Development of an Optimizing Compiler for a Fujitsu Fixed-Point Digital Signal Processor

Sreeranga P. Rajan and Masahiro Fujita
Fujitsu Laboratories of America
595 Lawrence Expwy, Sunnyvale CA 94086
{sree,fujita}@fla.fujitsu.com

Ashok Sudarsanam and Sharad Malik
Department of Electrical Engineering
Princeton, NJ 08544
{ashok,sharad}@ee.princeton.edu

Abstract

A common design methodology for embedded DSP systems is the integration of one or more digital signal processors (DSPs), program memory, and ASIC circuitry onto a single IC. Consequently, program memory size being limited, the criterion for optimality is that the embedded software must be very dense. We describe the development of an optimizing compiler, based on a retargetable compiler infrastructure, for the *Fujitsu Elixir*, a fixed-point DSP that is primarily used in cellular telephones. For small DSP benchmark programs (25-90 lines of C code), the average ratio of the size of compiler-generated code to the size of hand-written assembly code is 1.18. For a much larger program (more than 800 lines of C code), the ratio of the size of compiled code to the size of hand-written code is similar (1.14).

1 Introduction

A common design methodology for embedded DSP systems is the integration of one or more *digital signal processors* (DSPs), program memory, and ASIC circuitry onto a single IC. An immediate consequence of this system-on-a-chip design is the limited quantity of silicon area that is dedicated to program memory, whose purpose is to store the embedded software that executes on the DSP(s). Thus, the embedded software must be sufficiently dense so as to fit within the allocated area. Additionally, the embedded software must be written so as to meet various high-performance constraints, which may include *hard* real-time constraints.

1.1 Basic Compiler Review

One method that enables embedded software to be generated in an efficient manner is for the designer to program the application in a *high-level language* (HLL), such as C or C^{++}, and allow *compiler* technology to automatically translate the HLL program into target machine assembly code. The structure of a basic compiler is as follows: the *front-end* takes as input an HLL program and generates an *intermediate representation* (IR) of this program that is independent of the source language and target machine. The *back-end* translates the IR into target machine assembly code. In particular, the back-end performs the phases of *instruction selection*, *instruction scheduling*, and *register allocation*, which are collectively known as *code generation*.

An *optimizing compiler* features two sets of code optimization modules, in addition to the components that were described above: *machine independent* optimizations apply various optimizing transformations to the front-end-generated IR; *machine dependent* or *post-pass* optimizations apply various optimizing transformations to the generated assembly code.

Unfortunately, it is common knowledge that existing compilers for embedded DSPs are generally unable to generate assembly code that is sufficiently dense. It is typical for naive DSP compilers to generate assembly code whose size is more than 5 times greater than the size of the corresponding hand-written assembly code. There are two reasons for this: **first**, most existing embedded DSP compilers make use of traditional machine-independent optimization techniques, in which the primary metric is performance, rather than code density. In some cases such as loop unrolling these two metrics are not closely associated. **Second**, current embedded DSP compilers fail to provide adequate support for the specialized architectural features of DSPs via machine-dependent code optimizations. These features not only allow for the fast execution of common DSP operations, but also allow for the generation of dense assembly code that specifies these operations.

In order to guarantee that all code density and performance requirements are safely met, system designers typically hand-program the embedded software in assembly, which is a very time-consuming, tedious, and error-prone task. In order to increase productivity, compilers must be developed that are capable of generating high-quality code for embedded DSPs.

In this paper, we discuss how we have developed, based on a retargetable compiler infrastructure, a high-quality compiler for the *Fujitsu Elixir*, a fixed-point DSP that is primarily used in cellular telephones. Included in this paper is a description of the various specialized features of the Elixir architecture that allow for the generation of dense and efficient assembly code. Also included is a description of the machine-dependent code optimization algorithms that we have incorporated into our compiler that provide support for each of these features. Experimental results demonstrate that for relatively small benchmark programs (25-90 lines of C code), the average ratio of the size of the compiler-generated code to the size of reference hand-written assembly code is 1.18. For a much larger program (around 800 lines of C code), the ratio of the size of the compiler-generated code to the size of hand-written assembly code is 1.14.

This paper is organized as follows: Section 2 describes the *SPAM* compiler infrastructure, a retargetable compiler infrastructure from which the Elixir optimizing compiler has been constructed; Section 3 describes the Fujitsu Elixir architecture; Section 4 describes the various code optimizations that have been incorporated

into the Elixir compiler; Section 5 provides quantitative measurements pertaining to the quality of code that is generated by our compiler, as well as the amount of effort that was involved in constructing this compiler; finally, we present our conclusions in Section 6.

2 The SPAM Compiler Infrastructure

In order to build the Elixir optimizing compiler, we have utilized the *SPAM* compiler infrastructure, which is a retargetable compiler infrastructure for embedded fixed-point DSPs. Most efforts in retargetable compilation for embedded DSPs (see [6]) have focused on a level of retargetability known as *user retargetability*. In a user-retargetable compilation environment, a description of the target processor is provided to a *compiler-compiler*, which first determines those built-in code optimization algorithms that are applicable to this processor, then automatically constructs an optimizing compiler for it. User retargetability has been successfully incorporated into the code generation phase of compilation: the *Twig* and *Iburg* [3] code generator-generators take as input a description of the target machine *instruction set architecture* (ISA), then automatically construct code generators that perform, in linear time, optimal instruction selection for expression trees.

Unfortunately, research in user-retargetable DSP compilation has been unable to achieve high code quality in parallel with automation – user retargetability is currently limited to instruction selection plus a very small number of machine-dependent optimizations. The main reason for this is that there currently does not exist a machine-description format that enables a compiler-compiler to automatically infer the set of all post-pass code optimizations that are applicable to the target DSP architecture – these optimizations must be performed in order for high-quality code to be generated.

In contrast to previous works, the SPAM compiler infrastructure enforces a level of retargetability known as *developer retargetability*, where a developer is defined to be a person who has had experience in compiler design. Built into a developer-retargetable compilation framework is a *library* of *parameterized algorithms* that perform code generation and machine-dependent code optimization. The developer may retarget such a framework to a particular DSP using the following methodology:

- first, the developer infers the set of all post-pass code optimizations that are applicable to the target machine by thoroughly studying the architectural characteristics of this machine.
- second, for each applicable post-pass optimization *O*, the developer determines whether or not a parameterized algorithm for *O* exists in the compilation framework. If such an algorithm exists, then the developer simply provides it with the appropriate machine-specific parameters (it should be noted that the specification of machine-specific parameters may involve the development of a sizable amount of source code); otherwise, the developer *implements* a parameterized algorithm for *O*, adds this algorithm to the built-in library, and then provides the algorithm with the appropriate parameters.
- finally, the developer invokes all applicable code generation and optimization algorithms in the most effective order.

The effectiveness of developer-retargetable compilation is measured by the amount of *code reuse* that occurs during the retargetting process. In particular, a developer-retargetable compilation framework is said to exhibit sufficient retargetability if the developer is able to make use of a significant quantity of source code that already exists in the framework.

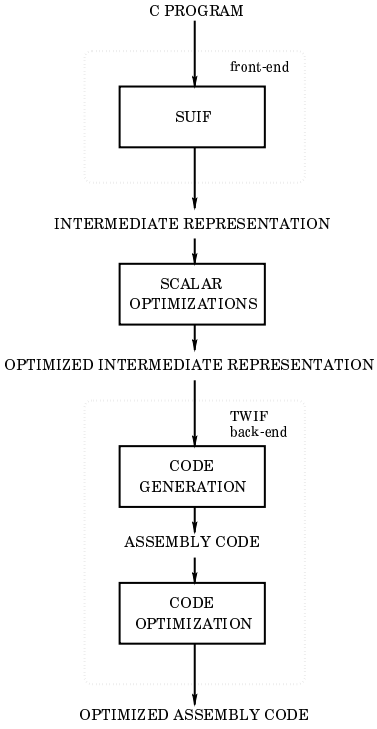

Figure 1: Structure of the SPAM compiler infrastructure

The structure of the SPAM compiler infrastructure, which is publicly available on the internet for research or development purposes, is shown in Figure 1 and outlined below:

The *Stanford University Intermediate Format* (SUIF) compiler [6] implements the front-end of the SPAM compiler. SUIF takes as input a source program written in C, and generates an unoptimized IR.

A series of machine-independent *scalar optimizations* apply various code density-improving transformations to this IR.

The back-end of SPAM, which is known as *TWIF*, consists of two components. The first component is a library of *data structures* that encapsulate the various source program intermediate representations (e.g. *calling graph*, *control-flow graphs*, *expression DAGs* and *trees*, etc.). The second component is a library of *algorithms* that perform code generation and machine-dependent code optimization by analyzing and manipulating these data structures. These algorithms currently support embedded fixed-point DSPs with limited parallelism. However, we are in the process of augmenting the SPAM infrastructure with new algorithms that will enable it to provide support for other classes of embedded DSPs, such as floating-point DSPs and VLIW DSPs.

3 Fujitsu Elixir Architecture

The Fujitsu Elixir, whose architecture is shown in Figure 2, is a proprietary DSP that is primarily used in cellular telephones. It is composed of an efficient 16-bit digital processor core, program and data memories, and supporting peripherals. In this section, we describe the various architectural features of the Elixir. We begin by describing the register set.

3.1 Registers

The Elixir architecture features the following sets of registers:

Two 40-bit accumulators: *CX* and *DX*.

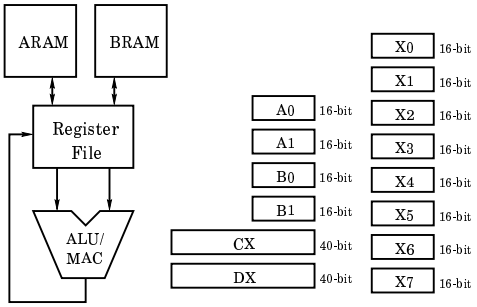

Figure 2: Fujitsu Elixir architectural overview

Four 16-bit general-purpose data registers: *A0*, *A1*, *B0*, and *B1*.

Eight 16-bit address registers, which are used to indirectly address data memory: *X0*, *X1*, *X2*, *X3*, *X4*, *X5*, *X6*, and *X7*.

3.2 Data memory banks

The Elixir architecture features two independent 4K-word x 16-bit data memory banks, *ARAM* and *BRAM*, which enable two data memory accesses to occur in parallel provided that the referenced variables have been allocated to different banks, and the registers involved have been allocated appropriately.

3.3 Addressing modes

The Elixir ISA supports the following three addressing modes: *Absolute addressing*, *Register-indirect addressing*, and *Modulo addressing*. In the Register-indirect addressing mode, associated with the address register file in the Elixir architecture is a specialized *auto-increment* update mode, which enables an address register to be post-incremented or post-decremented by the constant *one* at no extra cost after a register-indirect memory access. The auto-increment update mode allows for efficient sequential accessing of data memory.

3.4 Arithmetic/Logic Unit (ALU)

The Elixir ALU features hardware that supports single-cycle 16x16 multiplication and multiply-accumulation (MAC). The ALU also efficiently performs a rich set of logical and arithmetic operations. A significant amount of *instruction-level parallelism* (ILP) exists between the ALU and data memory banks. In particular, certain ALU operations may be performed concurrently with data memory access operations.

3.5 Zero-overhead looping hardware

The Elixir architecture features *zero-overhead looping* hardware [6], which is specialized hardware that supports the efficient execution of loops and also allows for the generation of dense assembly code that specifies looping constructs. Zero-overhead looping hardware consists of specialized logic that repeatedly executes one or more instructions without incurring the loop-index-variable-update and conditional-branching overhead generally associated with software implemented loops.

4 Elixir Compiler Optimization Algorithms

In this section, we discuss the various code optimization algorithms that have been incorporated into the Elixir optimizing compiler. Each of these algorithms attempts to reduce overall code size by providing support for one of the specialized architectural features described in the previous section. We begin by discussing the *pointer variable* optimization.

4.1 Pointer variable optimization

This optimization exploits the auto-increment capability of the address registers in the Elixir architecture by translating each pointer variable access with post-increment in the source program into a register-indirect memory access with auto-increment in the assembly program. It is assumed by this optimization that an unlimited number of address registers are available – hence, each use of register indirection involves a *symbolic*, rather than a *physical*, address register.

4.2 Optimization of loops

This optimization exploits the specialized zero-overhead looping hardware in the Elixir architecture by transforming looping constructs such that they execute on this hardware. Specifically, given a looping construct in which the iteration count is statically known and the loop index variable is used as a counter only, this optimization inserts an appropriate loop hardware-controlling instruction prior to the loop body (this instruction is either a *REP* or *DO*, depending on whether the body is comprised of a single instruction or multiple instructions, respectively).

4.3 Symbolic register allocation

This phase allocates the *local* program variables to registers, assuming that an unlimited number of registers are available. In particular, each reference in the source program of a non-pointer (pointer) local variable is replaced by a *symbolic* data (address) register reference, as opposed to a direct data memory reference. It is the goal of the subsequent register *assignment* phase to map each symbolic data (address) register onto a physical data (address) register, while keeping to a minimum the number of symbolic registers that need to be stored in memory.

4.4 Local code compaction

The heuristic for performing code compaction, employed in the Elixir compiler, is the *list scheduling* local compaction algorithm [4], which performs code compaction within *basic block* boundaries. List scheduling schedules two memory accesses into a long instruction word provided that no data dependencies are violated. Additionally, an ALU operation *O* is scheduled with data memory accesses provided that no data dependencies are violated and it is permissible for *O* to execute in parallel with data memory accesses. The Elixir compiler performs compaction *before* variables have been allocated to memory banks and symbolic registers have been assigned to physical registers. If the compiler were to perform compaction after memory bank allocation and register assignment, then the quality of the compacted code could suffer (due to the stringent memory bank allocation and register assignment constraints that are imposed on compacted instructions by the Elixir ISA).

4.5 Memory bank allocation

In order for an instruction containing two memory accesses to be legally specified, the referenced variables must be allocated to different memory banks, and the symbolic registers involved must be appropriately assigned to physical registers. Otherwise, the parallel memory accesses must be decomposed into two sequential accesses. The goal of the memory bank allocation phase is to intelligently allocate variables to memory banks ARAM and BRAM such

that a minimum number of parallel memory accesses are illegally specified.

4.6 Symbolic data register assignment

The objective of the symbolic data register assignment phase is to assign each symbolic data register to a physical data register, while keeping to a minimum the number of symbolic registers that need to be *spilled* to (i.e. stored in) data memory. The Elixir compiler makes use of the graph coloring heuristic that was proposed by *Briggs et al.* [2].

4.7 Modulo addressing optimization

The modulo addressing code optimization attempts to exploit the specialized modulo addressing mode in the Elixir ISA, which allows for the implementation of circular data buffers. In particular, this optimization searches the code for non-arithmetic uses of the modulo operator, replaces each such use by a modulo-addressed memory access, and inserts at the appropriate locations assembly instructions that define a circular buffer.

4.8 Array index allocation

Given a sequence of array accesses within a loop, the array index allocation optimization [1] utilizes the auto-increment capability of the address registers in order to efficiently walk through the elements of the arrays. This algorithm is based on finding a *minimum-cardinality disjoint path covering* of an acyclic *indexing graph*, which conveys the distance between each pair of array indices in the loop. Additionally, this algorithm assumes the existence of an unlimited number of address registers.

4.9 Offset assignment optimization

Given a compacted instruction containing two data memory accesses, the compiler must employ register-indirect addressing for each access. Given an instruction containing a single memory access, the compiler may employ either absolute or indirect addressing. However, since each use of absolute addressing in the Elixir ISA incurs a one-word penalty (an additional instruction word is required to encode the absolute address), the compiler references all local variables using the register-indirect addressing mode.

Now, in order for a memory location to be indirectly addressed, there must exist an address register that points to this location. One method of ensuring that such an address register exists is to perform *address arithmetic* on an address register – in the Elixir ISA, address arithmetic is implemented by either loading an address register with the desired address, or by adding an offset to an address register. However, note that the auto-increment update mode allows for efficient sequential accessing of memory. Consequently, instead of generating expensive address arithmetic instructions prior to each indirect memory access, the compiler could intelligently *order* the local variables within the two data memory banks and allow address arithmetic to be subsumed into cheaper auto-increment arithmetic. Determining such an intelligent variable ordering is the goal of the *Offset Assignment* (OA) problem [5].

The OA algorithm employed by the Elixir compiler operates on an intraprocedural basis, so it is not able to determine the optimal memory location for each global variable. Hence, the compiler references all global variables using the absolute addressing mode. Furthermore, the algorithm assumes the existence of an unlimited number of address registers.

Benchmark	Size (Hand)	Size (Compiled)	Code Size Ratio
hup	17	20	1.18
yhaten	28	36	1.29
kncal	36	38	1.06
dt_pow	148	176	1.19

Table 1: Comparison of Elixir compiled code with reference code (small benchmarks)

4.10 Symbolic address register assignment

This phase is analogous to the symbolic data register assignment phase, except that the graph coloring process is applied to the symbolic address registers in the assembly code.

5 Experimental Results

In this section, we present two sets of experimental results. The first set of results pertains to the quality of code that is generated by the Elixir optimizing compiler. The second set quantitatively describes the amount of developer effort that was involved in constructing this compiler.

5.1 Quality of the Elixir Optimizing Compiler

It is not uncommon for naive DSP compilers to generate assembly code whose size is more than *5* times greater than the size of the corresponding hand-written assembly code. In order to determine the quality of code generated by the SPAM-based Elixir compiler, we obtained two sets of DSP benchmark programs. The first set of benchmarks is comprised of four hand-written assembly-coded programs that form part of a large Fujitsu cellular telephone application. Associated with each of these programs is an algorithmic description of the program, which we have translated into semantically-equivalent C source code – the size of this source code ranges from 25 to 90 lines (excluding comments). The second set of benchmarks is comprised of the large *G721* and *G729* speech encoding/decoding applications. The G721 benchmark consists of approximately 800 lines of C source code, while the G729 benchmark consists of approximately 10,000 lines of C source code.

Table 1 presents results pertaining to the quality of code that is generated by the Elixir compiler for the first set of benchmarks. Based on these results, it can be seen that the average ratio of the size of the compiler-generated code to the size of the corresponding hand-written code is 1.18, which is significantly less than the code size ratio that is typically associated with naive DSP compilers. A close analysis of the compiler-generated and hand-written code revealed that there were two primary reasons for the compiled code overhead. First, the Elixir compiler did not perform code compaction within loops as efficiently as the assembly programmer. Second, the compiled code contained more address arithmetic instructions than the hand-written code.

In Table 2, experimental results are presented that pertain to the quality of code that is generated by the Elixir compiler for the G721 and G729 benchmarks. Table 2 demonstrates that the Elixir compiler generated good-quality code for the G721 program. In particular, the ratio of the size of the G721 compiled code to the size of the corresponding hand-written code is 1.14, which is also much less than the code size ratio typically associated with naive DSP compilers. Although hand-written code is not yet available for G729, we estimate that the size of this code (in instruction words) will be approximately two times the size of the corresponding C source code. Thus, we predict that the ratio of the size of compiler-generated code to the size of hand-written code for the decoder and

Bench mark	Code Sz C / Hand	Code Sz Compiled	Code Sz Ratio	Compile Time
G721	800 / 2,300	2,615	1.14	1 min.
G729 dec	4,000 / -	11,915	-	25 min.
G729 cod	6,000 / -	24,562	-	1 hr.

Table 2: Comparison of Elixir compiled code with reference code (large benchmarks)

DSP	Code Size (lines of code)				Reuse Rate
	Total	Reused	Algo	Struc	
C25	24,100	14,800	5,600	9,200	61.4%
56K	27,400	17,400	8,200	9,200	63.5%
Elixir	29,150	17,400	8,200	9,200	59.7%

Table 3: Code reuse measurements

coder components of G729 will be approximately 1.49 and 2.05, respectively.

As shown in Table 2, it should be pointed out that on a *200 MHz Sun Microsystems SPARC20* with *128MB* of RAM compilation of G721 required approximately 1 minute; and compilation of the decoder and coder components of G729 required approximately 25 minutes and 1 hour respectively.

5.2 Quantitative Measurements of Code Reuse

As part of our research in developer-retargetable DSP compilation, we have used the SPAM compiler infrastructure to build optimizing compilers for two other fixed-point DSPs besides the Fujitsu Elixir, namely the *Texas Instruments TMS320C25*, and *Motorola DSP56000* [6] (which will subsequently be referred to as the *C25* and *56K*, respectively). The effectiveness of developer-retargetable compilation is measured by the amount of *code reuse* that occurs during the retargetting process. With respect to the SPAM infrastructure, we define reusable source code to encompass the built-in optimization algorithms and data structures in the TWIF back-end. Furthermore, we characterize the use of built-in source code in a particular compiler as an instance of code reuse if and only if this source code is utilized in *at least one other compiler*.

Table 3 presents measurements that quantitatively describe the amount of code reuse that occurred during the construction of the three optimizing DSP compilers. This table is organized as follows: the three DSPs are listed in the first column; the second column specifies the total size (in lines of code) of the source code that implements the corresponding compiler – these numbers only include source code that forms part of TWIF, as well as any machine-specific source code that has been written by the developer (e.g. for purposes of specifying machine-specific parameters); the third column specifies the total size (in lines of code) of all source code in the compiler that was reused (i.e. code that was employed in at least one other compiler); the next two columns specify the total size (in lines of code) of the reused code that represents built-in algorithms and data structures, respectively; the final column specifies the percentage of source code in each compiler that was reused.

Table 3 demonstrates that a significant amount of code reuse occurred during the retargetting of the SPAM infrastructure to the three fixed-point DSPs – the average percentage of source code in each compiler that is utilized in at least one other compiler is 61.5%. This high rate of code reuse may be attributed to two factors. First, the entire data structures library in TWIF is utilized in each compiler. Second, many of the built-in optimization algorithms in TWIF are utilized across multiple compilers.

Another metric that quantitatively describes the amount of developer effort involved during the compiler-construction process is overall development time – on average, approximately 100 hours were spent developing each compiler. However, we believe that overall development time is an inappropriate metric for developer effort for the following reason: since most developers will initially have little familiarity with the SPAM infrastructure, these developers will most likely initially require considerably more than 100 hours to develop a compiler. We believe that overall code reuse is a much better metric for developer effort, since it is independent of the developer's familiarity with the SPAM infrastructure.

6 Conclusions

We have described the development of a high-quality compiler for the *Elixir* DSP. This compiler is built on top of the SPAM compiler infrastructure, which is a developer-retargetable compilation framework for embedded fixed-point DSPs. Most work in retargetable DSP compilation has focused on the automatic construction of optimizing compilers from a given target machine description; however, due to the extreme difficulty in automatically inferring the set of all post-pass optimizations that are applicable to the target machine, these works suffer in terms of the quality of code that is generated. In contrast, the developer-retargetable compilation methodology enables a wide range of post-pass optimizations to be supported in a retargetable manner, although some developer effort is generally required.

Experimental results demonstrate that our compiler generates good-quality code for a variety of DSP benchmark programs. Specifically, for a group of small programs, the average ratio of the size of compiled code to the size of hand-written code is 1.18; for the larger program G721, the average ratio of the size of compiled code to the size of hand-written code is 1.14; and although hand-written code for the G729 application is currently unavailable, we expect the ratio of the size of compiled code to the size of hand-written code for the decoder and coder components to be approximately 1.49 and 2.05, respectively.

References

[1] G. Araujo, A. Sudarsanam, and S. Malik. Instruction Set Design and Optimization for Address Computation in DSP Architectures. In *Proceedings of 9^{th} International Symposium on System Synthesis*, pages 102–107, 1996.

[2] P. Briggs, K.D. Cooper, and L. Torczon. Improvements to Graph Coloring Register Allocation. ACM *Transactions on Programming Languages and Systems*, 16(3):428–455, May 1994.

[3] C.W. Fraser, D.R. Hanson, and T.A. Proebsting. Engineering a Simple, Efficient Code-Generator Generator. *ACM Letters of Programming Languages and Systems*, 1(3):213–226, September 1992.

[4] D. Landskov, S. Davidson, B.D. Shriver, and P.W. Mallett. Local Microcode Compaction Techniques. *ACM Computing Surveys*, 12(3):261–294, September 1980.

[5] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, and A. Wang. Storage Assignment to Decrease Code Size. In *ACM Transactions on Programming Languages and Systems*, volume 18, pages 235–253, May 1996.

[6] Ashok Sudarsanam. *Code Optimization Libraries for Retargetable Compilation for Embedded Digital Signal Processors*. PhD thesis, Princeton University, November 1998.