Address Calculation for Retargetable Compilation and Exploration of Instruction-Set Architectures

Clifford Liem[1,2], Pierre Paulin[2], Ahmed Jerraya[1]

(1) TIMA Laboratory, Inst. Nat. Polytech. de Grenoble (INPG)
46, ave Félix Viallet, 38031 Grenoble, France
liem@verdon.imag.fr jerraya@verdon.imag.fr

(2) Central R&D, SGS-Thomson Microelectronics (ST)
850, rue Jean Monnet, 38921 Crolles, France
pierre.paulin@st.com

Abstract

The advent of parallel executing Address Calculation Units (ACUs) in Digital Signal Processor (DSP) and Application Specific Instruction-Set Processor (ASIP) architectures has made a strong impact on an application's ability to efficiently access memories. Unfortunately, successful compiler techniques which map high-level language data constructs to the addressing units of the architecture have lagged far behind. Since access to data is often the most demanding task in DSP, this mapping can be the most crucial function of the compiler. This paper introduces a new retargetable approach and prototype tool for the analysis of array references and traversals for efficient use of ACUs. The ArrSyn utility is designed to be used either as an enhancement to an existing dedicated compiler or as an aid for architecture exploration.

1 Introduction

As data intensive algorithms push for higher speeds on Digital Signal Processing (DSP) architectures, access to data memories become the limiting factor. In response to this, designers have conceived the Address Calculation Unit (ACU) (sometimes termed Address Generation Unit (AGU), Address Arithmetic Unit (AAU), or Memory Management Unit (MMU)), an arithmetic unit which works in parallel to the main Data Calculation Unit (DCU). The ACU works solely on address generation to ensure efficient retrieval and storage of data that is calculated on the DCU. In most cases, the ACU works in a post-increment/decrement fashion to ensure high speed. Pre-increment/decrement addressing is rare because this would require at least two operations to occur in the same instruction cycle, namely the address calculation, then the memory access.

Post-increment/decrement address units are present on countless general-purpose DSPs and cores, for example the SGS-Thomson D950 core [1] (shown in Figure 1), the Motorola 56000 series, the Texas Instruments TMS320C25 [2], and the Lode DSP Engine [3], to name a few. They are also common in Application Specific Instruction-Set Processors (ASIPs) used in applications such as MPEG audio [4][5], Dolby decoding [6], and DSP for telecommunications [7].

Although ACUs have existed for some time, the compiler techniques for mapping high-level language constructs onto the register structures are immature. This is immediately reflected in the poor performance of today's DSP compilers [9]. The problem of mapping array structures onto these calculation units manifests itself in two ways:

1. difficulties of dealing with special registers and connections.
2. difficulties in treating the disjunction in dependency between the use of addresses and the calculation of new addresses, inherent in the post-modify operation of the unit.

The lack of adequate compilation techniques has forced designers to write assembly-level programs, which has disadvantages in maintenance and design evolution to new processors.

Previous experience [5] has shown that lowering array-based C code to pointer-based code can significantly improve performance. This paper addresses this type of transformation.

2 Traditional Compiler Approaches

2.1 Address Pre-calculation

A straight-forward method of calculating addresses for arrays is *on-the-fly* generation. For a simple array reference, this involves the addition of a base address with an induction variable (assuming the data size is 1; otherwise, a multiplication by a constant is needed) as depicted in the example of Figure 2. The shortcoming with this approach is that the value of the address must be calculated before the reference because the operation is data dependent. Within the context of loop bodies, array calculations of this sort can be an enormous performance penalty.

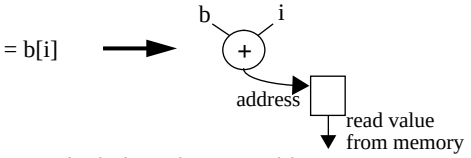

Figure 2. Pre-calculation of Array Addresses

An improvement to the straight-forward approach is loop pipelining [8], where addresses for iteration *i* are calculated at iteration *i-1*. This can be extended depth-wise for the number of operations of the induction variables. However, the incremented *values* of the induction variables themselves must still be calculated for each iteration of the loop (usually at the end of the loop). Moreover, this type of pipeline does not offer a natural mapping to a DSP type of ACU (Figure 1), especially within the context of the address register connections.

2.2 Address Post-calculation

A second approach to this problem involves reducing an array reference to an address (pointer) reference and incrementing/decrementing the resulting address for the next reference of the array. This optimization has a two-fold advantage. The address calculation can be done in parallel to any principal operations and there is no more need for an induction variable or induction variable calculation (assuming it is not needed for other purposes). An example is shown in Figure 3, where for instance, the next use of the pointer *bp* is one position higher than the current position of bp.

This approach maps most naturally to the ACU post-incrementing structure described earlier. The approach requires, as is also true for loop pipelining, a careful semantic analysis of the subscript dependencies through the various control constructs of the source program.

33rd Design Automation Conference ®

DAC 96 - 06/96 Las Vegas, NV, USA

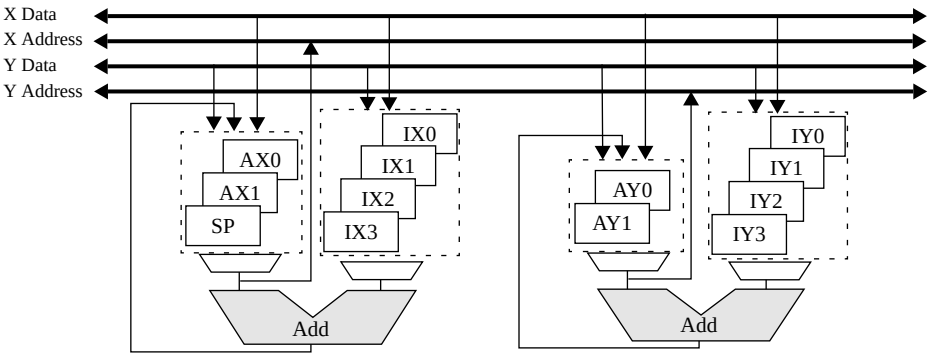

Figure 1. Linear Post-Indexing ACU of the SGS-Thomson D950 Core [1]

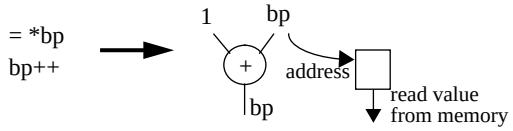

Figure 3. Post-calculation of Array Addresses

Previous work related to this area is found in [10][11]. Methods include loop manipulations and transformations where the loops are strongly restricted to certain behaviours. These methods do not directly take into account the target architecture.

3 Array Transformation Approach

The following approach is aimed towards array to pointer reducing type of optimizations described in Section 2.2, with the following goals:

- Retargetability: ability to reconfigure for other architectures.
- Designer Feedback: a maximum of information useful for architecture exploration.
- Efficient Analysis: a minimum of complex semantic analysis.
- Facility to Integrate: ease of integrating into existing compiler systems.

3.1 Overall Flow

The proposed array analysis flow is depicted in Figure 4. From the user's viewpoint, only the shaded boxes are visible. A C source containing array references and a specification file indicating the addressing resources in the target architecture are provided. The system then transforms the array references of the source to pointer references and appropriate increments and decrements of those pointers optimized for the provided ACU specification. In addition, statistics are provided to the user during compilation and as comments embedded in the target (C source with addressing). These statistics include basic block frequencies, array reference frequencies, and the number of pointers created. For the created pointers (of which the number may not correspond directly to the number of arrays), the system also provides the reference frequencies and the frequencies of increment/decrement operations.

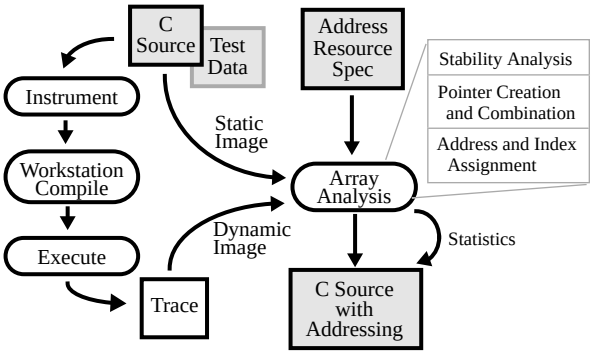

Figure 4. Array Analysis Flow

The choice of the C language as the target provides the following benefits:

- the target can be compiled and verified against the behaviour of the source.
- the target can be fed directly to a processor-specific compiler.
- the semantics are easily understood by a human reader.

However, a drawback of generating C is that fine-tuning for parallelism is not possible. Parallelization (compaction) is left for the back-end architecture compiler.

The central analysis block uses both a static and dynamic (run-time) image of the source algorithm. The advantages of using static and dynamic information versus only static information are:

- the ability to determine non-obvious linear relationships.
- the ability to allow calculated loop variables and array references, provided the calculations do not come from input data.
- the simplicity and speed of the analysis.
- the availability of relative frequencies of basic blocks, which provide a realistic cost function of the insertion of operations.

The dynamic image of the source is created through instrumentation of the original source, compilation, and execution on the workstation. Details are provided in Section 3.3. As a result of this methodology, the following items are required:

- source must be compilable and executable on the workstation.
- data must be provided that exercises all the basic blocks to be analyzed.
- execution on the workstation must have reasonable run-time.

In our experience, these items are common in an embedded system development methodology, where firmware is simulated on a desk-top platform before being used in the field. This differs in nature from a general computing environment.

3.2 Address Resource Specification

An example resource specification is shown in the left side of Figure 5. The specification includes two main parts: a declaration of resources (address and index registers) and the operations that can be performed on these resources. This specification is represented internally as a structural connection of registers, adders, and constants. This behavioural representation describes naturally the full operation of the unit and it is used for allocation and assignment, where pointers and increments can be bound to registers and constants.

3.3 Instrumentation and Tracing

Instrumentation is defined as the transformation of the origi-

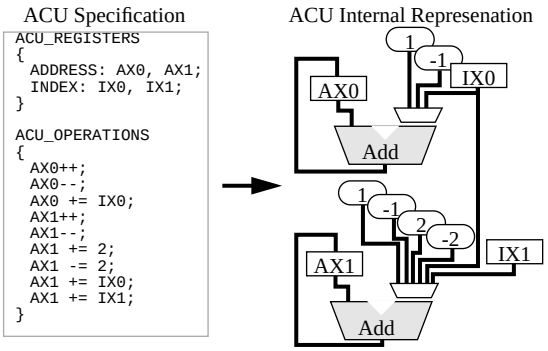

Figure 5. ACU Specification and Representation

nal source code to a duplicate plus the addition of tags. Tagging is formulated as a lexical and semantic analysis of the source program for the annotation of output statements that indicate run-time information. The tags include: function entries, function exits, loop entries, loop begins, loop-exits, array references including induction variables and run-time values of induction variables. The run-time values of the induction variables is the key component which allows analysis of the array traversals.

Execution of the instrumented code produces a trace that is consumed by the main analysis block of the system. In this manner, the array access patterns can be determined quickly and with a minimum of semantic analysis. Tracing has also been used in other contexts to improve run-time performance [12].

3.4 Stability Analysis

Given a reference to an array at a static position in the source program, stability analysis determines if this reference is visited in a linear fashion within a loop hiearchy throughout the execution of a program. If so, it may be replaced by a pointer and an increment/decrement set. The analysis makes use of the dynamic trace of the program which quickly evaluates the characteristics of the array reference run-time progression. Array references which are stable within a set of loops may be replaced by a pointer reference and a set of increment/decrement operations or combined with another pointer reference.

3.5 Pointer Creation and Combination

Pointer creation and combination is the allocation phase of the analysis. The goal is to produce an appropriate number of pointers which match the capabilities of the address calculation unit. The approach begins by creating a pointer for each stable static array reference of the source program. From this starting point, static pointers are combined (i.e. the references use the same pointer) until a reasonable number exist for the architecture at hand. The combination strategy uses the following rules:

- pointers created for array references with exactly the same signature within the same nest of loops may be combined.
- pointers with non-overlapping lifetimes may be combined.
- pointers referencing the same array at different relative positions within the same nest of loops may be combined.

As these transformations have various effects on the resulting code, rules are executed with the following objectives in mind:

- reduce the number of pointers to an amount equal or below the number of available address registers.
- minimize the frequency of inserted increments/decrements of pointers.
- minimize the number of different valued increments/decrements for each pointer.

3.6 Address and Index Register Assignment

Following their creation is the assignment of pointers and increments to address registers and index registers or constants. Lifetime analysis has already been done in the combination stage; therefore, the problem is to find the best one-to-one matching of pointers to address registers and their respective increments to constants or index registers. If the number of pointers that exist after combining is less than the number of address registers, a direct mapping is usually possible. If the number of pointers exceeds the number of address registers, then some pointers will be assigned to memory and must be loaded/stored into a free address register. Pointers which are referenced more frequently will reside directly in registers, while those which are referenced infrequently may reside in memory and be retrieved for usage.

For direct mappings, assignment proceeds in two steps:

1. An estimated cost of all possible assignments of pointers to address registers is determined.
2. Pointers are assigned to address registers using a heuristic based on the estimated costs in an attempt to reduce the overall real cost.

The real cost is sometimes dependent on the order of assignment, which explains why step 1 is an estimate.

The cost of an assignment is based on the frequency of increment/decrement instructions in the final code. This is best explained through an example (Figure 6). The assignment cost function is defined as the frequency of ACU operations that will be executed in the final code, making use of the dynamic information provided by tracing. Two example assignments are shown in Figure 6. Pointer *bp* is incremented 12 times by +1 and decremented 4 times by -1. The assignment to the address register *AX0* and constant indices of +1 and -1 gives an assignment cost of 17, since the operations *AX0++* will be executed 12 times, *AX0--* will be executed 4 times and an initialization *AX0 = &b[n]* will be executed one time (the value of *n* and the number of initializations is dependent on the context in the program). This cost function does not correspond directly to the number of whole instructions that will be executed in the final code, since these instructions are likely to be compacted and executed in parallel with other operations; however, the function is a good reflection of the trade-off between different assignments.

For index registers, an assignment cost is defined in the same manner. In Figure 6, pointer *xp* is incremented 25 times by +4 and 14 times by +13. The assignment to the address register *AX1*, the constant index of +2 and the index register *IX1* gives an assignment cost of 66, since the operation *AX1 += 2* will be executed 50 times, *AX1 += IX1* will be executed 14 times, and the initializations *AX1 = &x[n]*, *IX1 = 13* will each occur once. Note that the assignment cost for index registers is a value dependent on the order of pointer assignment. In this example, if *IX0* were chosen for the assignment to +13, then an earlier as-

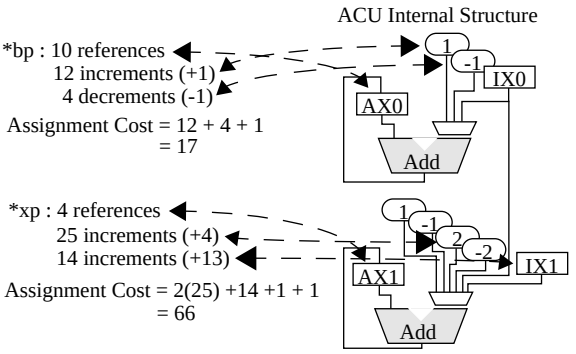

Figure 6. Address and Index Assignment Cost Function

signment of another pointer and increment to *AX0* and *IX0* in the same region of code may cause *IX0* to be reloaded with another increment value. This would be an expensive instruction especially inside loops.

This estimated cost guides the assignment heuristic since it can determine the best places for potential savings. As well, during assignment for index registers, the algorithm attempts to share common index values in index registers wherever possible to reduce the number of initializations.

4 Results

The array analysis flow has been implemented in a prototype called *ArrSyn* and tested on a set of benchmark examples together with a dedicated compiler for an MPEG VLIW processor [5]. These examples include various DSP functions, some specific to MPEG Audio, others for standard DSP tasks such as interpolation and noise addition. Table 1 shows code size results, while Table 2 shows performance results. Speed is calculated as one cycle per instruction multiplied by the frequency a basic block is executed. These frequencies are luckily found in the target C source produced from tracing. This is a first order estimate of the execution time, which does not take into account conditional paths.

Example	Assembly Lines C compiler	Assembly Lines ArrSyn + C compiler	% Improvement in Code Size
simple_loop	31	21	32 %
median	83	56	33 %
interpolate	72	49	32 %
addnoise	59	48	19 %
alloc	80	75	6 %
Total	**325**	**249**	**23 %**

Table 1. Code Size comparison of C compiler with and without the ArrSyn utility.

Example	Number of cycles C compiler	Number of cycles ArrSyn + C compiler	% Improvement in Speed
simple_loop	103	69	33%
median	715	350	51%
interpolate	1017	499	61%
addnoise	1219	802	34%
alloc	10309	8526	17%
Average			**39%**

Table 2. Performance comparison of C compiler with and without the ArrSyn utility.

Table 1 and Table 2 show that a significant improvement in both the code size (23% reduction) and the performance (39% speed-up) results from the ArrSyn transformation to the C code for better utilization of the address calculation unit. Note that in these examples, the hardware loops which are available on the core have not as of yet been utilized. This will lead to an additional improvement due to both the replacement of expensive branch instructions for hardware loop instructions as well as the removal of looping variables.

5 Conclusion

The contribution of this paper has been to introduce a new approach to transforming C code to make better use of address calculation units on DSP and ASIP architectures. We believe this to be one of the key improvements needed for today's DSP compilers [9]. The approach has been implemented in a prototype called *ArrSyn* and tested on benchmark examples to enhance a dedicated DSP compiler. Results show a significant improvement in code size (23% reduction) and execution speed (39% speedup) when comparing array-based code to pointer-based code transformed using the ArrSyn utility.

In addition to enhancement of the existing algorithms, future work includes extensions to hardware features such as the commonly found circular buffer (modulo addressing) modes and bit-reversal addressing modes. Another hardware consideration is the ability to hold a constant increment value in the instruction register as opposed to the value being hard-wired or in an increment register. On the C-end of the system, extensions for multiple-dimension arrays and structures are of significant interest, especially for image data in the video domain.

References

[1] SGS-Thomson Microelectronics, "D950-CORE Preliminary Specification", January 1995.

[2] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, A. Wang, "Code Optimization Techniques for Embedded DSP Microprocessors", *Design Automation Conf.*, June, 1995, pp. 599-604.

[3] The Corporate Software Integrator, "Lode DSP Engine: Preliminary Data Sheet", May 1995.

[4] L. Bergher, X. Figari, F. Frederiksen, M. Froidevaux, J.M. Gentit, O. Queinnec, "MPEG Audio Decoder for Consumer Applications", *CICC*, 1995.

[5] C. Liem, P. Paulin, M. Cornero, A. Jerraya, "Industrial Experience using Rule-driven Retargetable Code Generation for Multimedia Applications", *Int. Symposium on System-Level Synthesis*, Sept. 1995.

[6] "ZR38500 Six-channel Dolby Digital Surround Processor: Preliminary Specification", Zoran Corporation, Nov. 1994.

[7] C. Liem, T. May, P. Paulin, "Instruction-Set Matching and Selection for DSP and ASIP Code Generation", *European Design & Test Conference*, Feb 1994, pp. 31-37.

[8] D. Bacon, S. Graham, O. Sharp, "Compiler Transformations for High-Performance Computing", *ACM Computing Surveys*, Vol. 26, No. 4, December, 1994, pp. 345-420.

[9] V. Zivojnovic et al, "DSPstone: A DSP-Oriented Benchmarking Methodology", *Proc. of the Int. Conf. on Signal Processing and Technology (ICSPAT)*, Dallas, Oct. 1994.

[10] A. Aho, R. Sethi, J. Ullman, Compilers: Principles, Techniques and Tools, Addison-Wesley, Reading, Ma., 1988.

[11] U. Banerjee, Loop Parallelization, Kluwer AcadJanc Publishers, 1994, 171 pages.

[12] J.R. Larus, "Efficient Program Tracing", *IEEE Computer*, May 1993, pp. 52-61.

[13] C. Liem, T. May, P. Paulin, "Register Assignment through Resource Classification for ASIP Microcode Generation'', *Int. Conference on Computer Aided Design*, Nov 1994.

[14] P. Paulin, C. Liem, T. May, S. Sutarwala, "FlexWare: A Flexible FirmWare Development Environment", in Code Generation for Embedded Processors, ed. by P. Marwedel, G. Goossens, Kluwer Academic Publishers, 1995.