The Mahler Experience: Using an Intermediate Language as the Machine Description

David W. Wall
Michael L. Powell

Digital Equipment Corporation
Western Research Laboratory

Abstract

Division of a compiler into a front end and a back end that communicate via an intermediate language is a well-known technique. We go farther and use the intermediate language as the official description of a family of machines with simple instruction sets and addressing capabilities, hiding some of the inconvenient details of the real machine from the users and the front end compilers. Then we can implement each machine in this family with whatever technology is appropriate, without having to make the details match those of other machines in the family. Each machine can therefore be faster than it would be if we required the machines to be object-code compatible, but the front end compilers need not change to accommodate that flexibility.

To do this credibly, we have had to hide not only the existence of the details but also the performance consequences of hiding them. The back end that compiles and links the intermediate language tries to produce code that does not suffer a performance penalty because of the details that were hidden from the front end compiler. To accomplish this, we have used a number of link-time optimizations, including instruction scheduling and interprocedural register allocation, to hide the existence of such idiosyncracies as delayed branches and non-infinite register sets. For the most part we have been successful.

1. Introduction

Many compilers consist of two pieces: a front end that translates the source language into an intermediate language, and a back end that translates the intermediate language into object code. Using an intermediate language allows one to retarget a compiler by writing only a new back end, or to implement an additional language by writing only a new front end. Thus an intermediate language serves either as an abstract machine description or as an abstract language description, depending on the side of the abstraction with which one is working.

The problem with using an intermediate language as an abstract machine description is that it can hide important features of the machine and make it harder for a front end to generate good code. Thus it is important for the back end to do a good job of hiding the real machine: it must hide not only the details of the real machine, but also the performance consequences of those details. A common example of this arises when we use an intermediate language like P-code [12,13] which is based on an abstract stack machine; a good P-code translator must not implement the P-code stack operations as real stack operations, but rather must make good use of registers and addressing modes so that there is no performance penalty for the detour through P-code.

A shorter version of this paper appears in the Proceedings of the Second Symposium on Architectural Support for Programming Languages and Operating Systems,

If the back end hides the tricky details of the machine and then does a good job of compensating for the performance consequences, we can use the intermediate language for most purposes as the official description of the machine. This is particularly appropriate if the machine itself has the simplified instruction set and addressing power associated with ''RISC'' machines [6,7,9]. We can then think of the underlying instruction set as the microcode, and the intermediate language as the actual machine. As we produce different generations of machine with small variations, we can treat these as merely different implementations of the same official machine.

This paper describes our experience using an intermediate language as the ''official'' machine description, and assesses how well we have compensated for hiding some of the details of the machine behind this official description.

2. Titan and Mahler

The Titan is an experimental, high-performance, 32-bit scientific workstation developed at DEC Western Research Lab. It includes a central processor with 63 general-purpose registers, and a coprocessor with a single 64-bit accumulator. The coprocessor does floating-point operations and integer multiplication and division. The central processor has a four-stage pipeline with a peak instruction issue rate of one instruction per cycle.

Memory is accessed only through simple load and store instructions with a single base register and 16-bit displacement. Other operations like addition and subtraction are three-register instructions but cannot access memory.

Branch instructions in the Titan are *delayed branches.* A branch instruction in the Titan does not cause the pipeline to be stalled or flushed; instead, the next instruction in line is executed normally and then the branch is taken if required. For example, if *(w x y z)* is a sequence of instructions in memory, and if *w* is an unconditional branch to *z,* the actual execution sequence will be *(w x z).* If in addition *x* is a branch back to *w,* then it will be *(w x z w).* This behavior is common in microengine instruction sets, but is not normally seen in macrocode. This means that the next instruction must be one that may correctly be executed whether or not the branch is taken; otherwise, the branch must be followed by a null instruction. The instruction executed after a branch is said to occupy the *branch slot;* filling the branch slot with a useful instruction rather than a null instruction is desirable.

The coprocessor operates asynchronously with the central processor. The central processor can simultaneously fetch the previous result from the coprocessor and initiate a new coprocessor operation. Many of the coprocessor operations take more than one cycle to complete, and some take several tens of cycles. Thus the central processor may be doing other things while the coprocessor is working. If the processor tries to fetch a result from the coprocessor before the result is ready, the processor stalls for one or more cycles waiting for the coprocessor to finish.

The pipeline can also stall because of a memory reference. A one-cycle stall occurs if a load or store instruction immediately follows a store instruction, or if the base register of a load or store instruction was set by the preceding instruction.

Thus we can waste cycles for a variety of reasons: because of a memory reference stall or a coprocessor stall, or because we can't find anything better than a null instruction to put in a branch slot. The Titan is unlike some other RISC machines [6] in that memory reference stalls or coprocessor stalls are handled by the pipeline

and do not require explicit null instructions to be inserted. Nevertheless, a programmer working at the machine level would have to avoid these wasted cycles in order to use the machine well.

Mahler is the intermediate language used by all of the compilers implemented for the Titan. It has a uniform syntax for operations performed on addresses, signed integers, unsigned integers, single-precision reals and double-precision reals. These operations are performed on constants or named variables. A variable may be either local to a procedure or global, and may be either a user variable from the source program or a temporary variable generated by the front end compiler. Branches in Mahler are not delayed. There is no concept of a register in Mahler; it is as if there is an unbounded number of registers available, each corresponding to a named scalar variable. Procedures are explicitly declared and explicitly called, as in a high-level language.

Thus Mahler hides several things about the Titan: the limit of 63 registers, the multi-cycle operations in the coprocessor, the delayed branches, and the pipeline stalls near loads and stores. We wanted the implementation of Mahler to make it irrelevant that we had hidden these things, by doing a good job of simulating a machine that did not have these idiosyncracies. The next step is to describe the implementation so that we can assess it.

3. The Mahler implementation

The Mahler implementation consists of the Mahler translator and an extended linker. The translator reads Mahler source files and produces object modules. These object files can be linked in the usual manner to form an executable program, or they can be improved at link time by invoking an interprocedural register allocator and an instruction scheduler. The translator annotates the object modules extensively with information that enables these link-time improvements to be done quickly and correctly.

3.1. The Mahler translator

Translating a Mahler source file into object code for a RISC machine is not hard. Because the instruction set of a RISC machine is so simple, there is usually one preferred way to implement most operations. In particular, there is no need for tricky parser-based code-generation techniques [3], which are useful on non-RISC machines with powerful instructions and addressing modes only because one must in a sense ‘‘decompile’’ an intermediate form to combine the low-level operations into the complicated machine instructions.

We nevertheless apply many classical local optimizations that involve detecting cheap special cases of expensive operations. For example, it is well-known that integer multiplication by a constant power of two is equivalent to a left shift. This optimization is useful to us because integer multiplication is done by the coprocessor and requires several cycles. In fact, it requires eight cycles altogether, which means that we can profitably replace a constant multiplication even by the sum or difference of as many as four shifts. Since multiplications by the constants 1 through 170 can be expressed this way, this takes care of all the multiplications in many programs.*

The Mahler translator starts by parsing the Mahler source file and producing an internal form that represents a basic block as a directed acyclic graph (DAG). All of the local optimization is then performed on

*The replacement of a multiplication by a combination of shifts was recently explored more generally by Bernstein [1].

this DAG structure, so that it is relatively independent of the details of the particular generation of Titan the translator is targeted for. Within each basic block, the DAGs for common subexpressions, including multiple loads of the same variable, are merged when possible.

After the DAG processing, the translator traverses the DAG generating code for each node in turn. It allocates temporary registers from a small set reserved for expression evaluation, and maintains a description of what is in each register so they can be spilled to memory if necessary. It maintains the state of the coprocessor accumulator as well. This lets it keeps a value in the accumulator if it will be needed there later in the basic block, and lets it spill the value in order to compute another partial result if it must.

For the most part, the translator does not try to avoid pipeline stalls, and it generates a null instruction after every branch. The exceptions to this occur only inside ‘‘idioms,’’ our term for single operations that are requested by the front end but that require sequences of several instructions at the machine level. One example is procedure call. Another is variable left shift, which is done by doing an indexed jump into a table of thirty-two constant left shift instructions, as follows:

```
/* compute rx left-shifted by ry
   bits, where 0 <= ry <= 31 */
      r1 := pc + ry
      goto 3[r1]
      goto done
      r2 := rx leftshift 0
      r2 := rx leftshift 1
      r2 := rx leftshift 2
      r2 := rx leftshift 3
      . . .
      r2 := rx leftshift 31
done:
```

Because the branch slot of the first goto contains another goto, the effect is to execute one of the 32 constant shifts and then go to the point labelled ‘‘done’’ and continue. These idioms are somewhat like a set of standard macros, and as such we have carefully tailored the expansions to take advantage of the cycles after branches and of the time that the processor would otherwise wait for the coprocessor to finish its own operations.

The translator also makes no attempt to keep variables in registers, except locally within a basic block, nor does it try to fetch coprocessor results and initiate new coprocessor operations simultaneously, even though the coprocessor instructions allow this. In general the translator generates simple code that will run reasonably even if the user requests no link-time improvements.

However, the translator does do a lot of work that makes it easier for the interprocedural register allocator and the instruction scheduler to do their jobs when they are invoked at link time. For the benefit of the register allocator, it includes in the object module a table of the global variables and procedures in that module, and for each procedure it includes a table of the variables that it uses, the procedures that it calls, and its local variables. It also annotates the code heavily with a kind of generalized relocation information that the register allocator can use to remove loads and stores after it has decided which variables should reside in registers. Finally, it flags certain code sequences in idioms as unchangeable, which tells the instruction scheduler not to change the instructions so marked; this is important if the sequence depends on position, such as the computed jump in the

variable shift example above.

In addition, the translator optionally does a dataflow analysis on each procedure to determine whether two locals of the same procedure are ever simultaneously live. This means that the register allocator can assign them to the same register when it assigns variables in different non-conflicting procedures to the same register.

When the translator has finished with all of the source files, the resulting object modules can be linked either in the usual manner or with register allocation and instruction scheduling based on the information that the translator has included with each object module.

3.2. The interprocedural register allocator

The interprocedural register allocator was described more fully in a previous paper [11].

The register allocator is invoked by the linker. In the first pass it accumulates the summary information that the translator included with each module. From this it builds a complete call graph for the program and estimates the frequency with which each scalar variable or constant is used.

The aim of the register allocator is to select the most frequently used variables and assign them permanently to registers instead of to memory. The variables not assigned to registers will be accessed from memory via the small number of temporary registers controlled by the translator, just as if no link-time allocation was done. The variables eligible for assignment to registers are any scalar quantities which would otherwise require memory space: user variables, temporary variables created by the front end compiler or by the Mahler translator, addresses of procedures, records, or arrays, and numeric constants.

Two (or more) variables may be assigned to the same register if those two variables never simultaneously contain live values; the allocator guarantees this by using the same register only for variables that are local to different procedures that are never simultaneously active. The allocator can also use the same register for two locals of the same procedure if the translator has done dataflow analysis for that procedure to determine whether the two are simultaneously live. Thus the local variables are combined into groups that compete for registers as if they were very heavily used single variables.

The most frequently used variables or groups of nonconflicting variables are assigned to registers. The decision about which variables are frequently used can be based on estimates accumulated by the translator or on profile information produced in a previous execution of the program.

After the assignment of variables to registers, a second pass is performed that modifies each module to be linked, based upon the variables assigned and the annotations to the code that were made by the translator. Most of these modifications are simply the removal of a load or store, or the replacement of a temporary register by a register allocated to a variable. For example, the assignment ‘‘x = y + z’’ would lead to annotated code like this:

instruction	*actions*
r1 := load y	REMOVE.y
r2 := load z	REMOVE.z
r3 := r1 + r2	OP1.y OP2.z RESULT.x
store x := r3	REMOVE.x

If y is assigned to a register, the REMOVE.y action says to remove the first load, and the OP1.y action says to replace the first operand of the add with the register allocated to y. If x, y, and z are all assigned to registers, these four instructions are replaced by a single add instruction.

Recursive or variable procedure calls require extra work, as do global variables with preloaded initial values. The allocator also rewrites calling sequences in order to pass arguments directly to parameter registers. These topics are beyond the scope of this paper.

The modified object module is then returned to the linker for instruction scheduling.

3.3. The instruction scheduler

The aim of the instruction scheduler is to find the places where cycles will be wasted, and to move instructions around so that these wasted cycles are filled with productive instructions that would otherwise require cycles of their own. These wasted cycles can happen because of a pipeline stall in accessing memory, a stall waiting for the coprocessor to finish, or a null instruction after a branch.

The instruction scheduler reorders each basic block independently, and then makes a final pass looking for branch slots that have not been filled, trying to fill each of them with a useful or potentially useful instruction from the next block to be executed.

3.3.1. Scheduling a basic block

The scheduler starts by computing the earliest time at which each instruction can be executed relative to the beginning of its basic block. This earliest time is determined by considering all of the preceding instructions that conflict with the instruction. Two instructions conflict when they could not be reversed if they were adjacent. This can happen if they both set the same register, if one modifies a register used by the other, or if one of them stores to memory and the other loads or stores the same location. Usually it is impossible to tell if two memory references are to the same location, so we must assume the worst and suppose that they are. The one exception is when the two references use the same base register and different displacements. These two references do not conflict even though an intervening instruction might change the value of the base register, because if that intervening instruction is present, then it will conflict with both memory references, and the order of the three will be preserved.

In the same way, the scheduler computes the latest time at which each instruction can be executed, by considering the instructions that follow it for possible conflicts.

The instructions in the block (except for the final branch, if present, and its branch slot) then get sorted according to latest start time, breaking ties according to the earliest time. Thus if the first few instructions in a block are mutually independent set-up instructions that can be performed in any order, the earliest possible start time for any of these is time 0.

Another sorting pass looks for instructions that stall because of an interaction with the previous instruction. The scheduler delays such an instruction by exchanging it with one or more following instructions whose earliest allowed time is earlier than the earliest allowed time of this instruction. It is worth noting that no DAG representation of a basic block is used [5]; conflicts are encoded by the latest and earliest allowable times.

The scheduler makes another linear pass looking for coprocessor fetches that are immediately followed by coprocessor initiations. If the initiation simply reloads the accumulator with what was previously fetched, the scheduler eliminates the reload; if not, and if the operand registers of the new operation are different from the destination of the previous fetch, then it merges the two coprocessor instructions into one.

At this point, the scheduler has filled as many of the memory stalls and coprocessor stalls in the basic block as it can.* It then looks to see if the block ends with a branch whose slot is empty, and if the instruction that has been sorted to the position just before the branch can be performed later than the branch itself. If so, this instruction is moved into the branch slot to replace the null instruction there. This is enough to fill many branch slots; for instance, it is common for a loop to end with the operations:

```
more := i < limit
i := i + 1
if more goto TopOfLoop
```

The instruction that stores the incremented value in i can be done in the branch slot. (If interprocedural register allocation has assigned i to a register, the instruction in the branch slot can perform the entire increment.)

3.3.2. Filling slots with future instructions

When the scheduler reorganizes a basic block, it may fill the branch slot with an instruction from earlier in the basic block. When it has reorganized all of the blocks, however, many branch slots may still be unfilled. The possibility remains of looking in the block that will be executed *after* a branch, and moving an instruction from there into the branch slot. A final pass over the object code tries to do this.

It turns out to be convenient at the same time to coalesce branch chains. If we have a branch that jumps to an unconditional branch, we can sometimes change the first branch so it bypasses the second branch by jumping directly to the same place. The only problem is that the second branch's slot may contain an instruction that must be executed. If the first branch is unconditional, we can guarantee that this instruction gets executed anyway by copying it into the slot of the first branch. Note that we cannot coalesce a chain of branches if they both have non-empty branch slots.

If we have an unconditional branch that jumps to a nonbranch, we can copy the nonbranch into the branch slot and change the branch so that it jumps to the instruction after the nonbranch. This also works if the destination is a conditional branch, because of the way delayed branches work.

The most interesting case is when we have a conditional branch to a nonbranch. In this case we are both helped and hindered by the fact that the branch may or may not be taken. We can look in the destination block for an instruction to fill the branch slot, and we can also look in the fall-through block that comes next in the code, where we will be if the branch is not taken. In either case, however, we must make sure that the instruction we select is harmless if we actually go to the other block instead.

Suppose we are looking in the destination block; the other case is symmetric. A candidate instruction is one whose earliest time is 0 and which can therefore be executed first; since the block is sorted most of these are at the beginning. One of these instructions is harmless if the register it modifies is dead at the beginning of

*More might be filled by doing global dataflow analysis of the code and reorganizing it completely, including making different use of registers, but this seems impractical.

the fall-through block. Dataflow analysis would give us precise information about this, but we can do pretty well just by looking to see if the fall-through block sets that register before using it.

For example, consider the following code:

```
        if r4 = 0 goto L1
        null

        r1 := load x
        r2 := load y
        r1 := r1 + r2
        goto L2
        store z := r1

L1:     r3 := load u
        r1 := load v
        r3 := r3 - r1
        store z := r3

L2:
```

Suppose we want to fill the branch slot after the conditional branch, and we look in the destination block at L1 for a suitable instruction. Our first candidate is the load of u into r3, but there is no place where the fall-through block sets r3. As such, r3 might have a live value at that point, and it is therefore not safe to copy the load of r3 into the branch slot. Our next candidate in block L1 is the load of v into r1. Since the fall-through block *does* set r1 before using it, we know that it would be safe to copy this load into the branch slot. The subtraction would not be eligible at all, because its earliest allowable time is not 0. A similar analysis would tell us that the only possible instruction in the fall-through block is its load of x into r1.

If we select an instruction from the fall-through block, we must actually delete it from the block. Otherwise it will be executed twice: once in the branch slot and once in the block itself. We have no such problem with an instruction from the destination block, since we can move the instruction into the first position in the destination block, copy it into the branch slot, and then change the destination of the branch so that it skips over the original. However, we must then mark the original as immovable in the destination block, so that we don't move a different instruction into first place later on when we look for an instruction to fill some other branch slot.

In the above example we can fill the branch slot with a load into r1 from either the fall-through block or the destination block. The resulting code in each case would be:

	from fall-through	*from destination*
	if r4 = 0 goto L1	if r4 = 0 goto L1+1
	r1 := load x	r1 := load v
		r1 := load x
	r2 := load y	r2 := load y
	r1 := r1 + r2	r1 := r1 + r2
	goto L2	goto L2
	store z := r1	store z := r1
L1:	r3 := load u	r1 := load v
	r1 := load v	r3 := load u
	r3 := r3 - r1	r3 := r3 - r1
	store z := r3	store z := r3
L2:		

We consider using the fall-through block only if it cannot be branched to from somewhere else, since we must actually remove the instruction from that block. We also prefer to use the destination block if the branch is a backward branch, since such a branch is probably a loop and therefore is more likely to be taken than not taken.

It is worth noting that our ability to fill one conditional branch slot may be impaired by decisions we made in filling a previous branch slot. Trying to find the optimal filling of the branch slots is more work, and we will see later that we do quite well without it.

Any time we change a branch destination, it may open the possibility of another transformation on the same instruction (as with a branch to a branch to a branch), so for each branch we repeat this analysis until no change is possible.

3.4. Cooperation

We have already seen that the translator provides information that the register allocator uses to choose register variables and rewrite the code based on that choice. We have also seen that the translator marks tricky ‘‘idioms’’ so that the scheduler will not hurt them. There are other forms of cooperation as well.

The most important example is the synergy between the register allocation and the instruction scheduler. The scheduler tends to do much better at scheduling a block in which most of the variables referenced have been assigned to registers, because there are much fewer conflicts between instructions. If

```
r1 := x
r1 := r1 + 1
x := r1
```

appears in the middle of a block, other instructions involving r1 cannot be moved past it in either direction. If x resides in a register, however, these uses of r1 disappear and other instructions become more mobile. The main

reason why the instruction scheduler is in the linker rather than in the translator is that it can schedule the code after register allocation has been done.

The translator also helps the scheduler by assigning the dedicated temporary registers in a cycle, so that when we are finished using r1, we will not allocate it for a new purpose until several instructions later. Thus even a block without many register variables will tend to have more mobile instructions than it would if we looked for a free temporary register from the same position every time.

These considerations unfortunately hamper the effectiveness of the postpass that fills conditional branch slots from one of the two possible successor blocks, since it is harder to be sure that a particular register is initially dead. The translator alleviates this somewhat by starting the cycle over each time a new basic block begins. Thus the first temporary register to be assigned by a block is more likely to be r1 than any other, so it is more likely that the two blocks will both have such assignments. In that case either one could be moved into the branch slot.

4. Evaluation

How well does Mahler do at simulating a machine without the idiosyncrasies mentioned earlier? This question is sometimes difficult to answer, because it is often hard to know how well it would be *possible* to do. However, by instrumenting the Mahler implementation and the code it produces, we were able to produce some partial answers to the question.

We used six benchmarks to derive the measurements described in this section: the Livermore Loops, the Whetstones benchmark, the LINPACK linear equation benchmark [2], the Stanford suite collected by Hennessy [4], the logic simulator RSIM [10], and a timing verifier.

	lines	*vars*	*procs*
Livermore	268	166	1
Whetstones	462	254	13
Linpack	814	214	12
Stanford	1019	402	49
Simulator	3003	811	97
Verifier	4287	1395	112

Table 1. The six benchmarks.

4.1. Limited number of registers

If the Titan really had an unlimited number of registers, we could keep all scalar variables and constants in registers rather than in memory. (Memory references would still be needed, of course, to access array or pointer structures.) We instrumented the register allocator to tell us which variables were assigned to registers, and compared that to an execution profile that tells us how often each variable was referenced in memory when interprocedural register allocation was not done. This let us compute the fraction of these variable references

that were made from globally allocated registers rather than from memory.* We can also compare the list of register variables to a static count of the variables in the code of the program. This tells us how much we get of the possible improvement in code space rather than execution speed.

	static	*dynamic*	*dynamic w/ profile*
Livermore	66%	81%	94%
Whetstones	55%	75%	88%
Linpack	77%	96%	99%
Stanford	71%	92%	98%
Simulator	68%	83%	95%
Verifier	49%	61%	83%

Table 2. Percentage of variable memory references replaced by register references.

Table 2 gives the proportion of variable references that were changed from memory references to register references for each of the six benchmarks. The first column counts the references statically, each reference appearing in the object code for the program counted once. The second column counts the references dynamically over an execution of the program, each reference counted as many times as it is executed. Even a large program can make two-thirds of its executed variable references from registers rather than memory; smaller programs can make nearly all of their variable references from registers. The static proportions are not as good as the dynamic proportions, because the allocator selects register variables so as to reduce the latter; nevertheless the static proportions are also quite good.

The dynamic counts in the second column are for a register allocation based only on the usage frequency estimates made by the translator. The third column also counts the references dynamically, but the allocation was based instead on a previously gathered profile. Doing this can improve the proportion dramatically for large programs.

4.2. Multi-cycle operations in coprocessor

Some operations like multiplication and division simply take a long time to do. Our only hope of hiding the fact, short of slowing down the other operations to match them, is to do these operations in parallel with other, independent operations. This was the reason for implementing these operations in an asynchronous coprocessor.

Coprocessor operations vary considerably in speed. A value can be moved into or out of the accumulator in one cycle. Floating point addition or subtraction takes four cycles. Double precision division takes 59 cycles. In practice, the faster instructions predominate; the average amount of time it takes to finish a coprocessor instruction in the absence of instruction scheduling is about three and a half cycles.

*The effective fraction is actually a little better than this, because the translator loads or stores a variable twice in the same basic block only when it must. Thus several appearances of a variable in the source program may turn into only one or two memory references. We ignore this effect because it is a well-known compiler technique, and is independent of our attempts to make variables reside in registers.

The scheduled code was instrumented to count the total number of instructions executed, the number of coprocessor instructions executed, the number of extra cycles spent doing coprocessor operations, and the number of these extra cycles that were filled by doing operations in the main processor at the same time. This let us compute the average speed of an instruction, and the average speed of a coprocessor instruction, measured in cycles/instruction.*

	instr speed before	*instr speed after*	*coproc speed before*	*coproc speed after*
Livermore	1.39	1.15	2.51	1.60
Whetstones	2.38	2.21	5.67	5.09
Linpack	1.79	1.19	2.87	1.45
Stanford	1.08	1.02	2.84	1.49
Simulator	1.39	1.27	3.55	2.76
Verifier	1.01	1.00	4.49	1.86

Table 3. Effect on instruction speed of moving cpu operations into coprocessor stall cycles.

Table 3 summarizes these results. The first two columns show the average instruction speeds in cycles/instruction, computed in two different ways. The first column assumes that the operations done in parallel with the coprocessor require cycles of their own; the second column assumes that they come free because they are done in parallel with the coprocessor. The last two columns show the average speed in cycles of a coprocessor instruction, computed the same two ways.

	stall cycles filled
Livermore	60%
Whetstones	12%
Linpack	76%
Stanford	74%
Simulator	31%
Verifier	75%

Table 4. Proportion of potential coprocessor stall cycles that were filled with cpu instructions.

We also computed the fraction of the extra coprocessor cycles were filled with useful operations in the main processor. Table 4 shows these proportions.

*These speeds do not take into account the effects of other kinds of stalls, nor of cache misses, page faults, and the like, but they give us a good measure of the effect we are interested in now.

We usually fill more than half of the coprocessor stall time with useful instructions, with the result that we come very close to an average instruction speed of one cycle. The exception in this benchmark suite is Whetstones, which is very floating-point intensive: since most of the program's work is being done in the coprocessor, there is less cpu work available to fill up the time.

4.3. Delayed branches

We hide the fact that a Titan branch is delayed one cycle by finding an instruction to move into the branch slot, so that the extra cycle does not add to the execution time of the program. The main measure we have of how well we do is therefore the number of slots that we manage to fill.

Conditional branches complicate matters a little, because we might fill the slot with an instruction that is useful only when the branch is taken, or only when it is not. So we consider conditional and unconditional branches separately, and we consider the dynamic behavior of the executing program, so that we can distinguish between instructions that are useful and instructions that are not.

	cond empty	*cond wasted*	*cond useful*	*uncond filled*
Livermore	1%	<1%	99%	100%
Whetstones	32%	6%	62%	>99%
Linpack	11%	6%	83%	99%
Stanford	30%	3%	67%	100%
Simulator	31%	15%	54%	>99%
Verifier	23%	13%	64%	95%

Table 5. Dynamic distribution of conditional and unconditional branch slots after scheduling.

The scheduled code was instrumented to count the instructions executed in branch slots. The results are summarized in Table 5. The first three columns describe the instructions executed after conditional branches: the first column counts null instructions, the second counts non-null instructions that were needlessly executed because the branch went the wrong direction that time, and the third column counts instructions that were usefully executed. (Note that a executing a wasted instruction is no worse than executing an empty slot.) The fourth column counts the non-null instructions executed after unconditional branches; these are always useful.

Thus the certainty of an unconditional jump allows us to fill the slot almost all the time. In fact, the only time we cannot fill the slot of an unconditional branch is when the destination is not known, as when we are jumping into a table. We are able to execute a useful instruction after a conditional jump somewhat less often, but it is easy to do so more than half of the time.

It would be hard for the processor to do better than that by itself. A common hardware approach to the problem is to predict that a backward branch will in fact be taken, and start to execute the destination instruction early, flushing the pipeline if the branch is not taken after all. The scheduler uses the same heuristic, and the results are good: Table 6 shows the analog of the first three columns in Table 5, but includes only backward conditional branches. We nearly always execute a useful instruction in a backward conditional branch slot.

	empty	*wasted*	*useful*
Livermore	<1%	<1%	>99%
Whetstones	<1%	<1%	>99%
Linpack	<1%	<1%	>99%
Stanford	7%	<1%	92%
Simulator	2%	1%	96%
Verifier	11%	9%	80%

Table 6. Dynamic distribution of conditional backward branch slots after scheduling.

4.4. Memory stalls

The Titan's pipeline stalls for one cycle at a load or store instruction if the preceding instruction set the base register of the load or store, or if the preceding instruction was also a store instruction. We can hide these stalls by moving other instructions into them, so that the instruction preceding a load or store does not cause it to stall.

The scheduled code was instrumented to count the stalled and unstalled loads and stores executed. This allowed us to compute the proportions in each case, and let us compute the average speed of loads and of stores, by counting the stalled instructions as two cycles instead of one.*

	unstalled loads	*unstalled stores*	*load speed*	*store speed*
Livermore	89%	99%	1.11	1.01
Whetstones	88%	98%	1.12	1.02
Linpack	66%	99%	1.34	1.01
Stanford	72%	80%	1.28	1.19
Simulator	94%	71%	1.06	1.29
Verifier	77%	86%	1.23	1.14

Table 7. Proportions of loads and stores that are unstalled, and average speeds of loads and stores.

Table 7 summarizes these results. The first column shows the proportion of executed loads that are not stalled; the second column shows it for stores. Most loads and stores happen without stalls. The third and fourth columns show the average execution speed in cycles/instruction for loads and stores. While we have not hidden the possibility of memory stalls completely, the difference is usually small enough to ignore.

*Again, these computations ignore the effects of cache misses and page faults, which are independent of pipeline stalls.

5. Loopholes

Sometimes you really do need precise control over the machine. You might need to write the absolutely best possible version of a library routine that will be executed millions of times, like input/output conversion. Or you might need to write a routine that is responsible for directly maintaining part of the machine state, like an interrupt handler.

Mahler has two mechanisms for getting this kind of fine control over the machine: builtin procedures, and startup procedures.

A builtin procedure is called in Mahler just like a normal procedure. The Mahler translator recognizes the name of the builtin procedure and generates inline code for it instead of generating a call. Builtin procedures are used to implement operations that cannot be expressed (or cannot be expressed efficiently enough) in normal Mahler. Because the builtin procedure call looks just like a normal call, it can be made from a high-level language as well as from a Mahler routine; the front-end compiler will pass the call through to the Mahler translator, which will intercept it. The name of the builtin procedure is therefore essentially a Mahler keyword and cannot be redefined by the user.

In defining Mahler builtins we have tried to find out what problem the user was really trying to solve, and design a builtin that would satisfy the user but would also let Mahler know what was going on so that its analysis would not be impaired. Thus we have shied away from builtins that simply let the user examine or change a specific register or the like. Such builtins are occasionally useful to let the user decide what is really needed, but these low-level loopholes are used rarely and are replaced by more semantically meaningful builtins when we understand the problem better.

Thus Mahler has builtin procedures to save and restore all or part of the register state, which can be used in interrupt handling. It has builtin procedures to access certain extended-precision arithmetic instructions, for use by the input/output conversion routines; these routines would likely have to be changed for a new generation of Titan, and perhaps it would be better to make the entire conversion routine be builtin. Even the C library routines of setjmp, longjmp, and alloca are Mahler builtins, because Mahler must know when these things happen if it is to optimize data references safely.

A startup procedure has no name and is never called by another routine; some piece of magic is needed, either by hardware intervention (for operating system code) or by operating system activity (for user code) to give this routine control. A startup procedure has no local variables, and must ensure that a valid data environment exists before it can call a normal procedure or reference a global variable.

The Unix* operating system was experimentally ported to the Titan using only these two Mahler extensions. Even the port in use includes only 1000 instructions of ‘‘true’’ assembler code, which switches processes to handle interrupts.

6. Conclusions

Using Mahler as the official description of the Titan has worked fairly well. It is hard to know how much programmer time has been saved by eliminating the temptation to program in assembler and worry about delayed branches and the coprocessor, but it seems clear that this benefit is not small. By keeping most of our

*Unix is a trademark of Bell Laboratories.

variables in registers and by scheduling instructions to hide the vagaries of the pipeline, coprocessor, and delayed branches, we have managed to hide the worst of the idiosyncrasies of the real machine.

A final measure of the effectiveness of the approach is that we recently changed to a second generation of the Titan, one which is byte-addressed rather than word-addressed; although this caused some changes to the front end compilers and even to the definition of Mahler, the change was relatively painless on both counts. Mahler has also been retargeted to a substantially different RISC processor being developed here, with a different number of registers and a very different pipeline and coprocessor structure. Although this required some work, it presented no major problems, and the change to the front-end compilers was trivial.

Acknowledgements

Joel Bartlett, Jeremy Dion, Mary Jo Doherty, and Bill Hamburgen read and criticized early drafts of this paper, substantially improving it. Our thanks.

References

[1] Robert Bernstein. Multiplication by integer constants. *Software -- Practice and Experience 16* (7): 641-652 (July 1986).

[2] Jack J. Dongarra. Performance of various computers using standard linear equations software in a Fortran environment. *Computer Architecture News 11* (5): 22-27, December 1983.

[3] R. Steven Glanville and Susan L. Graham. A new method for compiler code generation. *Fifth Annual ACM Symposium on Principles of Programming Languages:* 231-240 (January 1978).

[4] John Hennessy. Stanford benchmark suite. Personal communication.

[5] John L. Hennessy and Thomas R. Gross. Code generation and reorganization in the presence of pipeline constraints. *Ninth Annual ACM Symposium on Principles of Programming Languages:* 120-127 (January 1982).

[6] John L. Hennessy, Norman P. Jouppi, Steven Przybylski, Christopher Rowen, and Thomas Gross. Design of a high performance VLSI processor. In Randal Bryant, editor, *Third Caltech Conference on Very Large Scale Integration,* pages 33-54. Computer Science Press, 11 Taft Court, Rockville, Maryland.

[7] David A. Patterson. Reduced instruction set computers. *Communications of the ACM 28* (1): 8-21, January 1985.

[8] Michael L. Powell. Efficient dynamic measurement of deterministic run-time behavior. In preparation.

[9] George Radin. The 801 minicomputer. *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems, pages 39-47 (March 1982).* Published as *SIGARCH Computer Architecture News 10* (2), March 1982, and as *SIGPLAN Notices 17* (4), April 1982.

[10] Christopher J. Terman. *User's Guide to NET, PRESIM, and RNL/NL.* M.I.T. Laboratory for Computer Science, 545 Technology Square, Room 418, Cambridge, Massachusetts.

[11] David W. Wall. Global register allocation at link-time. *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction.* Published as *SIGPLAN Notices 21* (7): 264-275 (July 1986).

[12] Niklaus Wirth. The programming language Pascal. *Acta Informatica 1* (1): 35-63 (1971).

[13] Niklaus Wirth. The design of the Pascal compiler. *Software -- Practice and Experience 1* (4): 309-333 (1971).