The Rewards of Generating True 32–bit Code

Trading Additional Compiler Complexity for Better Performance

Michael Franz
Institut für Computersysteme
ETH Zürich, Switzerland
franz@inf.ethz.ch

Many of today's system architectures are actually extensions of older ones. One such growth path within processor families applies to the width of the data bus. While earlier processors often already incorporated relatively broad internal registers, these registers had to be read and written in multiple cycles over a narrow bus. The modern successors of these processors offer upwardly compatible instruction sets and greater bus widths, reflecting the widths of the internal registers.

Utilizing a wide bus and a memory port of corresponding width, a new factor suddenly influences performance, *data alignment*, which was irrelevant while there was only a narrow bus. Data that is lined up with the memory port can be read in a single bus cycle; extra bus cycles are required for misaligned data accesses. Taking the Apple Macintosh II and the programming language *Oberon* as an example, we show that there is a remarkable payoff for generating code that avoids misaligned data accesses, and how to achieve this under constraints to be backward–compatible with existing software. It should be noted that backward compatibility with older *processors* is not an issue here: all programs optimized for wide–bus processors in respect to data alignment will run just as well on narrow–bus processors of the same family; they will simply run faster on the former than on the latter.

The original Macintosh employed a Motorola 68000 microprocessor, which has a 16–bit data bus. The interface to its operating system specifies *word alignment* of data structures and parameters. As a result, on average one half of all longword accesses are misaligned on Macintosh computers that have 32–bit data busses, slowing down computation. An on–chip data cache can somewhat alleviate this effect, but cannot completely compensate for it.

While this state of affairs cannot be mended in respect to the *operating system interface* without invalidating all existing software, one can in fact, at some additional expense in compiler complexity, improve the quality of *application code* generated for a 32–bit–data–bus machine. We have created a compiler that differentiates between code that must be backward–compatible with the existing 16–bit interface, and more efficient "*genuine 32–bit*" code, which is generated for new routines. This applies to the layout of data structures and to the procedure calling mechanism.

Macintosh operating system procedures are activated by executing characteristic *unimplemented processor instructions*, which trigger a processor exception and are then emulated in software. Since calls to the operating system are represented by single processor instructions, some construct at the language level is required for declaring the processor instruction that is associated with a particular procedure. The compiler thus necessarily has to distinguish between user–defined procedures and operating system interface procedures; instead of offering only a special means of *procedure activation* for interface procedures, it might just as well support a complete alternate set of conventions for the internal structuring of their parameter types and for the calling mechanism, liberating us to choose a more suitable protocol elsewhere.

Using a modular programming language such as Oberon, the distinction between the two varieties of data types and procedures may be concealed completely from application programmers, by encapsulating all interface procedures and the data structures they operate on in so–called *interface modules*. Even *type extension* of data types exported by interface modules is legitimate, in which case the additional fields of the extension are 32–bit aligned for optimal access speed, while the common fields have a backward–compatible 16–bit internal structure; projections of such hybrid types onto their respective (16–bit) base types may therefore be passed to interface procedures.

As long as we support the operating system's calling conventions for calling interface procedures, we are free to define arbitrary protocols for application procedures calling each other. Since function procedures in Oberon may only return basic types or pointers (just as in Pascal), we have opted to return

function results in dedicated registers rather than on the stack. Our calling protocol further differs from the traditional Macintosh conventions, in that we enforce longword–alignment of the stack pointer, except for short intervals when operating system routines are called.

Longword–alignment of the stack pointer is achieved by pushing word and byte–size parameters as longwords and by padding stack frames to longword boundaries. Parameters are thus always longword–aligned, whereas local and global variables need only be aligned by their size. Longword alignment of parameters is necessary, because the actual parameters might be calculated by calling function procedures; the stack must thus reside on a longword boundary at the end of each individual parameter assignment.

As an example, consider the following declarations:

```
VAR n: INTEGER;
PROCEDURE P(i, j: INTEGER): INTEGER;
```

Now study the expression on the next line, which contains a call to function procedure P:

```
n := P(1234H, n)
```

If P is an interface procedure, then our compiler will output the following instruction sequence:

```
SUBQ.L #2, SP                ; reserve space for result
MOVE.W #$1234, -(SP)         ; first parameter
MOVE.W n, -(SP)              ; second parameter
JSR P
MOVE.W (SP)+, n              ; pop result
```

Otherwise, if P is an ordinary procedure, the compiler will generate the object code listed below:

```
MOVE.L #$12340000, -(SP)     ; first parameter is constant shifted to correct alignment
SUBQ.L #4, SP                ; keep stack longword aligned
MOVE.W n, (SP)               ; second parameter
JSR P
MOVE.W D0, n                 ; always return results in registers
```

In spite of the fact that the processor has to execute more instructions (explicit decrementation of the stack pointer is generally necessary) and move greater amounts of data (constants are always pushed as longwords), procedure calls that follow the second protocol actually *execute faster* than those that employ the traditional Macintosh calling conventions. In order to measure the extent of this performance improvement, we ran a classic suite of benchmarks (originally collected by John Hennessy) for each of the calling and data alignment schemes. Except for the instructions for parameter and result passing, all other operations in the programs compared below were identical.

Three different benchmarks were run. First of all (Situation A), we timed the programs utilizing the customary (operating system) calling conventions (word alignment of data, parameters pushed by size). We then incremented the stack pointer by two and ran the same series of benchmarks again (Situation B). Very different timings resulted, some better and some worse, because all longword accesses now had different alignments than before. We then related the average of these two timings to the values obtained using "genuine 32–bit" compilation (data aligned by size, longword alignment of parameters). The following table lists the time in milliseconds for each of these benchmarks, on an Apple Macintosh II computer (MC68020/16MHz), as well as the performance of the "32–bit" code relative to the average performance of "word–oriented" code under two different stack alignments.

	Operating System Conventions		Genuine 32–bit	Relative
	Situation A	Situation B	Conventions	Performance
Permutation	800	1150	800	122%
Towers of Hanoi	1283	1500	1200	116%
Eight Queens	850	633	633	117%
Integer MatMult	1166	1416	1166	111%
Real MatMult	2000	2233	1967	108%
Puzzle	6784	6800	6033	113%
Quicksort	717	950	700	119%
Bubblesort	1284	1583	1283	112%
Treesort	733	983	750	114%
Fast Fourier Transform	3216	3700	3200	108%

While object files that support "genuine 32–bit" conventions are longer by up to 5% (due to the expansion of word–sized literals to longword and the need to decrement the stack pointer explicitly when passing byte or word–sized parameters), the code thus generated *outperforms consistently* the word–oriented equivalent that is so commonly produced by other compilers. This improvement becomes less pronounced on machines including a processor data cache; on a Macintosh IIci (MC68030/25MHz) the average relative performance drops to 110% and on a Macintosh IIfx (MC68030/40MHz/64kByte cache) it plunges to 105%. However, we feel that the overall performance gain is sufficiently large to justify the increase in code size and additional compiler complexity.

References

[1] Apple Computer, Inc.: Inside Macintosh (Volumes I – V).
Addison–Wesley, Reading, Massachusetts, 1985–88.

[2] Motorola, Inc.: MC68030 Enhanced 32–Bit Microprocessor User's Manual.
Prentice Hall, Englewood Cliffs, New Jersey, 1987.

[3] N. Wirth: Type Extensions.
ACM Transactions on Programming Languages and Systems, 10, 2 (April 1988), 204–214.

[4] N. Wirth: The Programming Language Oberon.
Software – Practice and Experience, 18, 7 (July 1988), 671– 690.