Chapter 7
The Potential of Special-Purpose Hardware

The preceding chapters have described various implementation methods and performance data for TIGRE. This chapter uses those data points to propose architecture and implementation features which could be used to speed up the execution of TIGRE. The reason for examining such features is to determine the feasibility of constructing special-purpose hardware, or, if construction of special-purpose hardware is not attractive, the features that should be selected when choosing standard hardware to execute TIGRE.

Section 7.1 discusses using the DECstation 3100 as a baseline for consideration. Section 7.2 discusses improvements that are possible by modifying the cache management strategy of the 3100. Section 7.2 discusses improvements that are possibly by modifying other architectural parameters of the 3100 and its MIPS R2000 processor. Section 7.4 summarizes the results of the discussions in the chapter.

7.1. DECSTATION 3100 AS A BASELINE

Since many of the performance measurements for TIGRE are expressed in terms of the MIPS R2000 assembly language implementation, and since the R2000 is a reasonably efficient processor, the approach used for examining processor features to support TIGRE will be made in terms of incremental modifications to the MIPS R2000 processor. This approach will give a rough estimate for the potential performance improvement, while maintaining some basis in reality. These possibilities for change should not be construed as implying that such a modified processor should (necessarily) be built, but rather taken as a method for approximating the value of adding certain features to conventional processors.

The baseline performance of the MIPS R2000 as implemented in the DECstation 3100 was discussed in Chapter 6. For the purposes of

Instructions per combinator	27.82
clock cycles per combinator	29.47
bus-limited clock cycles per combinator	30.09
cache read misses per combinator	0.33
bus writes per combinator	4.74
traffic ratio	0.59
heap nodes per combinator	0.74
spine unwinds per combinator	1.37
stack accesses per combinator	4.61

Table 7-1. Summary of TIGRE DECstation 3100 performance charac-

the following performance analysis, the characteristics of the **SKI** implementation of the fib benchmark shall be used. A summary of these characteristics, given in Table 7-1, will be used as the basis for estimating performance improvements possible with architectural modifications.

Since TIGRE has been shown to have some unusual cache access behavior, the first area for improvement that will be considered is improving the arrangement of cache memory. Then, improvements in the instruction set of the R2000 will be considered.

7.2. IMPROVEMENTS IN CACHE MANAGEMENT

7.2.1. Copy-Back Cache

The most obvious limitation of the DECstation 3100 cache is that it uses a write-through memory update strategy. This caused the limiting performance factor to be bus bandwidth for memory write accesses, instead of instruction read or data read miss ratios. A simple improvement, then, is to employ a copy-back cache. A cache simulation of fib for the DECstation 3100 shows that this reduces the data cache traffic ratio from 0.59 to 0.29, removing the bus bandwidth as the limiting factor to performance. This reduces the execution time of an average combinator from 30.09 clock cycles (the bus bandwidth-limited performance) to 29.47 clock cycles (the cache hit ratio-limited performance).

7.2.2. Increased Block Size

A second parameter of the cache that could be improved is the block size. TIGRE executes well with a large block size, so increasing the cache block size from 4 bytes to, say 256 bytes, should dramatically decrease the cache miss ratio. A simulation using a block size of 256 bytes (and assuming a 256 byte data bus) reduces the data cache miss ratio from 0.1103 to 0.0046, yielding a performance of 27.96 clock cycles per combinator. Unfortunately, the R2000 has a 32-bit bus, and pin count limitations prohibit the use of a very wide memory bus.

Using a wide bus write buffer with a 4 byte cache block size can capture many of the benefits of a large block size, and reduce bus traffic. A write buffer width of 8 bytes (one full graph node) can be utilized efficiently by a supercombinator compiler to get a very high percentage of paired 4-byte writes to the left- and right-hand sides of cells when updating the graph.

Even if a very sophisticated cache mechanism were used to reduce cache misses to the absolute minimum possible (ideally, 0.0000 miss ratio), the speedup possibilities are somewhat small, since only 1.65 clock cycles of the 29.47 clock cycles per combinator are spent on cache misses to begin with.

7.2.3. Prefetch on Read Misses

One variation of an increased block size is to use prefetch logic to fetch the succeeding word in memory after each access. There are various ways of determining when to fetch particular words, but the one of interest here is the strategy of fetching a word into cache only after a read generates a cache miss *and* the cache read miss was to an even-addressed word location.

Data cache read misses to even-addressed words (words which align on 8-byte boundaries) specifically occur whenever a threading operation is taking place. As threading takes place, the left-hand cells of nodes are read to traverse the program graph. Since left-hand cells are aligned on even word boundaries by TIGRE, these prefetches will automatically fetch the right-hand sides of nodes being threaded into the cache. The right-hand sides will probably be read as input arguments to one of the next few combinators executed. This decreases the overall cache miss ratio.

Unfortunately, the cache simulator cannot be coerced into simulating this very specific cache miss behavior. It is estimated that the performance speedup obtainable is probably rather small, since the high temporal locality of accessing program graphs ensures that the right-hand sides of nodes will probably already be cache resident (not having been flushed from cache since they were created) when the threading operation takes place.

7.3. IMPROVEMENTS IN CPU ARCHITECTURE

The opportunities for improvement by changing the architecture of the R2000 are somewhat more promising than those possible by modifying the cache management strategy. In particular, it is possible to significantly increase the speed of stack unwinding and performing indirections through the stack elements.

7.3.1. Stack Unwinding Support

The one serious drawback of the MIPS R2000 architecture for executing TIGRE is the lack of a subroutine call instruction. The current TIGRE implementation on the R2000 uses a five-instruction interpretive loop for performing threading (*i.e.* stack unwinding). Since 1.37 stack unwind operations are performed per combinator, this presents 6.85 instructions which, assuming no cache misses, execute in 6.85 clock cycles.

But, there is a further penalty for performing the threading operation through graphs with the R2000. A seven-instruction overhead is used for each combinator to perform a preliminary test for threading, and to access a jump table to jump to the combinator code when threading is completed.* This imposes an additional 7.00 clock cycle penalty on each combinator.

So, the total time spent on threading is 13.85 clock cycles per combinator. It takes three clock cycles to simulate a subroutine call on the R2000:

```
sw     $31, 0($sp)   # store current return address
jal    subr_address  # subroutine call
       # branch delay slot instruction follows
addu   $sp, $sp, -4  # decrement stack pointer
```

so it is reasonable to assume that a hardware-implemented subroutine call instruction could be made to operate in three clock cycles. Thus, if

* One of these instructions increments a counter used for performance measurement. It can be removed for production code, as long as measuring the number of combinators executed is not important.

the instruction cache were made to track writes to memory (permitting the use of self-modifying code), a savings of 10.85 clock cycles is possible. One important change to the instruction set would be necessary to allow the use of subroutine call instructions — the subroutine call instruction would have to be defined to have all zero bits in the opcode field (so that the instruction could be used as a pointer to memory as well).

An alternate strategy that does not require the use of self-modifying code is to incorporate a subroutine call instruction that takes its address from the next location in memory, but fetches that location from the *data cache* instead of the instruction cache. Because the subroutine call addresses are the only information in the instruction stream subject to modification, this strategy would eliminate problems with instruction cache consistency.

As suggestive evidence that support for fast stack unwinding can greatly improve performance, consider that the 10 MHz Harris RTX 2000 was almost as fast as the 16.7 MHz DECstation 3100 in Table 5-1, despite the fact that the RTX 2000 takes two clock cycles for memory loads and stores compared to the R2000's one clock cycle per memory access (for cache hits). This competitiveness was in large part due to the RTX 2000's hardware support of single-cycle subroutine calls, and use of an on-chip subroutine return stack buffer.

7.3.2. Stack Access Support

An important aspect of TIGRE's operation is that it makes frequent reference to the top elements on the spine stack. In fact, 4.61 reads to the spine stack are performed per average combinator. Most of the load and store instructions that perform these stack accesses can be eliminated by the use of on-CPU stack buffers that are pushed and popped as a side effect of other instructions.

For spine stack unwinding, two of the three instructions could be eliminated with the use of hardware stack support, leaving just a single `jal` instruction to perform the threading operation at each node. Of course, the R2000 has a built-in branch delay slot, so it is not probable that the actual time for the threading operation could be reduced to less than two clock cycles. But, the second clock cycle could be used to allow writing a potential stack buffer overflow element to memory. This technique results in a savings of 1.37 clock cycles (1 clock cycle per thread operation, with 1.37 stack unwinds per combinator) for an average combinator.

Of the 4.61 instructions that access the spine stack, the threading technique just described may be used to eliminate the effect of 1.37 of the instructions. The remaining 3.24 instructions can also be eliminated by introducing an indirect-through-spine-stack addressing mode to the R2000. All that would be required is to access the top, second, and third element of a spine stack buffer as the source of an address instead of a register. A simple implementation method could map the top of stack buffer registers into the registers already available on the R2000. This gives a potential savings of 3.24 clock cycles, since explicit load instructions from the spine stack need not be executed when performing indirection operations.

Of course, the issue of stack overflows and underflows arises whenever one discusses the issue of hardware stack buffers. Overflows and underflows with the described techniques should cost almost nothing. Overflows can only happen during threading, which specifically allocates the branch delay slot of the `jal` instruction to handle a potential write to memory. Underflows can only happen one element at a time, if TIGRE code is assembled so as to pop stack elements as they are used (as the TIGRE compiler does for VAX code). If only a single element is overflowed, this overflow can be handled in a single clock cycle. With a reasonably large stack buffer (16 or 32 elements), such overflows tend to occur on less than 1% or 2% of stack accesses for stack-based programs in general (Koopman 1989).

7.3.3. Doubleword Store

The astute reader has noticed that TIGRE is usually able to write cells in pairs, with both the left- and right-hand cells of a single node written at approximately the same point in the code for a particular combinator. Thus, it becomes attractive to define a "double store" instruction format. Such an instruction would take two source register operands (for example, an even/odd register pair), and store them into a 64-bit memory doubleword. If the processor were designed with a 64-bit memory bus, such a "double store" could take place in a single clock cycle instead of as a two-clock sequence. The savings of using 64-bit stores is 0.895 clock cycles per combinator for the **SKI** implementations of fib, and 1.192 clock cycles per combinator for the Turner Set implementation of fib (measured by instrumenting TIGRE code to count the opportunities for these stores as the benchmark program is executed). Support of 64-bit memory stores would speed up supercombinator definitions even more, since the body of supercombinators often contains long sequences of node creations. For example, the supercombinator implementation of fib can make use of 1.33 64-bit stores per combinator.

optimizations	clocks per combinator
current implementation	30.09
copy-back cache	29.47
100% cache hit ratio	27.82
subroutine call through data cache	16.97
hardware stack for `jal`	15.60
hardware stack indirect addressing	12.36
8-byte store instructions	11.47

Table 7-2. Summary of possible performance improvements.

7.4. PERFORMANCE IMPROVEMENT POSSIBILITIES

The previous suggestions for optimizations present some good news and some bad news. The good news is that special hardware support for TIGRE can result in substantial speedups, as shown in Table 7-2. Most of this speedup comes from providing hardware stack support and support for fast operations on 64-bit quantities. A 100% cache hit ratio is assumed for the third and subsequent performance figures to present a best-case speedup.

The bad news is that the potential speedup available is substantially less than an order of magnitude. In fact, an order of magnitude speed improvement is rather unlikely no matter what architectural innovations are possible, since a factor of 10 speedup from the current R2000 TIGRE implementation leaves just 3.009 clock cycles per combinator for program execution time. Speeding up TIGRE operation by that amount exceeds all plausible expectations. So, it is probably not worthwhile building a special-purpose CPU to support TIGRE, since current RISC technology will probably have increased in speed enough by the time a TIGRE chip could be designed and fabricated to make the exercise pointless.

From a positive aspect, the exercise of identifying architectural features to speed up TIGRE has not been a complete waste of time. The features described definitely have potential to support high-speed execution of TIGRE. Therefore, instead of building custom TIGRE silicon, TIGRE developers should look for commercially produced architectures that provide some of the features desired. For example, several stack-based processors designed to support the Forth programming language provide hardware stack support and single-cycle subroutine calls. Examples of 32-bit processors with such support include the RTX-32P from

Harris Semiconductor and the FRISC 3 from Johns Hopkins/Applied Physics Laboratory (Koopman 1989). Support for 64-bit store operations and copy-back caches also exists on many mainframes, and will probably be supported by workstation-class computers soon. So, the task of designing special-purpose TIGRE hardware is best accomplished by collecting and reviewing a large library of product information sheets from manufacturers of commercial hardware, and choosing wisely.