

Compilation Strategies for Reducing Code Size on a VLIW Processor with Variable Length Instructions

Todd T. Hahn, Eric Stotzer, Dineel Sule, Mike Asal

Texas Instruments Incorporated
12203 Southwest Freeway, Stafford, TX 77477
{tthahn, estotzer, dineel, m-asal}@ti.com

Abstract. This paper describes the development and compiler utilization of variable length instruction set extensions to an existing high-performance, 32-bit VLIW DSP processor. We describe how the instruction set extensions (1) significantly reduce code size, (2) are binary compatible with older object code, (3) do not require the processor to switch “modes”, and (4) are exploited by a compiler. We describe the compiler strategies that utilize the new instruction set extensions to reduce code size. When compiling our benchmark suite for best performance, we show that our compiler uses the variable length instructions to decrease code size by 11.5 percent, with no reduction in performance. We also show that our implementation allows a wider code size and performance tradeoff range than earlier versions of the architecture.

1 Introduction

VLIW (very long instruction word) processors are well-suited for embedded signal and video processing applications, which are characterized by mathematically oriented loop kernels and abundant ILP (instruction level parallelism). Because they do not have hardware to dynamically find implicit ILP at run-time, VLIW architectures rely on the compiler to statically encode the ILP in the program before its execution [1]. Since ILP must be explicitly expressed in the program code, VLIW program optimizations often replicate instructions, increasing code size. While code size is a secondary concern in the computing community overall, it can be significant in the embedded community. In this paper we present an approach for reducing code size on an embedded VLIW processor using a combination of compiler and instruction encoding techniques.

VLIW processors combine multiple instructions into an execute packet. All instructions in an execute packet are issued in parallel. Code compiled for a VLIW will often include many NOP instructions, which occur because there is not enough ILP to completely fill an execute packet with useful instructions. Since code size is a concern, instruction encoding techniques have been developed to implicitly encode VLIW NOP instructions [2]. Another approach to reduce code size is to store a compressed image of the VLIW program code in external

memory, and use run-time software or hardware to decompress the code as it is executed or loaded into a program cache [3, 4]. Other approaches have used variable length instruction encoding techniques to reduce the size of execute packets [5]. Finally, recognizing that code size is often the priority in embedded systems, other approaches use processor modes with smaller opcode encodings for a subset of frequently occurring instructions. Examples of mode-based architectures are the ARM Limited ARM architecture’s Thumb mode [6, 7] and the MIPS Technologies MIPS32 architecture’s MIPS16 mode [8].

In this paper, we discuss a hybrid approach for reducing code size on a VLIW, using a combination of compilation techniques and compact instruction encodings. We provide an overview of the TMS320C6000 (C6x) family of VLIW processors and how instructions are encoded to reduce the code size. We then describe a unique modeless binary compatible encoding for variable length instructions that has been implemented on the TMS320C64+ (C64+) processor, which is the latest member of the C6x processor family. We describe how, consistent with the VLIW architecture philosophy, the compact instruction encoding is directed by the compiler. We discuss how the different compiler code size strategies leverage the compact instructions to reduce program code size and balance program performance. Finally, we conclude with results showing how program code size is reduced on a set of typical DSP applications. We also show that our implementation allows a significantly wider range of size and performance tradeoffs versus earlier versions of the architecture.

2 The TMS320C6000 VLIW DSP Core

The TMS320C62x (C62) is a fully pipelined VLIW processor, which allows eight new instructions to be issued per cycle. All instructions can be optionally guarded by a static predicate. The C62 is the base member of the Texas Instruments’ C6x family (Fig. 1), providing a foundation of integer instructions. It has 32 static general-purpose registers, partitioned into two register files. A small subset of the registers may be used for predication. The TMS320C64x (C64) builds on the C62 by removing scheduling restrictions on existing instructions and providing additional instructions for packed-data/SIMD (single instruction multiple data) processing. It also extends the register file by providing an additional 16 static general-purpose registers in each register file [9].

The C6x is targeted toward embedded DSP (digital signal processing) applications, such as telecom and image processing. These applications spend the bulk of their time in computationally intensive loop nests which exhibit high degrees of ILP. Software pipelining, a powerful loop-based transformation, is key to extracting this parallelism and exploiting the many functional units on the C6x [10].

Each instruction on the C6x is 32-bit. Instructions are fetched eight at a time from program memory in bundles called *fetch packets*. Fetch packets are aligned on 256-bit (8-word) boundaries. The C6x architecture can execute from one to eight instructions in parallel. Parallel instructions are bundled together

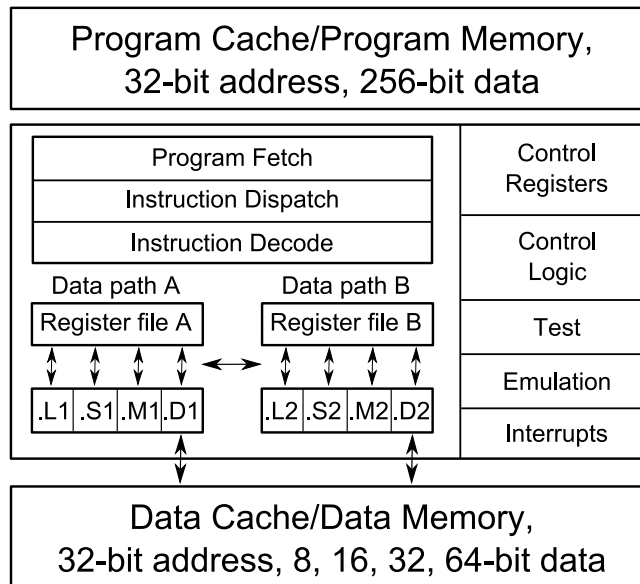


Fig. 1. TMS320C6000 architecture block diagram

into an *execute packet*. As fetch packets are read from program memory, the instruction dispatch logic extracts execute packets from the fetch packets. All of the instructions in an execute packet execute in parallel. Each instruction in an execute packet must use a different functional unit.

The execute packet boundary is determined by a bit in each instruction called the *parallel-bit* (or *p-bit*). The p-bit (bit 0) controls whether the next instruction executes in parallel.

On the C62, execute packets cannot span a fetch packet boundary. Therefore, the last p-bit in a fetch packet is always set to 0, and each fetch packet starts a new execute packet. Execute packets are padded with parallel NOP instructions by the assembler to align spanning execute packets to a fetch packet boundary. The C64 allows execute packets to span fetch packet boundaries with some minimal restrictions, thus reducing code size by eliminating the need for parallel padding NOP instructions.

Except for a few special case instructions such as the NOP, each instruction has a predicate encoded in the first four bits. Figure 2 is a generalization of the C6x 32-bit three operand instruction encoding format. The predicate register is encoded in the condition (creg) field, and the z-bit encodes the true or not-true sense of the predicate. The dst, src2, and src1 fields encode operands. The x-bit encodes whether src2 is read from the opposite cluster's register file. The op field encodes the operation and functional unit. The s-bit specifies the cluster that the instruction executes on.

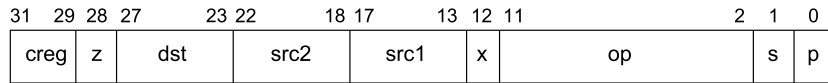


Fig. 2. Typical 32-bit instruction encoding format

3 C64+ Compact Instructions

The C64+ core has new instruction encodings that (1) reduce program size, (2) allow binary compatibility with older object files, (3) and do not require the processor to switch modes to access the new instructions encodings. In addition, the existing C/C++ compiler exploits these new instruction encodings without extensive modification.

3.1 Compact 16-bit Instructions

A 16-bit instruction set was developed in which each 16-bit instruction is a compact version of an existing 32-bit instruction. All existing control, data path, and functional unit logic beyond the decode stage remains unchanged with respect to the 16-bit instructions. The 16-bit and 32-bit instructions can be mixed.

The ability to mix 32- and 16-bit instructions has several advantages. First, an explicit instruction to switch between instruction sets is unnecessary, eliminating the associated performance and code size penalty. Second, algorithms that need more complex and expressive 32-bit instructions can still realize code size savings since many of the instructions can be 16-bit. Finally, the ability to mix 32- and 16-bit instructions in the C64+ architecture frees the compiler from the complexities associated with a processor mode.

The 16-bit instructions selected are frequently occurring 32-bit instructions that perform operations such as addition, subtraction, multiplication, shift, load, and store. By necessity, the 16-bit instructions have reduced functionality. For example, immediate fields are smaller, there is a reduced set of available registers, the instructions may only operate on one functional unit per cluster, and some standard arithmetic and logic instructions may only have two operands instead of three (one source register is the same as the destination register). All 16-bit instructions do not have a condition operand and execute unconditionally except for certain branch instructions.

The selection and makeup of 16-bit instructions was developed by rapidly prototyping the existing compiler, compiling many different DSP applications, and examining the subsequent performance and code size. In this way, we were able to compare and refine many different versions of the 16-bit instruction set. Due to design requirements of a high performance VLIW architecture, the C6x 32-bit instructions must be kept on a 32-bit boundary. Therefore, the 16-bit instructions occur in pairs in order to honor the 32-bit instruction alignment.

3.2 The Fetch Packet Header

The C64+ has a new type of fetch packet that encodes a mixture of 16-bit and 32-bit instructions. Thus, there are two kinds of fetch packets: A standard fetch packet that contains only 32-bit instructions and a header-based fetch packet that contains a mixture of 32- and 16-bit instructions. Figure 3 shows a standard fetch packet and an example of a header-based fetch packet. Fetch packet headers are detected by looking at the first four bits of the last word in a fetch packet. A previously unused *creg/z* value indicates that the fetch packet is header-based. The header-based fetch packet encodes how to interpret the bits in the rest of the fetch packet. On C64+, execute packets may span standard and header-based fetch packets.

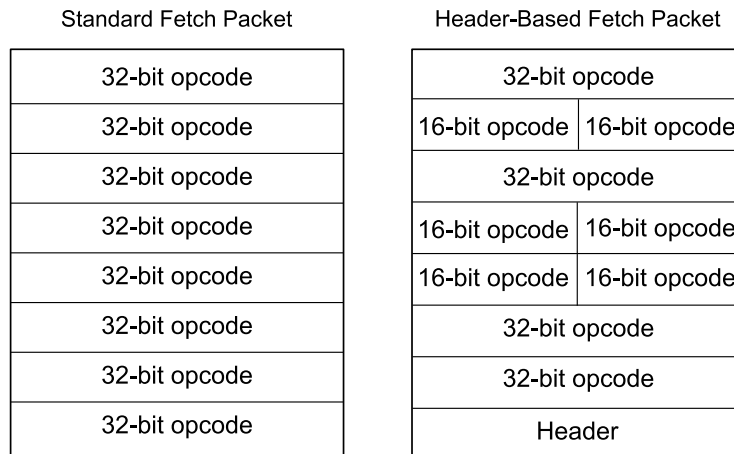


Fig. 3. C64+ fetch packet formats

Figure 4 shows the layout of the fetch packet header. The predicate field used to signify a fetch packet header occupies four bits (bits 28-31). There are seven *layout bits* (bits 21-27) that designate if the corresponding word in the fetch packet is a 32-bit instruction or a pair of 16-bit instructions. Bits 0-13 are p-bits for 16-bit instructions. For a 32-bit instruction, the corresponding two p-bits in the header are not used (set to zero). The remaining seven *expansion bits* (bits 14-20) are used to specify different variations of the 16-bit instruction set.

The expansion bits and p-bits are effectively extra opcode bits that are attached to each instruction in the fetch packet. The compressor software (discussed in section 4) encodes the expansion bits to maximize the number of instructions in a fetch packet.

The protected load instruction bit (bit 20) indicates if all load instructions in the fetch packet are protected. This eliminates the NOP that occurs after a load instruction in code with limited ILP, which is common in control oriented

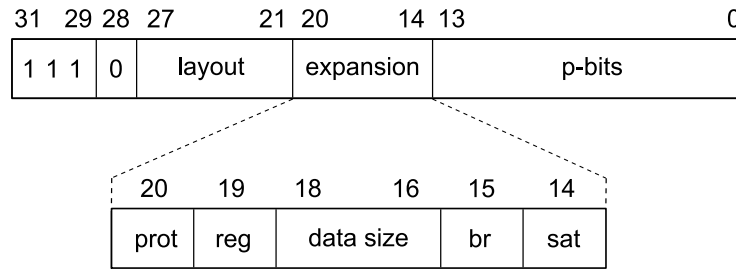


Fig. 4. Compact instruction header format

code. The register set bit (bit 19) indicates which set of eight registers is used for three operand 16-bit instructions. The data size field (bits 16-18) encodes the access size (byte, half-word, word, double-word) of all 16-bit load and store instructions. The branch bit (bit 15) controls if branch instructions or certain S-unit arithmetic and shift instructions are available. Finally, the saturation bit (bit 14) indicates if many of the basic arithmetic operations saturate on overflow.

3.3 Branch and Call Instructions

Certain branch instructions appearing in header-based fetch packets can reach half-word program addresses. Ensuring that branch instructions can reach intended destination addresses is handled by the compressor software.

A new 32-bit instruction, CALLP, can be used to take the place of a B (branch) instruction and an ADDKPC (set up return address) instruction. However, unlike branch instructions, where the five delay slots must be filled with other instructions or NOPs, the CALLP instruction is “protected,” meaning other instructions cannot start in the delay slots of the CALLP. The use of this instruction can reduce code size up to six percent on some applications with a small degradation in performance.

4 The Compressor

When compiling code for C64+, an instruction’s size is determined at assembly-time. (This is possible because each 16-bit instruction has a 32-bit counterpart.) The *compressor* runs after the assembly phase and is responsible for converting as many 32-bit instructions as possible to equivalent 16-bit instructions. The compressor takes a specially instrumented object file (where all instructions are 32-bit), and produces an object file where some instructions have been converted to 16-bit instructions. Figure 5 depicts this arrangement. Code compression could also be performed in the linker since the final addresses and immediate fields of instructions with relocation entries are known and may allow more 16-bit instructions to be used.

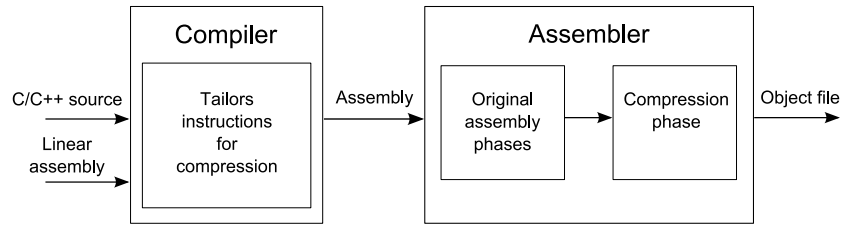


Fig. 5. Back-end compiler and assembler flow depicting the compression of instructions

The compressor also has the responsibility of handling certain tasks that cannot be performed in the assembler because the addresses of instructions will change during compression. In addition, the compressor must fulfill certain architecture requirements that cannot be easily handled by the compiler. These requirements involve pairing 16-bit instructions, adding occasional padding to prevent a stall situation, and encoding the fetch packet header when 16-bit instructions are present.

Compression is an iterative process consisting of one or more *compression iterations*. In each compression iteration the compressor starts at the beginning of the section's instruction list and generates new fetch packets until all instructions are consumed. Each new fetch packet may contain eight 32-bit instructions (a regular fetch packet), or contain a mixture of 16- and 32-bit instructions (a header-based fetch packet).

The compressor must select an *overlay* which is an expansion bit combination used for a fetch packet that contains 16-bit instructions. There are several expansion bits in the fetch packet header that indicate how the 16-bit instructions in the fetch packet are to be interpreted. For each new fetch packet, the compressor selects a window of instructions and records for each overlay which instructions may be converted to 16-bit. It then selects the overlay that packs the most instructions in the new fetch packet. Figure 6 outlines the high-level compressor algorithm.

The compressor looks for several conditions at the end of a compression iteration that might force another iteration. When the compressor finds none of these conditions, no further compression iterations are required for that section. Next, we describe one of the conditions that forces another compression iteration.

Initially, the compressor optimistically uses 16-bit branches for all forward branches whose target is in the same file and in the same code section. These 16-bit branches have smaller displacement fields than their 32-bit counterparts and so may not be capable of reaching their intended destination. At the end of a compression iteration, the compressor determines if any of these branches will not reach their target. If so, another compression iteration is required and the branch is marked indicating that a 32-bit branch must be used.

During a compression iteration, there is often a potential 16-bit instruction with no other potential 16-bit instruction immediately before or after to complete the 16-bit pair. In this case, the compressor can swap instructions *within*

```

compressor(){
  for each code section in object file {
    do {
      compress_section()
      check legality of section and note any problems
    } while (section has one or more problems)

    finalize instructions
    adjust debug, symbol, and relocation info
    write compressed section
  }
  finalize and emit the object file
}

compress_section(){

  idx = 0; // idx denotes start of "window"

  while (idx < number_of_instructions) {
    window_sz = set_window_size();

    for (each instruction in window)
      for (each overlay)
        // 128 possible combinations of expansion bits
        record if instruction can be 16-bit

    for (each overlay) {
      record how a fetch packet encodes
      // Note that the best packing may be a regular fetch packet
      <best_mode, best_insts_packed> = determine_best_overlay();
    }

    idx += best_insts_packed;
  }
}

```

Fig. 6. High-level compressor algorithm

an execute packet to try to make a pair. Since the C6x C/C++ compiler often produces execute packets with multiple instructions, the swapping of instructions within an execute packet increases the conversion rate of potential 16-bit instructions. The compressor does not swap or move instructions outside of execute packets, nor change registers of instructions in order to improve compression. The compressor will always converge on a solution, typically after five or fewer compression iterations.

5 Compiler Strategies

The C64+ architecture's novel implementation of variable length instructions provides unique opportunities for compiler-architecture collaboration. The primary benefit is that it provides flexibility to users for selecting and managing code-size and performance tradeoffs while maintaining full backward compatibility for established code bases.

In our implementation, the compressor has the responsibility for packing instructions into fetch packets. The compiler does not make the final decision whether an instruction will become a 16-bit instruction. It does, however, play the most critical role in specializing instructions so that they are likely to become 16-bit instructions. We call such instruction specialization *tailoring*.

Because instructions tailored to be 16-bit are restricted to use a subset of the register file and functional units, they can potentially degrade performance. Therefore, the compiler implements a set of command line options that allow users to control the aggressiveness of the tailoring optimizations.

In this section, we describe the various compilation strategies developed to help exploit the new instruction set features of the C64+ core.

5.1 Instruction Selection

The first strategy involves biasing the compiler to select instructions that have a high probability of becoming 16-bit instructions. This is done aggressively in regions of code that have obvious and repetitive patterns. One instance involves the construction and destruction of local stack frames. Here the compiler prefers using memory instructions with smaller or zero offsets, increasing their chance of becoming 16-bit instructions. Another instance is the usage of the CALLP instruction which handles the common code sequences around function calls.

The compiler will replace a single 32-bit instruction with two instructions that will likely compress to two 16-bit instructions. Since 16-bit instructions must be paired in the compressor, replacing a 32-bit instruction with two potential 16-bit instructions reduces the impact of 32-bit alignment restrictions, which improves the compression of the surrounding instructions. These transformations are more aggressively performed when the user is compiling for smaller code size.

During instruction selection, the compiler chooses different sequences of instructions based on the user's goal of either maximum speed or minimum code

size. When compiling for minimum code size, the compiler attempts to generate instructions that have only 16-bit formats.

The compiler assumes that any potential 16-bit instruction will ultimately become 16-bit in the compressor. That is, the compiler assumes another 16-bit instruction will be available to complete the required 16-bit instruction pair.

5.2 Register Allocation

The 16-bit instructions can access only a subset of the register file. The compiler implements register allocation using graph-coloring [11]. When compiling for minimum code size, one of the strategies employed by the compiler is to tailor the register allocation to maximize the usage of the 16-bit instructions' register file subset. This improves the likelihood that the compressor will be able to convert these instructions into their 16-bit forms.

We call this register allocation strategy *tiered register allocation*. Internally, the compiler keeps all instructions in their 32-bit format. An oracle is available that determines whether an instruction can become a 16-bit instruction given its current state.

Using tiered register allocation, the compiler limits the available registers for operands in potential 16-bit instructions to the 16-bit instructions' register file subset. If the register allocation attempt succeeds, the operands in potential 16-bit instructions are allocated registers from the 16-bit instructions' register file subset. If the register allocation attempt fails, the compiler incrementally releases registers for allocation from the rest of the register file for the operands of potential 16-bit instructions. The release of registers is done gradually, initially for operands with the longest live ranges. Should register allocation attempts continue failing, the whole register set is made available for all instruction operands thereby falling back on the compiler's traditional register allocation mechanism.

There can be a performance penalty when using tiered register allocation. When the user has directed the compiler to balance performance and code size, the compiler limits tiered register allocation to the operands of potential 16-bit instructions outside of critical loops.

When compiling to maximize performance, the compiler disables tiered register allocation and instead relies on register preferencing. Preferencing involves adding a bias to register operands of potential 16-bit instructions. Potential 16-bit instruction operands are biased to the 16-bit instructions' register file subset. During register allocation, when the cost among a set of legal registers is the same for a particular register operand, the register allocator assigns the register with the highest preference.

5.3 Instruction Scheduling

When performing instruction scheduling, the compiler is free to place instructions in the delay slot of a load instruction. After instruction scheduling, if there are no instructions in the delay slot of a load, the compressor removes the NOP

instruction used to fill the delay slot and marks the load as protected. An improvement would be to restrict the scheduler to place instructions in the delay slot of a load only when it is necessary for performance, and not simply when it is convenient. This helps maximize the number of protected loads and thus reduces code size.

When compiling to minimize code size, the compiler prevents instructions from being placed in the delay slots of call instructions, which allows the compiler to use the CALLP instruction and eliminates any potential NOP instructions required to fill delay slots.

5.4 Calling Convention Customization

The calling convention defines how registers are managed across a function call. The set of registers that must be saved by the caller function are the SOC (save on call) registers. The set of registers that are saved by the callee function are the SOE (save on entry) registers. Since SOE registers are saved and restored by the called function, the compiler attempts to allocate SOE registers to operands that have live ranges across a call. For backward compatibility, new versions of the compiler must honor the existing calling convention.

Because the 16-bit instructions' register file subset does not include SOE registers, the compiler implements *calling convention customization*. This concept is similar to veneer functions outlined by Davis et. al. [12]. The compiler identifies call sites with potential 16-bit instruction operands that have live ranges across a call. The call is then rewritten as an indirect call to a run-time support routine, which takes the address of the original call site function as an operand. The run-time support routine saves the 16-bit instructions' register file subset on the stack. Control is then transferred to the actual function that was being called at that call site. The called function returns to the run-time support routine, which restores the 16-bit instructions' register file subset and then returns to the original call site.

This technique effectively simulates changing the calling convention to include the 16-bit instructions' register file subset in the SOE registers. This greatly improves the usage of 16-bit instructions in non-leaf functions. However, there is a significant performance penalty at call sites where the calling convention has been customized. Calling convention customization is used only when compiling to aggressively minimize code size.

6 Results

A set of audio, video, voice, encryption, and control application benchmarks that are typical to the C6x were chosen to evaluate performance and code size. We ran the C6x C/C++ v6.0 compiler [13] on the applications at full optimization with each code size option. The applications were compiled with (C64+) and without (C64) instruction set extensions enabled. They were run on a cycle accurate

simulator. In order to focus only on the effect of the instruction set extensions, memory sub-system delays were not modeled.

Figure 7 shows the relative code size on each application when compiling for maximum performance (no `-ms` option) between C64 and C64+. At maximum performance, the compact instructions allow the compiler to reduce code size by an average 11.5 percent, while maintaining equivalent performance. Figure 8 shows the relative code size on each application when compiling with the `-ms2` option which tells the compiler to favor lower code size. When compiled with the `-ms2` option for C64+, the applications are 23.3 percent smaller than with `-ms2` for C64.

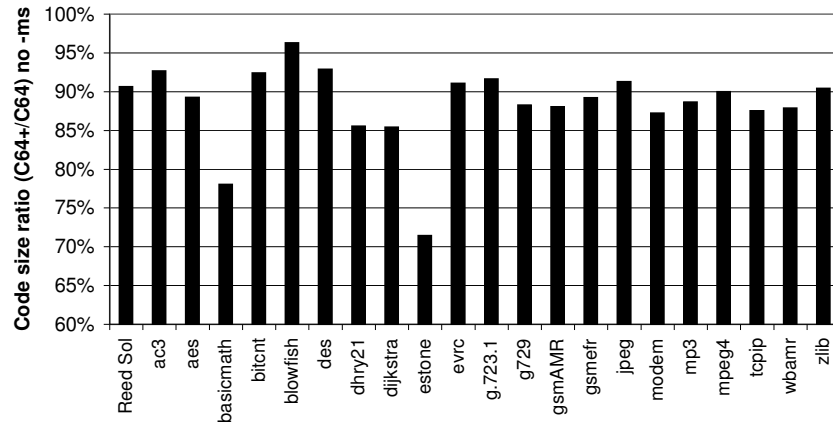


Fig. 7. Comparison of code size between C64x and C64+ at maximum performance

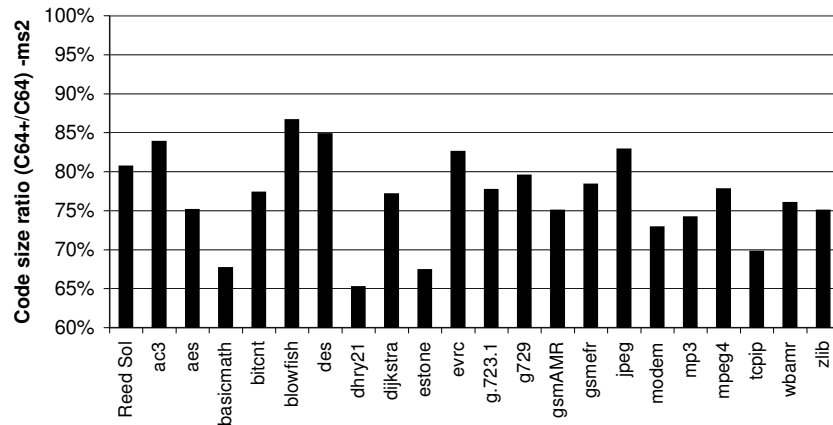


Fig. 8. Comparison of code size between C64x and C64+ at “-ms2”

In general, software pipelining increases code size. The C64+ processor implements a specialized hardware loop buffer, which, among other things, reduces the code size of software pipelined loops. The details of the loop buffer are beyond the scope of this paper, but a similar loop buffer is described by Merten and Hwu [14]. (Results shown in figures 7 and 8 do not include the effects of the loop buffer.)

Figure 9 shows the relative code size and performance differences between C64 and C64+ at all code size levels, averaged across the applications. The x-axis represents normalized bytes and the y-axis represents normalized cycles. All results are normalized to a C64 compiled for maximum performance. As the figure shows, when compiling for maximum performance, the instruction set extensions yield an average 11.5 percent code size savings with no reduction in performance (points on the middle line). The points on the leftmost line indicate the performance and code size with the addition of the software pipelined loop buffer (SPLOOP). When compiling for maximum performance, the use of the instruction set extensions and the software pipelined loop buffer result in an average 17.4 percent code size reduction with no change in performance.

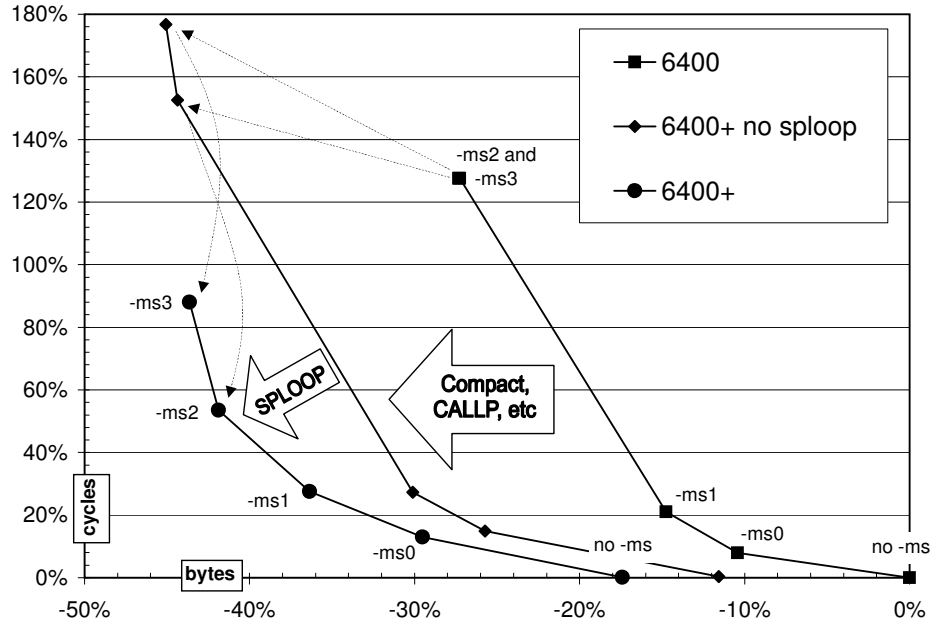


Fig. 9. Comparison of performance and code size between C64x and C64+ at various performance and code size tradeoff levels

As shown in fig. 9, the instruction set extensions create a larger code size and performance tradeoff range. When a developer's primary desire is to control

code size, this additional range can be useful in balancing performance and code size in a memory-constrained application common in embedded systems.

7 Conclusions

We have presented a hybrid approach for reducing the code size of VLIW programs using a combination of compact instruction encoding and compilation techniques. The C64+ implements a unique method for encoding variable length instructions using a fetch packet header. A post-code generation tool, the compressor, ultimately selects the size of the instructions. This approach relieves the code generator of unnecessary complexity in our constrained VLIW implementation, where 32-bit instructions cannot span a 32-bit boundary and 16-bit instructions must occur in pairs. The compiler tailors a 32-bit instruction with the potential to become 16-bit depending on the code size priority selected by the user and the location and nature of the instruction. The compiler uses various techniques, including tiered register allocation and calling convention customization to tradeoff code size and performance.

For a set of representative benchmarks, we presented results that showed an 11.5 percent to 23.3 percent code size reduction. Finally, the approach presented here is implemented in the C64+ processor and its production compiler. Future work includes improvements to the compiler to more effectively tailor code size optimizations for non-performance critical regions of a function.

References

1. Rau, B.R., Fisher, J.A.: Instruction-level parallel processing: History, overview, and perspective. *Journal of Supercomputing* 7(1-2) (1993) 9–50
2. Conte, T.M., Banerjia, S., Larin, S.Y., Menezes, K.N., Sathaye, S.W.: Instruction fetch mechanisms for VLIW architectures with compressed encodings. In: MICRO 29: Proceedings of the 29th annual ACM/IEEE International Symposium on Microarchitecture, Washington, DC, USA, IEEE Computer Society (1996) 201–211
3. Lin, C.H., Xie, Y., Wolf, W.: LZW-based code compression for VLIW embedded systems. In: DATE '04: Proceedings of the Conference on Design, Automation and Test in Europe. Volume 3., Washington, DC, USA, IEEE Computer Society (2004) 76–81
4. Ros, M., Sutton, P.: Compiler optimization and ordering effects on VLIW code compression. In: CASES '03: Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems, New York, NY, USA, ACM Press (2003) 95–103
5. Aditya, S., Mahlke, S.A., Rau, B.R.: Code size minimization and retargetable assembly for custom EPIC and VLIW instruction formats. *ACM Transactions on Design Automation of Electronic Systems* 5(4) (2000) 752–773
6. ARM Limited: ARM7TDMI (Rev. 4) Technical Reference Manual. (2001)
7. Phelan, R.: Improving ARM code density and performance. Technical report, ARM Limited (2003)
8. MIPS Technologies: MIPS32 Architecture for Programmers, Vol. IV-a: The MIPS16 Application Specific Extension to the MIPS32 Architecture. (2001)

P. Stenström et al. (Eds.): HiPEAC 2008, LNCS 4917, pp. 147–160, 2008.

© Springer-Verlag Berlin Heidelberg 2008

-
9. Texas Instruments: TMS320C64x/C64x+ DSP CPU and Instruction Set Reference Guide. (2006) Literature number spru732c.
 10. Stotzer, E., Leiss, E.: Modulo scheduling for the TMS320C6x VLIW DSP architecture. In: LCTES '99: Proceedings of the ACM SIGPLAN 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems, New York, NY, USA, ACM Press (1999) 28–34
 11. Briggs, P., Cooper, K.D., Torczon, L.: Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems* **16**(3) (1994) 428–455
 12. Davis, A.L., Humphreys, J.F., Tatge, R.E.: Maintaining code consistency among plural instruction sets via function naming convention. (1999) U.S. Patent 6,002,876.
 13. Texas Instruments: TMS320C6000 Optimizing Compiler User's Guide. (2000) Literature number spru187.
 14. Merten, M.C., Hwu, W.W.: Modulo schedule buffers. In: MICRO 34: Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture, Washington, DC, USA, IEEE Computer Society (2001) 138–149