code.

**Dependency Analysis**

A dependency is a constraint on the order of execution of two operations. A reordering transformation is valid if the new order conforms to all the dependencies of the old order. Dependency analysis discovers and represents the dependencies and potential dependencies within a program. The 5.0 compiler uses Banerjee's inequality and the GCD test in dependency analysis as described in [1] with minor modifications. These tests are augmented with a few ad-hoc tests, and complemented by symbolic manipulation.

**Vectorization**

Vectorization replaces a loop that uses scalar instructions with a loop (or basic block) that uses vector instructions. For example,

```fortran
DO 10 I=1,512
10 A(I)=B(I)+C(I)
```

becomes, in FORTRAN 8x notation,

```fortran
DO 10 I=1,512,128
10 A(I:1+127)=B(I:1+127)+C(I:1+127)
```

Where formerly elements were loaded into scalar registers, operated upon, and stored one at a time, now a single vector instruction loads 128 elements of B into a vector register, another loads 128 elements of C, a third adds the two vectors, and a fourth vector instruction stores the resulting vector into A. The outer loop here, called a strip-mine loop, is introduced to accommodate the 128-element limit on vector register length.

Vectorization is possible when a loop has no recurrences. A recurrence is present when an operation on one iteration depends on an operation corresponding to the same instruction but executed on an earlier iteration. For example,

```fortran
DO 10 I=3,N
10 A(I)=A(I-2)*A(I-1)
```

here the store into \( A(I) \) on the second iteration depends on the load of \( A(I-1) \), which in turn depends on the store of \( A(I) \) of the first iteration.

The 5.0 compiler has a powerful vectorization capability that uses general algorithms rather than a pattern matching approach. The only pattern matching performed by the compiler currently is to recognize recurrences that can in fact be vectorized but not by straightforward methods. The vectorizer can handle arbitrary
branching within loops, reorder statements, and expand scalars. For example,

\[
\begin{align*}
\text{DO } 10 & \text{ I=1,N} \\
\text{IF(A(I).LT.B(I))GO TO 20} \\
Z(I) & = T \times G(I) \\
\text{GOTO 30} \\
20 & \\
Z(I) & = T \times H(I) \\
30 & \\
T & = X(I) \\
10 & \text{ CONTINUE}
\end{align*}
\]

is vectorized, in effect, as if it had been written

\[
\begin{align*}
\text{TEMP(1)=T} \\
\text{DO } 10 & \text{ I=1,N} \\
\text{TEMP(I+1)=X(I)} \\
T & = X(I) \\
\text{IF(A(I).LT.B(I))GO TO 20} \\
Z(I) & = \text{TEMP(I)} \times G(I) \\
\text{GOTO 10} \\
20 & \\
Z(I) & = \text{TEMP(I)} \times H(I) \\
10 & \text{ CONTINUE}
\end{align*}
\]

As a comparison to other vectorizers, the following table shows the results of several well-known FORTRAN compilers on the 24 Livermore Loops test:

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Cft</th>
<th>Fujitsu</th>
<th>5.0</th>
<th>Optimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
</tr>
<tr>
<td>2</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
</tr>
<tr>
<td>3</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
</tr>
<tr>
<td>4</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
</tr>
<tr>
<td>5</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
</tr>
<tr>
<td>6</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
</tr>
<tr>
<td>7</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
</tr>
<tr>
<td>8</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
</tr>
<tr>
<td>9</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
</tr>
<tr>
<td>10</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
</tr>
<tr>
<td>11</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
</tr>
<tr>
<td>12</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
</tr>
<tr>
<td>13</td>
<td>p</td>
<td>p</td>
<td>p</td>
<td>p</td>
</tr>
<tr>
<td>14</td>
<td>p</td>
<td>p</td>
<td>p</td>
<td>p</td>
</tr>
<tr>
<td>15</td>
<td>p</td>
<td>p</td>
<td>p</td>
<td>p</td>
</tr>
<tr>
<td>16</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
</tr>
<tr>
<td>17</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
</tr>
<tr>
<td>18</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
</tr>
<tr>
<td>19</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
</tr>
<tr>
<td>20</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
</tr>
<tr>
<td>21</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
</tr>
<tr>
<td>22</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
</tr>
<tr>
<td>23</td>
<td>p</td>
<td>p</td>
<td>p</td>
<td>p</td>
</tr>
<tr>
<td>24</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
</tr>
<tr>
<td>Totals</td>
<td>11</td>
<td>17</td>
<td>19</td>
<td>18</td>
</tr>
</tbody>
</table>

The CONVEX compiler vectorizes or partially vectorizes 19 loops in a test in which optimal handcoding is supposedly able to vectorize or partially vectorize 18 loops. CONVEX uses directives with this benchmark, but only to keep performance results from being unrealistically high due to the compiler's effective elimination of "benchmark" loops, that is, loops introduced to extend the running time of the benchmark.

**Partial Vectorization**

Many loops contain recurrences, but the recurrences often constitute only a small percentage of the loop body. Partial vectorization breaks a loop into two or more loops to separate the recurrences from the vectorizable code, then vectorizes those loops not containing recurrences. The 5.0 compiler will successfully isolate vector code even if that requires breaking up statements executed conditionally. For example,

\[
\begin{align*}
\text{DO } 10 & \text{ I=1,100} \\
\text{IF(A(I).LT.B(I))GO TO 10} \\
T & = X(I) \\
\text{IF(A(I).LT.B(I))GO TO 20} \\
Z(I) & = X(I)^2 \times \text{SQRT}(B(I)) \\
\text{GOTO 10} \\
20 & \\
Z(I) & = X(I)^2 \times \text{TEMP(I)} \\
10 & \text{ CONTINUE}
\end{align*}
\]

is, effectively, rewritten as

\[
\begin{align*}
\text{DO } 10 & \text{ I=1,100} \\
\text{IF(A(I).LT.B(I))GO TO 10} \\
\text{TEMP(I)=SQRT(B(I))} \\
\text{GOTO 10} \\
10 & \text{ CONTINUE}
\end{align*}
\]

allowing the compiler to vectorize the square root computation.

**Parallelization**

Parallelization in this context refers to the generation of code that enables iterations of loops to be run simultaneously and without synchronization on multiple processors. A loop is parallelizable if there are no dependencies between iterations, that is, if the results computed by one iteration do not depend on the results of earlier iterations. Such loops have been referred to as \textit{DOALL} loops in the literature ([12], [3], [4]).

Parallelization is most frequently applied to the outer loop of a nest of loops or to the stripmine loop generated by vectorization. For example,

\[
\begin{align*}
\text{DO } 10 & \text{ J=1,M} \\
\text{DO } 10 & \text{ I=1,N} \\
A(I,J) & = B(I,J) + C(J,I) \\
\text{10 CONTINUE}
\end{align*}
\]

here the I loop is vectorized, but the J loop is parallelized. This allows up to N processors to be simultaneously engaged in executing vector operations. The compiler may further perform parallel
strip-mining on the outer loop to ensure that the synchronization overhead required for scheduling iterations is a minimum. Parallel strip-mining breaks the parallel loop into two loops, then parallelizes only the outer loop. Each processor then executes a contiguous strip of iterations of the original parallel loop.

When there is no loop outside the vector loop, the strip-mine loop is usually parallelized. For example,

\[
\text{DO 10 I=1,N} \\
10 \quad A(I) = B(I) + C(I)
\]

becomes

\[
\text{DO 10 I=1,N,128} \\
\text{DO 10 II=I,MIN(N,I+128)} \\
10 \quad A(II) = B(II) + C(II)
\]

a nest of two loops which can now be vectorized and parallelized as before. If N is a relatively small constant, but larger than 64, the compiler may generate a strip-mine length smaller than 128 to ensure there are enough iterations in the outer loop to make it worth parallelizing. If N is a variable, the compiler performs dynamic vector strip-mining to select the best strip-mine length at runtime.

The compiler can also handle most scalar assignments and reductions within parallel loops. For example, the following loop will be both vectorized and parallelized:

\[
\text{DO 10 I=1,N} \\
\text{IF(A(I).GT.0)GOTO 10} \\
\quad S = S + B(I)*C(I) \\
\quad X = B(I) \\
10 \quad \text{CONTINUE}
\]

The scalars are spread across the processors, partial sums accumulated for reductions, and iteration numbers recorded for other scalar assignments. After loop exit, the partial sums for S are added, and the thread-private copy of X of highest iteration number is copied into X.

The compiler can also extract concurrency from loops that contain recurrences or that have dependencies between iterations but not recurrences, so-called DOACROSS and FORALL loops [2], respectively. Parallel code is generated just as for parallelizable loops, but is supplemented with code to synchronize each iteration with the previous iteration where necessary to satisfy dependencies. However, loops where this technique will provide significant speedups are relatively rare. Also, many loops with recurrences are partially vectorized by the compiler. The residual serial loops from partial vectorization are fully recurrent and usually cannot benefit from applying multiple processors.

Loop Distribution

Loop distribution creates a sequence of simple loop nests from a loop nest. For example,

\[
\text{DO 10 I=1,N} \\
10 \quad A(I) = 0 \\
\text{DO 10 J=1,N} \\
\quad B(I,J) = A(I) \\
10 \quad \text{CONTINUE}
\]

becomes

\[
\text{DO 10 I=1,N} \\
10 \quad A(I) = 0 \\
\text{DO 20 J=1,N} \\
\quad B(I,J) = A(I) \\
20 \quad \text{CONTINUE}
\]

Loop distribution is performed to create more innermost loops suitable for vectorization, to create more potentially parallelizable loops by isolating recurrences of outer loops, and to create simple nests where loop interchange can be used effectively. Partial vectorization is really an application of loop distribution to an innermost loop followed by vectorization of the vectorizable resulting loops.

Loop Interchange

Loop interchange reorders the loops of a simple nest. The compiler may perform loop interchange for any of several reasons. It can be performed to decrease the stride with which vectors are accessed in a loop or, in some cases, to convert a vector into a scalar. (Converting vectors into scalars is what causes benchmark loops to seemingly "disappear" in the Livermore loops.) Loop interchange can also be used to increase the amount of code executed between synchronization points by moving the loop to be parallelized outward. Global vector register allocation is made possible by interchanging loops, relative to which vectors are loop constant, inside the strip-mine loop for vectorization. Finally, loop interchange may allow a loop that can be highly vectorized to be shifted inside an innermost loop that cannot be highly vectorized. A good summary of the
theoretical basis for loop interchange and its applications is given in [5].

An Example

An example of the compiler’s ability to vectorize, parallelize, interchange loops, distribute loops, and allocate vector registers globally to attain maximum performance for a matrix multiplication is given below. The choice of matrix multiplication is made, because it is simple, familiar, and an illustration how the compiler orchestrates many transformations to produce near optimal results. The algorithms, however, are general; the compiler makes no attempt to recognize that a matrix multiplication is being performed.

A common way to perform the matrix multiplication \( C = A \times B \) for \( N \) by \( N \) arrays \( A, B, \) and \( C \) is as follows:

\[
\text{DO } 10 \quad I = 1, N \\
\text{DO } 10 \quad J = 1, N \\
C(I, J) = 0 \\
\text{DO } 10 \quad K = 1, N \\
10 \quad C(I, J) = C(I, J) + A(I, K) \times B(K, J)
\]

The 5.0 compiler processes this loop nest by first distributing the \( I \) and \( J \) loops to yield:

\[
\text{DO } 10 \quad I = 1, N \\
\text{DO } 10 \quad J = 1, N \\
C(I, J) = 0 \\
\text{DO } 11 \quad I = 1, N \\
\text{DO } 11 \quad J = 1, N \\
\text{DO } 11 \quad K = 1, N \\
11 \quad C(I, J) = C(I, J) + A(I, K) \times B(K, J)
\]

It then interchanges the \( I \) loop to the innermost position in each nest so that vectors will be processed along the axis contiguously layed out in storage.

\[
\text{DO } 10 \quad J = 1, N \\
\text{DO } 10 \quad I = 1, N \\
C(I, J) = 0 \\
\text{DO } 11 \quad J = 1, N \\
\text{DO } 11 \quad K = 1, N \\
\text{DO } 11 \quad I = 1, N \\
11 \quad C(I, J) = C(I, J) + A(I, K) \times B(K, J)
\]

Both loops are now strip-mined, and in the second nest the outer loop resulting from strip-mining is moved outside the \( K \) loop:

\[
\text{DO } 10 \quad J = 1, N \\
\text{DO } 10 \quad I = 1, N \\
\text{DO } 10 \quad IOUTER = 1, N, 128 \\
10 \quad C(I, J) = 0 \\
\text{DO } 11 \quad J = 1, N \\
\text{DO } 11 \quad IOUTER = 1, N, 128 \\
\text{DO } 11 \quad K = 1, N \\
\text{DO } 11 \quad I = 1, N \\
11 \quad C(I, J) = C(I, J) + A(I, K) \times B(K, J)
\]

Vector code is now generated for both nests and parallel code is generated for the outermost loops. \( V0 \) and \( V1 \) here designate vector registers, 128 elements long.

\[
\text{DOALL } 10 \quad J = 1, N \\
\text{DO } 10 \quad IOUTER = 1, N, 128 \\
10 \quad C(IOUTER:MIN(N, IOUTER+127), J) = 0 \\
\text{DOALL } 11 \quad J = 1, N \\
\text{DO } 11 \quad IOUTER = 1, N, 128 \\
\text{DO } 11 \quad K = 1, N \\
V0 = C(IOUTER:MIN(N, IOUTER+127), J) \\
V1 = A(IOUTER:MIN(N, IOUTER+127), K) \\
V0 = V0 + V1 \times B(K, J) \\
11 \quad C(IOUTER:MIN(N, IOUTER+127), J) = V0
\]

Finally, global vector register allocation removes a vector load and a vector store from the \( K \) loop. There are only three vector references in the loop and the reference that remains in the loop body chains with the vector addition and the vector multiplication. The result is roughly a further 2.5x speedup.

\[
\text{DOALL } 10 \quad J = 1, N \\
\text{DO } 10 \quad IOUTER = 1, N, 128 \\
10 \quad C(IOUTER:MIN(N, IOUTER+127), J) = 0 \\
\text{DOALL } 11 \quad J = 1, N \\
\text{DO } 11 \quad IOUTER = 1, N, 128 \\
\text{DO } 12 \quad K = 1, N \\
V0 = C(IOUTER:MIN(N, IOUTER+127), J) \\
V1 = A(IOUTER:MIN(N, IOUTER+127), K) \\
V0 = V0 + V1 \times B(K, J) \\
12 \quad V0 = V0 + V1 \times B(K, J) \\
11 \quad C(IOUTER:MIN(N, IOUTER+127), J) = V0
\]

The following table summarizes the effect of these transformations on for \( N = 200 \):

<table>
<thead>
<tr>
<th>Extent of Optimization</th>
<th>Time (in ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Cumulative)</td>
<td></td>
</tr>
<tr>
<td>global optimization</td>
<td>7.82</td>
</tr>
<tr>
<td>+vectorization</td>
<td>0.84</td>
</tr>
<tr>
<td>+vector register allocation</td>
<td>0.35</td>
</tr>
<tr>
<td>+parallelization (4 heads)</td>
<td>0.09 (projected)</td>
</tr>
</tbody>
</table>
important because it simplifies the computation of mask expressions for masked vector code and often converts masked vector operations into unconditional vector operations. Arborification is fast, easily implemented, easily understood, and does a perfect job on well-structured code, although it will occasionally miss some mask simplifications that a slower, but more thorough algorithm, e.g., the Quine-McCluskey algorithm as used in [7], could identify.

OPTIMIZATIONS

The compiler performs a large number of optimizing transformations. Optimizing transformations, distinguishing them from reordering transformations, reduce the number of operations executed by a program or replace expensive operations with relatively cheap operations. All optimizing transformations except inline expansion of subprogram calls are performed by the back end. Optimizations are performed at the local and global levels on both scalar and vector code.

Local Scalar Optimizations

Local scalar optimizations include:

1. redundant assignment elimination (ræ),
2. redundant use elimination (rue),
3. assignment substitution (as),
4. common expression elimination,
5. simple strength reduction,
6. algebraic simplification,
7. branch simplification, and
8. constant folding.

Ræ, rue, and as are the fundamental transformations that create a DAG from a list of trees to represent a basic block. Ræ eliminates an assignment node, and other code thereby rendered redundant, if that assignment is followed in the same basic block by another assignment to the same variable. Rue combines two different use nodes for the same variable when there is no intervening assignment to the variable. The tails of dataflow arcs leaving the second such use node are moved to leave the first use node and the second use node is eliminated. As performs a forward assignment substitution from the source of an assignment to a subsequent use of the same variable. The assignment node is eliminated, the tails of arcs leaving the use node are moved to leave the node at the root of the source expression, and the use node is also eliminated. Assuming code can be generated for the basic block without running out of registers, ræ eliminates a redundant store, rue eliminates a redundant load, and as eliminates a redundant store/load pair.

Common expression elimination identifies two nodes representing the same operation but with identical inputs and deletes one in favor of the other after moving arc tails from the node to be deleted to the node to remain. The compiler also identifies common subexpressions, for example, detecting commonality between expressions like A+B+C and D+B+A. Each of these internally would be represented as a single ADD node with three operands, two of them shared. The common subexpression A+B is identified in both.

Simple strength reduction replaces expensive operations by sequences of cumulatively cheaper operations. For example, 5*I is replaced by SHIFT(I,2)+I.

Algebraic simplification applies a variety of algebraic identities to simplify expressions, for example, replacing I+0 with I or 3+5*I+J-4*(I+2) with I+J-5.

Branch simplification replaces a branch with a branch having fewer successors. For example, "IF(A.GT.0)10,10,10" is replaced by "GOTO 10". Control flow graph optimizations later remove the unreachable blocks and merge blocks with their predecessors when they follow unconditionally.

Constant folding replaces an operator with constant operands with a constant or replaces multiple constants feeding a commutative operator with a single constant.

Global Scalar Optimizations

Global scalar optimizations include

1. constant propagation,
2. copy propagation,
3. unreachable code elimination,
4. useless code elimination,
5. induction variable optimizations,
6. common expression elimination, and
7. partial redundancy elimination.

Constant propagation replaces a use of a scalar variable with a constant when all definitions reaching the use are assignments to the variable with equal, constant sources. Copy propagation is similar to constant propagation but replaces a use of a variable by the use of another variable of equal value. Copy propagation is only performed when replacement of the use renders an existing assignment useless. Such assignments are subsequently deleted by useless code elimination.
INTERNAL REPRESENTATION

This section describes the internal representations used by the compiler's back end. Front end input to the back end consists of a list of trees, roughly one per source statement. The back end immediately converts this to a normal graph representation on which local and global optimizations are performed. A distinct representation called the acyclic graph is used as the basis for vectorization and parallelization. These representations are closely related and in each the program is represented at the instruction level with the instructions represented as nodes and flows of data as arcs in a dataflow manner.

The Normal Graph

In the normal graph, basic blocks are represented as DAGs in which each node corresponds to an instruction. Dataflow arcs connect operands to operators. Use-definition (UD) arcs appropriately constrain the order in which memory reference instructions appear. Fig. 2 sketches a basic block for the two statements $A = C + B$ followed by $C = B - D$. Dataflow arcs are solid. UD arcs are dashed.

Basic blocks are embedded in a control flow graph which shows possible flows of control among basic blocks and identifies loop structures. Multiple entry loops are replaced with single entry loops for uniformity.

\[\begin{align*}
A &= C + B \\
C &= B - D
\end{align*}\]

Fig. 2: Basic Block

Finally, an iterative global data flow analysis algorithm [6] supplements the normal graph with arcs representing reaching definitions, that is, arcs that connect assignments to scalars with uses of scalars that they could reach. (The flow analysis algorithm is vectorized and strip-mined to process bit vectors 8192 bits long.)

The Acyclic Graph

Before each outermost loop is processed by the vectorizer/parallelizer, it is converted to the acyclic graph representation. The acyclic graph is formed from the normal graph by removing induction values, loop exits, loop end tests, and loop latch arcs from loop bodies and then arborifying the loop body. Finally, dependency analysis adds tagged dependency arcs to summarize the dependencies implied by the original program order.

An induction variable is a scalar variable that is incremented by the same loop constant amount on each iteration through a loop body. An induction value is any linear combination of induction variables. Loop constants and induction variables are first identified. Then, loop constants and induction values are removed from the loop and placed in the loop preheader. Induction values are represented in the loop preheader by nodes with base and stride inputs. As induction value nodes are created, they inherit the dataflow successors of the operators removed from the loop body, though, now, there are dataflow arcs crossing loop boundaries.

Loop end tests are removed after replacing the terminal branch of the preheader with a REPEAT node. The REPEAT node takes a single input, the loop trip count. The loop latch arc is also redundant now and is removed. An empty postlatch basic block is created to mark the end of the loop.

Exits are removed by moving their tails to issue from the postlatch block. Exit numbers are recorded in branch tables in the postlatch block and in the exit blocks of the loops. If a loop with an exit is vectorized (not implemented yet) or parallelized, then the appropriate exit number is recorded, and the loop is exited to the postlatch block where a branch on exit number is taken to the "original" exit. Loops that have exits and are not vectorized or parallelized are later simply restored to normal serial form.

Arborification (so called because it makes the graph more tree-like) identifies single-entry, single-exit regions of the control flow graph for a loop body, detaches the exit block and gives it the same predecessors as the entry block. This provides a graphic representation of the fact that there really are no control dependencies that constrain operations in the exit to follow those in the entry [14] (although there may be data dependencies that constrain them). Arborification is
Many of the local and global optimizations are performed using incremental algorithms that perform one of the optimizations and defer work for other optimizations. For example, global constant propagation may lead to local constant folding opportunities. Such opportunities are recorded on a work-list that is subsequently processed, perhaps leading to constant folding and more opportunities for global constant propagation, constant folding, algebraic simplification and other optimizations.

Unreachable code is code that cannot be reached by any path of execution from subprogram entry. Useless code may be reachable, but performs no computations that can affect final results. The compiler deletes both unreachable and useless code, warning the user of its existence in the original source.

As explained previously, induction values are removed from loops as a consequence of conversion to the acyclic graph representation. Since computations of induction values are expressed much as any other computations and reside in loop preheaders, they can be optimized there by all the ordinary local optimizations. Induction values are restored to loops when the acyclic graph representation is converted back to the normal graph representation. When restored, care is taken to minimize the computations required to produce each induction value sequence, taking into account simple relationships between induction values. For example, if $3^i$ and $3^i+1$ in the original program represent two induction values differing by 1, only one induction variable is introduced in the final code to represent $3^i$; $3^i+1$ is computed from it by adding 1.

Induction values are removed from loops twice, once before subscripts are expanded, simplifying dependency analysis and reordering transformations, and once after subscript expansion, removing expressions thereby introduced from loops. Code motion, strength reduction, and some global common expression elimination are implicitly performed by induction variable optimization. For example, code motion is the removal of induction values of stride 0 from a loop and global common expression elimination occurs when the common expressions, originally present in two different basic blocks, are both hoisted into a loop preheader, where local optimizations eliminate one in favor of the other.

The elimination of induction values from loop bodies is the reverse of a procedure used by source-to-source vectorizers to "normalize" induction values. Normalization expands the representations of induction values within a loop so that each appears as an expression linear in the trip counts of enclosing loops. This introduces inefficiencies into the body of the loop that are later eliminated by code motion and strength reduction. In contrast, the removal of induction values uses code motion and strength reduction to produce normalized induction values in the loop preheader, which can then be optimized by standard local optimizations.

To supplement the global common expression elimination implicit in induction variable elimination, a global partial redundant expression algorithm originally proposed in [8] was simplified, generalized, and adapted for use with our internal representation.

Local Vector Optimizations

Local vector optimizations include

1. vector meshing, and
2. condition mask simplification.

Vectorization of a loop reduces the loop to a single basic block. Vector meshing is a generalization of the scalar optimizations rae, rue, and as. Vectorization of loops with internal branching often produces many memory references to the same vector in memory. For example, there may be many references to array element $a(i)$ in a loop body, where each of these becomes a vector when the loop is vectorized. The effect of vector meshing is to reduce the number of memory references to a vector to at most a single load and a single store. Consider a loop in which there are several stores into a vector $V$ and all are vectorized, yielding:

\[
\text{stum}(V, \text{mask1}, \text{source1}) \\
\text{stum}(V, \text{mask2}, \text{source2}) \\
\text{stum}(V, \text{maskn}, \text{source}n)
\]

where \text{stum} represents a store-under-mask, that is, elements of the source are stored into the elements of the vector $V$ under control of a bit mask of the same length as $V$. These stores can be combined into a single store:

\[
\text{stum}(V, \text{ormask}, v(n))
\]

where

\[
\text{ormask} = \text{mask1} \text{or mask2} \text{or ... or maskn}
\]
Many of the local and global optimizations are performed using incremental algorithms that perform one of the optimizations and defer work for other optimizations. For example, global constant propagation may lead to local constant folding opportunities. Such opportunities are recorded on a work-list that is subsequently processed, perhaps leading to constant folding and more opportunities for global constant propagation, constant folding, algebraic simplification and other optimizations.

Unreachable code is code that cannot be reached by any path of execution from subprogram entry. Useless code may be reachable, but performs no computations that can affect final results. The compiler deletes both unreachable and useless code, warning the user of its existence in the original source.

As explained previously, induction values are removed from loops as a consequence of conversion to the acyclic graph representation. Since computations of induction values are expressed much as any other computations and reside in loop preheaders, they can be optimized there by all the ordinary local optimizations. Induction values are restored to loops when the acyclic graph representation is converted back to the normal graph representation. When restored, care is taken to minimize the computations required to produce each induction value sequence, taking into account simple relationships between induction values. For example, if 3*I and 3*I+1 in the original program represent two induction values differing by 1, only one induction variable is introduced in the final code to represent 3*I; 3*I+1 is computed from it by adding 1.

Induction values are removed from loops twice, once before subscripts are expanded, simplifying dependency analysis and reordering transformations, and once after subscript expansion, removing expressions thereby introduced from loops. Code motion, strength reduction, and some global common expression elimination are implicitly performed by induction value optimization. For example, code motion is the removal of induction values of stride 0 from a loop and global common expression elimination occurs when the common expressions, originally present in two different basic blocks, are both hoisted into a loop preheader, where local optimizations eliminate one in favor of the other.

The elimination of induction values from loop bodies is the reverse of a procedure used by source-to-source vectorizers to "normalize" induction values. Normalization expands the representations of induction values within a loop so that each appears as an expression linear in the trip counts of enclosing loops. This introduces inefficiencies into the body of the loop that are later eliminated by code motion and strength reduction. In contrast, the removal of induction values uses code motion and strength reduction to produced normalized induction values in the loop preheader, which can then be optimized by standard local optimizations.

To supplement the global common expression elimination implicit in induction variable elimination, a global partial redundant expression algorithm originally proposed in [8] was simplified, generalized, and adapted for use with our internal representation.

**Local Vector Optimizations**

Local vector optimizations include

1. vector meshing, and
2. condition mask simplification.

Vectorization of a loop reduces the loop to a single basic block. Vector meshing is a generalization of the scalar optimizations \( \text{rcr}, \text{rue}, \text{and \ as} \). Vectorization of loops with internal branching often produces many memory references to the same vector in memory. For example, there may be many references to array element \( a(i) \) in a loop body, where each of these becomes a vector when the loop is vectorized. The effect of vector meshing is to reduce the number of memory references to a vector to at most a single load and a single store. Consider a loop in which there are several stores into a vector \( V \) and all are vectorized, yielding:

\[
\text{stum}(V, \text{mask1}, \text{source1}) \\
\text{stum}(V, \text{mask2}, \text{source2}) \\
\text{...} \\
\text{stum}(V, \text{maskn}, \text{sourceen})
\]

where \( \text{stum} \) represents a store-under-mask, that is, elements of the source are stored into the elements of the vector \( V \) under control of a bit mask of the same length as \( V \). These stores can be combined into a single store:

\[
\text{stum}(V, \text{ormask}, \text{v(n)})
\]

where

\( \text{ormask} = \text{mask1 or mask2 or ... or maskn} \)
and \( v(j) = \text{mesh}(\text{mask}_j, \text{source}_j, v(j-1)) \),
and \( \text{mesh}(\text{mask}, v_1, v_2) \) is a vector of the same
length as \( v_1 \) and \( v_2 \) with elements of \( v_1 \) where the
mask is 1 and elements of \( v_2 \) where the mask is 0. 
\( n-1 \) masked vector stores have been eliminated at
the cost of introducing \( n-1 \) mesh operations and some or'ing of vector bit masks. On both the
CONVEX C1 and the CONVEX C2, this tradeoff
tends to be highly profitable due to the relative
inefficiency of store-under-mask on the C1 and the
chaining of the mesh operation with surrounding
loads and stores on the C2. (Most vector loops
are memory bound in the sense of [13]).

If uses appear embedded between assignments
in such a chain of stores-under-mask, straightforward assignment substitution leads to the elimina-
tion of vector loads. If an unconditional assign-
ment to the vector appears, all previous assign-
ments are eliminated.

A mask simplification optimization attempts
to simplify the mask expressions resulting from
vector meshing by iteratively applying two simple
boolean identities, \( p \text{ or } \neg p = \text{TRUE} \) and \( (p \text{ and } q) \text{ or } (p \text{ and } \neg q) = p \). Mask expressions frequently
simplify to \text{TRUE} leading to replacement of a
stum with a simple store. This typically occurs
when the same vector is assigned down comple-
mentary branches of an if-then-else construct.

**Inline Expansion**

Inline expansion replaces a call to a FOR-
TRAN subroutine or function directly with the
intermediate text of the subprogram. This elimi-
nates the usual call overhead. It also exposes the
code contained in the subprogram to the back end
for optimization in a context where it is fre-
quently possible to attain better optimization
results than could be attained in a separate com-
pilation. For example, an apparent recurrence in
a loop like:

\[
\text{DO 10 I=1,N} \\
10 \quad A(I) = A(I+J)*B(I)
\]
disappears if \( J \) is a formal argument with a
 corresponding dummy argument equal to 1. This
permits vectorization of the loop.

Inline expansion is performed on unoptimized
lists of trees in the FORTRAN front end and may
be recursively applied, that is, subprograms may
be inlined even if they contain subprogram calls
to other inlined subprograms.

**CODE GENERATION**

The use of an instruction level representation
by the optimizer greatly simplifies the job of the
code generator. Typically, nodes entering the
optimizer correspond to single machine-level
instructions except that no register assignments
have been made. Passes before actual instruction
generation replace macro-instructions with multi-
ple machine-level instructions or pseudo-
instructions and fold constants into displacement
fields for indexed addressing or into immediate
fields for appropriate operators. Pseudo-
instructions look like machine instructions to
most of the code generator, but are replaced after
instruction scheduling and generation with
sequences of machine instructions.

The code generator allocates registers glo-
bally. Since registers must be saved at call bound-
daries, the graph is segmented into call-free com-
ponents, which are optimized individually. A
map coloring algorithm similar to those described
in [9] and [10] was used and supplemented with
global target path determination, a generalization
of the local target-path determination described in
[11]. The address vs. scalar register problem
(integer computations can be carried on in either)
is handled by using an \text{either} class and assigning
whatever class is available and appropriate. The
result is a near optimal allocation of registers over
components.

Instruction generation and scheduling of
instructions within basic blocks are performed
together in a single pass after previous passes
have computed the information necessary for the
scheduling heuristics. Vector instructions are
scheduled to maximize vector chaining and scalar
instructions are scheduled to maximize scalar
pipelining and functional unit concurrency.
Potential conflicts arising from the banking of
vector registers are handled by four-coloring a
bank interference graph.

The most difficult problem in local code gen-
eration was finding a path through the dags of the
basic blocks that provided an appropriate pre-
ferred topological sort for the scheduler/generator. Several heuristics were used, including one suggested in [12]. Results are fre-
cently close to optimal, but on rare occasions a
poor order is chosen and too many spills are gen-
erated.

CONVEX languages use a caller-saves con-
vention for saving registers over subprogram calls.
A pleasant surprise was the interaction between
spills and the caller-saves convention. If the caller hasn’t changed the memory copy of some variable, it doesn’t need to write the register copy back; this saves half a spill (you still need a load, but no store). COMMON variables and arguments have to be written back by the caller in any case; this takes care of emptying some registers “for free”. Even some variables local to the caller, which could be left in a register across the call under a callee-saves convention, are going to get written to memory in any event to free up the register for something else, so these spills too are free. As long as the subprogram is likely to need more than a couple of registers, and on the C2 this can be relied upon, the caller ends up doing a much better job of optimizing spills.

The code generator resolves all commutative operators with three or more inputs into binary trees. The trees created are wide (bushy) or narrow (skinny) depending on the operator and the data type of the operands. Bushy trees allow instruction scheduling to arrange instructions to exploit the C2’s scalar pipelines. Skinny trees can be evaluated using fewer registers, at most two. For example, bushy trees are created for scalar floating point multiplication, because the multiplies can be pipelined by the C2’s scalar processor. Skinny trees are made for integer additions or vector multiplications, because no pipelining is possible in these cases.

Generation of code for parallelized loops is relatively straightforward. All induction variables in the loop are rewritten in terms of one “master” induction variable whose stride is the gcd of their strides. This variable is allocated to a communication register. A pointer to thread private stacks is loaded into the stack pointer, and a spawn instruction is executed to request idle processors to allocate themselves to the execution of the loop. Idle processors enter execution immediately following the spawn instruction and then enter the loop. At the top of the loop, the master induction variable is incremented, then local copies of the other induction variables are computed to drive the loop body for one iteration. Processors execute one iteration at a time, then branch to the top of the loop to take the next iteration. As processors exit the loop, they execute join instructions. The last processor to execute the join resumes the sequential thread and restores the original stack pointer.

CONCLUSIONS

CONVEX has created a family of compilers with a common back end capable of a wide variety of program transformations. The back end enables users to exploit the parallel and vector architectural features of the C2 supercomputer with minimal reprogramming effort and without requiring a source preprocessor or FORTRAN 8x notation.

ACKNOWLEDGEMENTS

Many people have contributed to the design and implementation of this compiler. Ken Kennedy provided us with invaluable advice and got us to read the right papers. I thank Bob Metzger, Terry Caracuzzo, Ron Lieberman, and Chris Smith for many helpful comments and suggestions.

REFERENCES


Effectiveness of a Machine-Level, Global Optimizer

Mark Scott Johnson
Terrence C. Miller

Hewlett-Packard Laboratories
P.O. Box 10490, Bldg. 3U
Palo Alto, CA 94303-0971

We present an overview of the design of a machine-code-level, global (intraprocedural) optimizer that supports several front-ends producing code for the Hewlett-Packard Precision Architecture family of machines. The basic optimization strategy is described, including information about the division of responsibilities between various components of the compiler. Optimization algorithms are described, including a discussion of the dataflow information they require. Measurements showing the collective and individual effects of various optimizer components are presented.

The performance data presented here was collected using a preliminary version of the optimizer. Development is continuing and further improvements are expected.

1 The Setting

The HP Precision Architecture defines a machine with a simple register-based instruction set using 32 general-purpose registers. Only load and store instructions reference memory. The architecture acknowledges the use of pipelined hardware by defining branches to not take effect until the second instruction cycle following the branch. Referencing the target of a load instruction in the instruction which follows the load will result in a one-instruction hardware interlock.

1.1 Memory Aliasing

The front-end of each compiler is responsible for gathering information about the set of memory locations actually or potentially touched by each variable reference or pointer dereference. This information is passed to the code generators by tagging each such (de)reference with an ID to specify the singleton (a single register or memory location) or aggregate (a set of memory locations) resource touched. The analysis handles overlap and containment as well as aliasing caused by use of pointers.

This mechanism is also used to specify the effect of procedure call, entry, and exit. Currently interprocedural analysis is not done and conservative assumptions are made (for example, in C the exit point of a procedure is marked as a potential use of all global variables), but the ID mechanism is capable of representing more accurate information.

1.2 Code Generation

The optimizer is currently connected to multiple code generators and is used with several languages, including C, Pascal, FORTRAN, and COBOL. Each code generator produces an in-memory data structure containing nodes representing machine instructions. This structure serves as the basis for all optimization processing.

The nodes contain fields of auxiliary information, such as markers of procedure boundaries and procedure calls. They also describe the effect of the instruction (for example, identifying which registers are sources and which are targets). Thus the optimizer need not repeatedly derive such information. Branch nodes point to the node of the branch target. This simplifies flow analysis and allows branch address calculation to be postponed until after optimization.
All nodes representing load or store instructions contain the ID of the memory resource(s) touched. The code is generated assuming an infinite register set. The same register is used for formally identical values (for example, the target of loads from the same memory resource). The registers are specified using singleton resource IDs, which allows dataflow analysis to be done in a uniform manner for all resources (memory and registers).

The code generators do not attempt to deal with pipeline interlocks and place NOP instructions after every branch instruction. The code generated is also incomplete since the instructions to save and restore registers across procedure calls and at procedure entry and exit cannot be generated until real register use has been determined.

2 Optimization Strategy

The optimizer was built to work at the level of machine instructions so as to permit the careful register allocation desired for the target architecture. The code sequences for address generation and loop control also provide significant opportunities for optimization. In this respect our design follows that of the PL.8 compiler.1 We feel we have improved on their work in the areas of aliasing and the performance of our register allocator (as compared to 4). Other comparisons are difficult to draw due to the lack of published details. More information on our design may be found in 7.

Placing the optimizer after code generation also makes it language independent. In fact separate teams, working in parallel, have constructed front-ends that use the optimizer.

The optimizer is structured into many separate components, each of which analyzes and/or transforms the program. Since different users have different priorities with regard to compile time, debuggability, and execution time/space, use of certain components is optional. In the data to follow, three levels of optimization are discussed. For each level, the components used are listed. A more detailed discussion of some components follows.

2.1 Level 0 Optimization

Level 0 is the minimal level of processing needed to complete the and to eliminate NOP instructions. The steps are:

1. Basic-block detection and dead-code elimination.

2. Conservative register allocation (includes generation of register save/restore sequences).

3. NOP elimination.

The NOP eliminator removes the NOP instructions the code generator inserted following each branch instruction. It operates by either interchanging the branch and the immediately preceding instruction or by changing the branch instruction to a form that nullifies execution of the following instruction.

2.2 Level 1 Optimization

Level 1 represents the set of optimizations that can be performed without global dataflow analysis. The steps are:

1. Basic-block detection and dead-code elimination.

2. Conservative register allocation.

3. Peephole optimization.

4. Branch optimization.

5. Instruction scheduling.

The branch optimizer does branch chaining and other simple optimizations of the interactions between conditional and unconditional branches. It also attempts to eliminate single-instruction basic blocks by replacing branches with instructions that conditionally nullify the execution of the following instruction.

2.3 Level 2 Optimization

Level 2 optimization uses the full set of available components. Global dataflow information is calculated. The steps are:

1. Basic-block detection and dead-code elimination.

2. Interval analysis and calculation of reaching definitions (including the local dataflow analysis required).


4. Calculation of upward exposed uses (including the local dataflow analysis required).

5. Calculation of use-definition webs for memory references and elimination of unnecessary memory references.
7. Calculation of use-definition webs for registers and the elimination of unused register definitions.
8. Coloring register allocation with copy elimination.
11. Instruction scheduling.

This list reflects the full set of components. Nothing was implemented and then discarded due to ineffectiveness. Further transformations such as code hoisting and global are being considered for later versions.

The transformation steps were ordered so as to reduce the amount of global dataflow information that needed to be recalculated after each transformation. The sections below which describe optimizer components include information about use of dataflow information.

Preliminary measurements showed the initial calculation of local and then global dataflow information taking more than 30% of the total running time of the optimizer. Having to redo that analysis after steps 3, 5, and 6 would have significantly impacted performance. Instead, the optimization components using global dataflow information also update the information as they perform each transformation. Local dataflow analysis is done only once. With the order chosen rebuilding dataflow information took less than 20% of the running time of the optimizer.

3 Global Optimizer Components

We present brief descriptions of those optimizer components that embody interesting differences from standard techniques.

3.1 Interval Analysis

The control structure of the procedure is recovered using the technique of interval analysis introduced by Sharir. Our implementation extends his work by adding an interval corresponding to the C switch statement and by changing the construction of improper intervals so as to reduce their size (thus reducing the number of iterations required to propagate dataflow information for those intervals).

Global dataflow information is generated based on the interval structure and it is also used to give dynamic weights (called loop costs) to each basic block. The cost of a block is raised (multiplied by 8) if it is included in a loop and lowered (halved) if it is conditionally executed.

3.2 Dataflow

Dataflow information is handled based on sequence numbers to identify a particular reference to a resource. There are also generic sequence numbers to represent references to a resource in an unspecified location. Because of aliasing, some resource IDs represent sets of memory locations. Depending on the type of aggregate, a reference to the resource can indicate that an individual member of the set might be referenced (for example, aliasing due to pointer ambiguity) or that each member of the set will be referenced (for example, equivalenced variables).

Local dataflow analysis must take into account the ambiguity introduced by aliasing. The following information is calculated for each basic block B.

1. M_GEN—definitions that might reach the end of basic block B.
2. W_D_KILL—definitions outside of B that define resources that are definitely defined within B.
3. M_USE—uses within B that have no definite preceding definition within B.
4. W_U_KILL—uses outside basic block B of resources that are definitely defined within B.

The information collected as the basic block is processed is sufficient to support several simple, local optimizations. These are:

1. replacing loads from memory with register-to-register copy operations when the value is available.
2. eliminating redundant definitions of a resource.
3. constant propagation.
4. simple peephole analysis.

With the exception of constant propagation, all the transformations would be done by other components, but would require dataflow update if done later. The early elimination of instructions also reduces the cost of later components.
performance data which follows this set of transformations is referred to as basic block optimizations.

An important consideration in our design was an attempt to minimize the amount of the global dataflow information required. All the transformations described here can be performed using only two kinds of global dataflow information:

1. REACH.T—identifies those sequence numbers representing what might be a redefinition of the resource that might reach the top of a given basic block along some path.

2. NEED.B—identifies those sequence numbers that might be uses of a resource that might be exposed along some path at the bottom of the given basic block.

The REACH.T set uses generic sequence numbers to track when a particular resource is undefined. Resources local to a procedure (such as most registers) are undefined on entry. A resource may become undefined when a value upon which it depends is redefined (for example, storing into a memory location causes the register into which the memory location is always loaded to become undefined).

### 3.3 Common-Subexpression Elimination

An instruction represents a common subexpression that may be eliminated if the undefined property (as defined by REACH.T - see above) for the target resource does not reach the instruction and all definitions that do reach it are the same as the instruction. By calculation of the reach of the undefined state of a resource we avoid having to compute available expressions.

Common-subexpression elimination is done first since it represents the only use of the undefined information (which proved difficult to update) and can be done prior to the calculation of NEED.B. The early elimination of instructions also simplifies later transformations.

### 3.4 Web Analysis

Web analysis logically builds (for each resource) a graph whose edges connect each instruction defining a resource with all those instructions that might reference it. Webs are defined as the maximal connected subregions of these graphs. A web which consists of a single definition represents an unused definition and that instruction is eliminated.

A web comprised solely of unambiguous references to a singleton memory resource is subject to store-copy transformation. The store instruction is replaced by a copy into the register that the code generator used as the target for all of the loads. The load instructions in the web can then be eliminated. This transformation greatly increases the use of registers. C register declarations are obeyed by the code generator, but are largely superfluous when level 2 optimization is applied.

The web calculation for memory and register resources is done separately since both store-copy transformation and the loop optimizations potentially invalidate much of the register information.

### 3.5 Loop Optimizations

Loop-invariant code motion may be applied to instructions that are conditionally executed within the loop. This is done if the instruction can be shown to execute correctly outside the conditional and if a moved definition will not then reach additional uses of the resource. This extension was motivated by the common programming construct:

```plaintext
if (...) 
    large block of code 
else 
    error();
```

and also permits optimization of while loops. Redundant code created when identical instructions are moved from different branches is eliminated.

Induction-variable elaboration is done using the procedure proposed in 5 and extended to apply constant propagation to induction variable initialization and to immediately eliminate code known to have been made redundant by a transformation. The elimination of multiplication operations is particularly important for an architecture that does not contain an integer multiply operation.

Because it requires use of new resources, induction-variable elaboration is the only occasion in which the flow propagation equations derived by interval analysis are reused. Even then, the changes are propagated only to the boundary of the loop interval, and the remaining information is updated.

### 3.6 Register Allocation
Two register allocators are available: a conservative allocator that cannot move information out of registers into memory (does not generate spill code) but can eliminate some copy instructions, and a graph-coloring allocator that performs register spilling as necessary.

Both generate the register save and restore code that is required at procedure entry and exit and around procedure calls since the exact set of registers used is not known until after register allocation. The allocators use a mixed register saving strategy dividing the available registers into two sets. The members of one set must be saved around a procedure call if live across the call (caller saves) and the others must be restored by a procedure that uses them (callee saves). Considerable memory traffic is eliminated by assigning values that are not live over a procedure call into a caller-saves register.

3.6.1 Conservative Register Allocator

The conservative register allocator does not require global flow analysis and does its own limited local analysis. It makes a forward pass over each basic block, building a definition-use web for each register resource. It tags any uses occurring before a definition as a register variable. If any such resources are tagged, the allocator then makes a backward pass looking for definitions on all paths in order to find the earliest and latest basic blocks for which the resource might be alive. Finally, the allocator makes a register-assignment pass, allocating register numbers from a pool of available numbers and eliminating redundant copy instructions.

Most programs do not require spill code when 32 real registers are available. Thus they can take advantage of this lower cost register allocator during development when the store-copy transformation is not being applied.

3.6.2 Coloring Register Allocator

The second register allocator was based on the work described in 4 and our original implementation closely followed that design. The algorithm was then changed in several respects.

Register allocation is applied to individual use-definition webs independently. Before allocation is done, the virtual registers are renumbered so that each web uses a different register.

Only one representation of the interference graph is used and it is updated during copy elimination. Thus the graph is generated only once, and in a single pass over the instructions.

Two costs are assigned to each register reference: the number of instructions required to store and reload the resource (its spill cost) and the impact that keeping a resource live will have on other resources (its live cost). Both are based on the loop costs assigned to each basic block during interval analysis. Spill cost is calculated as defined in 4 (including the special cases described there) and live cost is the sum of the loop costs for the basic blocks over which the register is live. Spill decisions are based on spillcost – livecost.

If the spill cost is zero (or negative) or if the live cost is high compared to its spill cost, the resource is automatically spilled before coloring.

Real registers are first assigned to virtual registers that are live in more than one basic block and within that group in order of decreasing adjacency count in the graph. The set of real registers available for assignment to each virtual registers is calculated based on the interference graph and previous assignments. When no real register is available, the virtual registers that have been assigned real registers and the one being processed are candidates for being spilled. The register with minimum spillcost – livecost is selected. Whatever virtual register is spilled is left without an assigned real register and allocation proceeds. While more expensive than use of the coloring heuristic, this algorithm was seen to produce better register assignments.

3.7 Instruction Scheduler

The instruction scheduler performs three types of scheduling:

1. replacement of the NOP instruction following each branch instruction by a useful instruction (branch scheduling).
2. rearrangement of instructions to avoid load/store interlocks (load/store scheduling)
3. rearrangement of instructions to avoid interlocks due to floating-point instructions (floating-point scheduling).

The algorithm used for load/store scheduling is similar to, but faster than, that described in 9 and a preliminary version is discussed in 8. That work was considerably extended for floating-point scheduling since floating-point computations execute in parallel with the normal instruction stream.

4 Optimizer Effectiveness