Retargetable Estimation Scheme for DSP Architecture Selection

Naji Ghazal, Richard Newton, Jan Rabaey

Department of Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720
email: {naji, newton, jan}@eecs.berkeley.edu

Abstract— Given the recent wave of innovation and diversification in digital signal processor (DSP) architecture, the need for quickly evaluating the true potential of considered architectural choices for a given application has been rising. We propose a new scheme, called Retargetable Estimation, that involves analysis of a high-level description of a DSP application, with aggressive optimization search, to provide a performance estimate of its optimal implementation on the architectures considered. With this scheme, we present a new parameterized architecture model that allows quick retargeting to a wide range of architectural choices, and that emphasizes capturing an architecture's salient optimizing features.

We show that for a set of DSP benchmarks and two full applications, hand-optimized performance can be predicted reliably. We applied this scheme to two different processors.

I. Introduction

With today's rapidly increasing use of signal processing and multi-media applications in embedded systems, Digital Signal Processors (DSPs) have emerged into the mainstream of embedded systems with architectures growing in diversification and innovation. Unfortunately, high-level software development support for DSP has remained far behind, whether for conventional programmable DSPs, new RISC DSPs, or DSP ASIPs (Application-Specific Instruction-set Processors). Due to the irregularity of the architectures, even the latest DSP compilers cannot exploit the architecture's crucial optimization features to reach optimal implementation, without the use of the designer's in-depth knowledge of the architecture. New retargetable DSP compilers ([1], [2], [3]), suffer from the same lack of optimizing technology. So it is challenging and time-consuming to explore the true potential of different architectural choices.

Most DSP applications, however, have special characteristics (predictable data access patterns, execution time locality, computation regularity, etc.[4]) that potentially allow for valuable estimation of run-time behavior on a given architecture, from behavioral specification. With such estimation, optimizing uses of a processor's special architectural features, and hence the true potential of architectural choices, can be exposed to the designer without the need for in-depth expertise in programming the processor.

In this work, we show that early estimation of and guidance toward a DSP architecture's hand-optimized performance can be achieved with static and dynamic code analysis, even for today's latest architectures (from conventional, to VLIW, to user-extendible DSPs). We have devised a new performance estimation scheme that uses a model of the architecture that captures its architectural composition as well as its differentiating, and frequently special-purpose, optimizing features. Our estimator takes a DSP application described in a high-level language (C in the current implementation), uses profiling and aggressive optimization search, and outputs a performance estimate and a trace of the application annotated with the suggested optimizations and ranked bottlenecks. This scheme reaches near hand-optimized performance by using optimization techniques beyond the instruction level, along with well-tested assumptions about the characteristics of DSP applications and processors.

In this paper, we start by describing how this new technique relates to existing approaches. We then explain the retargetable estimation scheme and the accompanying architecture model. We then demonstrate the feasibility of this technique as it is applied to two competing yet different architectures. Finally, We conclude with a summary of this scheme's applications and considerations for future work.

II. Related Work and Our Contributions

Existing approaches for DSP performance estimation are either increasingly unreliable or prohibitively expensive and time-consuming. First-order estimation of DSP performance is commonly done by correlating a DSP application to a set of representative kernels that have been hand-optimized in assembly language and benchmarked on several processors [5]. Unfortunately, as applications grow more complex, it has become less feasible to identify a small set of DSP kernels to represent them. This approach is also limited to the set of processor architectures that have been benchmarked.

There have been efforts in architecture evaluation([6], [7]) using static and dynamic code analysis that have focused on custom DSP or ASIP designs. However, they use architectural models that capture only the functional components of the machine and their interconnections and constraints. While that is suitable for exploring different instruction sets and functional units, these approaches do not

consider many crucial DSP-oriented high-level optimizing features. Such features as support for packing of memory accesses, and their subsequent optimizations (e.g. automatic unpacking of a double-word into a MUL), across loop iterations are found with increasing frequency in today's DSPs and can have substantial impact on performance in DSP applications.

The alternative in evaluating architectures is to implement the application on several processors: generate assembly code, manually search for optimizations, and simulate, for each choice. While there are retargetable compilers that allow examination of different choices, their optimization technology remains insufficient to eliminate the need for in-depth knowledge of the architecture and hand optimization. In addition, the architecture models used in them ([8], [9], [10]) tend to involve highly detailed descriptions of the processor, as they are geared toward correct code generation or simulation, and are hence difficult to retarget.

With retargetable estimation, there is no need for generating assembly code, simulating, and iteratively hand-optimizing on each architecture considered, whether doing so for kernel benchmarks or the full application. In addition to providing a performance estimate in a single iteration, the retargetable estimator generates a trace of the application that guides the designer through the potential optimization opportunities offered by the target processor architecture. There are several uses for early estimation of DSP performance, including:

- Quick quantitative comparison of multiple architectures
- Guidance toward improved use of architectural features: *(more patterns and optimization opportunities than can be explored by compilers)*
- Evaluation of the impact of different User-Configurable extensions of an architecture, on given application
- Quick comparison of different versions of an algorithm on a given architecture

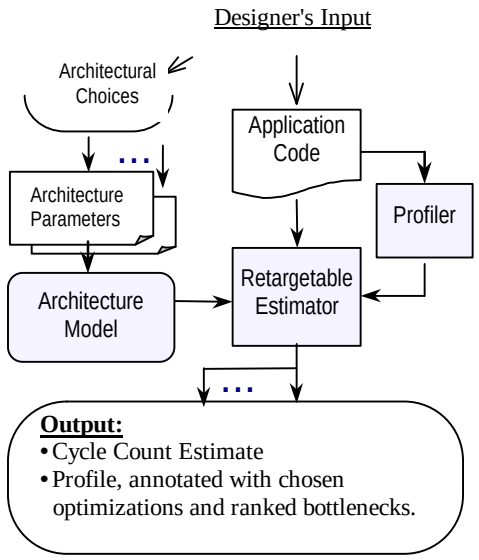

Fig. 1 Overall Flow of Retargetable Estimation

III. Retargetable Estimation Scheme

The overall flow of this technique is shown in Fig. 1. Each processor architecture considered for evaluation is described by the architecture model (designer can choose from a database of previously captured architectures or enter his/her own), and the DSP application to be examined is described in a high-level language (C in the current implementation). Prediction of optimal run-time behavior is achieved through profiling and a series of passes that search for generic optimizations, and more importantly, for the architecture-specific optimization features. As most DSP applications are dominated by loop-intensive kernels, heavy emphasis is placed on loop-level optimizations. The estimator provides the designer with a total cycle count estimate, and a profile of the code, annotated with the chosen optimizations and a ranking of the dominant code segments.

A. Parameterized Architecture Model

Several factors allow the aggressive search for optimizing features while keeping the architecture model sufficiently light to keep it retargetable. The architecture model, whose template is shown in Fig. 2, is described using a set of variable-length parameter tables, arrays, and scalars that capture the functional unit composition, the instruction set, and the special optimizing features available in the architecture.

Functional Composition/Topology: • Functional Unit (FU) Types • FU Usage Limits • FU Latencies and Throughputs • FU-to-FU Constraints	Special Optimization Features: •Optimized Computation: • Multi-operation Patterns (e.g. dual-MAC, Packed Load/Add) • Special Arithmetic modes (e.g. saturation, rounding) •Memory Addressing Features: • Address Generation Costs • Hardware Circular Addressing: # and size of Circular buffers •Control Flow Optimization Features: • Simple If-Else Elimination • Looping Support (# of Loop counters) • Loop Vectorization/Packing (List of "packable" instructions)
Instruction Set: • List of Instruction Type: default: Add/Sub, ALU (gen.), Mul, MAC, Load, Store, Br. • Max Instructions in Parallel • Instructions' FU Usage (table for each operand size) • Operand Handling Rules (List of instructions that use 3 operands, the rest allow 2 only)	

Fig. 2 Parameterized Architecture Template

The machine description template captures the differentiating features of the instruction set and special functional resources, rather than the complete specification required for code generation or simulation. For instance, all instructions that exercise the same functional units and have the same latency can be captured by one instruction type in the model. The model also captures the three major types of optimizing features found in today's DSPs: optimized special operations, memory addressing support, and control-flow (most importantly loop-level) optimization support. Such features include those that have not been well targeted by DSP compilers or previous estimation schemes, such as:

- Specific instruction packing
- Memory pack/unpack support
- Loop vectorization (e.g. processing two iterations in one, i.e. halving the trip count)
- Complex optimized arithmetic patterns for stream data processing (dual MAC, ComplexMUL, etc.).

Well-tested assumptions about the implementation of some of those features in typical DSPs allow for a simplified means of describing them in the architecture template. For instance, only the different costs of addressing modes need to be specified, with the conditions to use them, rather than each address mode available. This is made possible by using the assumption that most modern DSPs handle memory array addressing similarly.

FU Types	Limits	Latency	Throughput
ALU	2	1	1
MAC	1	1	1
Prefetch-Lds	2	0	1
Load	1	1	1
Store	1	1	1
Branch	1	1(miss=3)	1

FU-to-FU Latency: Str->Ld 1 cycle min

Instruction Types: (12)
Ld, Pref-Ld, Str, Branch
Add, Sub, ALU(all others)
Mul, MAC, Add2, Sub2, MAC2
Max Instructions issued in Parallel: 4
Instructions'FU Usage
(16- and 32-bit 12instr x 6FU tables)
Operand Handling Rules:
Instructions allowing 3 op'ds per instr:
MAC, MAC2, Add2, Sub2

Optimized Computation:
- Multi-operation Patterns:
(SUIF Expression trees)
MAC: x += ? * ?
Ld2/Str2: pair of x[..], x[..+1] in same basic block
MAC2: x += x[…] * y[…] + x[… +1] * y[… +1]
Add2/Sub2: …
- Special Arithmetic modes:
saturation, Rounding

Memory Addressing Features:
- Address Generation Cost:
1 cycle, if not a register, and if not array with offset [-8..7]
- Hardware Circular Addressing:
2 circular buffers

Ctrl-Flow Optimization Features:
- Looping Support:
FOR Loop cost = 0
- Loop Vectorization/Packing:
(List of eligible instructions)

Fig. 3 Architecture Model of LSI401 superscalar DSP

One of the processors studied in this method is LSI Logic's LSI401 (formerly ZSP Corporation's ZSP16401) 4-way superscalar Fixed-Point DSP[11], whose model is provided as an example in Fig. 2b. For this processor, the most advantageous features are its single-cycle dual MAC (multiply-accumulate) and its dual prefetched load-doubles (32-bits -- 2x16-bit words -- each load, 0-cycle latency). A pattern for finding pairs of adjacent array accesses (pattern Ld2 in Fig. 3) is used, whether through explicit array indexing or pointer stepping. Another pattern is used for joining any two Ld2's into any dual mul-add/sub set of operations (capturing all instructions that employ ZSP's dual-MAC). To allow the estimator to expose more potential uses of this set of features, such patterns are also searched for at the loop level, that is, across adjacent iterations of a loop body. Given that complex memory aliasing is not common practice in DSP algorithm implementation, more opportunities can be found for such an optimization than compilers can achieve, with a little risk of misprediction. As the aim is estimation, and not code generation, we make simplifying assumptions about the architectures that are typically not restrictive in the DSP domain. Details we have not found necessary to capture in the architecture model are register allocation conflicts, and branch and cache misses, as in most DSP applications, we have found their effect to be negligible.

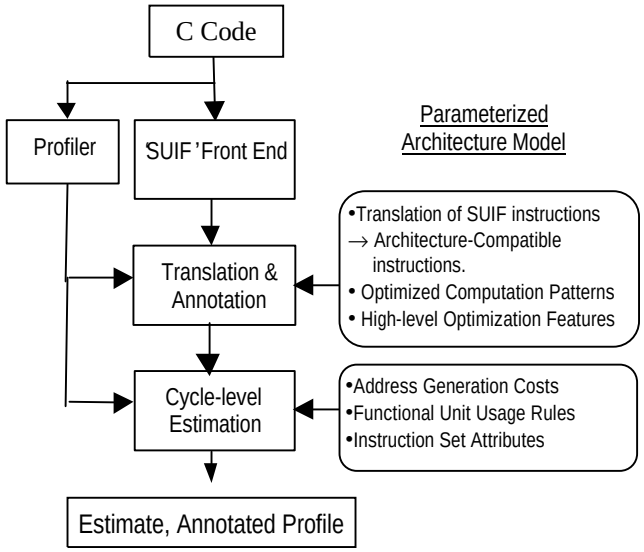

Fig. 4 Flow of Esimtation Scheme

B. Estimation using Static and Dynamic Code Analysis

In our approach, a DSP application is described in a high-level language and is processed for estimation as shown in Fig. 4. The application is first transformed into an intermediate representation using the Stanford University Intermediate Format (SUIF) compiler front-end [12]. The application is also executed on the host machine to gather profiling information, needed to predict dynamic control flow in the application.

1) Architecture-specific Translation and Annotation

The first phase of code analysis begins with translating any incompatible SUIF instructions to those allowed in the target architecture. Second is a series of passes performed on the code (the abstract syntax tree) that search for gradually more complex optimizations. The first pass is for identifying building blocks that can lead to optimizations, such as "stream data" array access patterns (load/store instructions to adjacent memory elements in the same basic block, and loop-index based array accesses). Once stream accesses are identified, it becomes easy to expose many of the crucial optimization opportunities found in today's DSPs. All opportunities for using the optimized arithmetic patterns listed in the architecture model are identified, followed by loop-level optimizations targeting optimized processing of streams. The types of architecture-specific optimization that this estimator currently handles include: Optimized Multi-Operation Pattern Matching (e.g. MAC), Address Mode Optimization (e.g. auto-update, circular buffer), Loop Optimization (e.g. Zero-overhead Looping), Loop-optimized Pattern Matching (e.g. pre-fetched sequential Loads), Loop

Vectorization/Packing, and Simple If-Else Conversion (by use of predicated instructions). Optimizations considered in this scheme also include those in generic compiler technology: rescheduling within basic blocks, loop unrolling, and software pipelining (as is crucial for VLIW processors such as the TI C6x DSP).

2) Cycle-level Estimation

The last phase involves traversing the code to emulate the execution of the instruction stream. The code traversal is hierarchical, starting with procedures, then high-level constructs (For Loops, If-Else branches), and finally, instructions. For each loop, the cost (cycle count) is estimated using its trip count, computed or predicted by the profiler, multiplied by the cost of one iteration of its body, along with any overhead incurred by the chosen loop-level optimizations. For each instruction, the estimator first checks for operand handling rules (e.g. accounts for cost of splitting, a 3-operand instruction into two, if needed). Then it checks if the instruction is part of any optimized patterns and computes address generation costs. Each basic block is scheduled with list scheduling, which involves checking for each instruction's data and resource dependencies on previous instructions. That is followed by an attempt to perform software pipelining using a standard software pipelining algorithm[13]. The lower of the software pipelined and unpipelined cycle counts is chosen in the estimation.

At the end, a post-processing step is taken to rank the procedures traversed in the application by cycle-count dominance, and if desired, the loops of the application are similarly ranked. The output shows a cycle-by-cycle estimated trace to expose the way the code can be transformed and treated by the target architecture to reach the corresponding optimized performance.

IV. Results

We have applied this estimation method to two of today's latest competing programmable DSP processors that differ significantly in their architectures: LSI Logic's LSI401[14], and Texas Instruments' 320C6201 [15]. Both have a clock frequency of 200MHz. The LSI401 is a new 4-way superscalar DSP with many conventional DSP features such as single-cycle operations including MACs, and hardware loop support. By contrast, the C6x is a RISC-like 8-way VLIW DSP with multi-cycle simple instructions. In both cases, the architecture model did not take longer than two weeks to capture the DSPs sufficiently. In comparison, in the case of an expert designer using the SPAM retargetable compiler[1], it took approximately one month to retarget to a simple DSP architecture. For each DSP, we have obtained hand-optimized assembly-code versions of a set of common DSP benchmarks from the DSP vendors, and measured their actual performance, then we took their generic versions in C language (with no architecture-specific optimizations) and ran them through the estimator. To verify that it is not limited to small DSP loop kernels, we also tested the estimation tool on two full DSP applications, targeting the ZSP processor. We used C-language implementations of an IS-136 Viterbi decoder and a VSELP (vector-sum excited linear predictor) speech encoder. The latter is the larger of the two, with approximately 1400 lines of code, and two dominant procedures: "Compute Lag" and "Codebook Search."

Table 1 Cycle Count Estimates of Benchmarks & Applications

DSP Kernel N (Inputs) = 64 T (Taps) = 8	LSI401 (200MHz)			TI 'C6201 (200MHz)		
	Predicted	Actual	% Error	Pred.	Act.	% Error
Vector-Scalar Multiply	133	133	0	84	86	1.0
FIR Filter (Real)	518	521	0.6	239	238	0.4
FIR Filter (Complex)	1728	1800	4.0	1040	1034	0.6
IIR (Biquad)	46	48	4.2	33	32	3.1
DSP Application						
IS-136 Viterbi Decoder	3990	4361	8.5			
VSELP Encoder (per frame)	0.85 M	1.01 M	16			

Note: "Actual" cycle counts here refers to vender-supplied extensively hand-optimized versions of the code.

The results of this experiment show that for DSP kernels, most commonly dominant in DSP software, the estimation accuracy, as compared to manually achieved optimal implementation, did not exceed 5%. For a large DSP application, such as a speech encoder, prediction of hand-optimized performance was within 16%.

V. Conclusions

With static analysis of an application's behavioral description, and a model that captures the differentiating features of a processor's architecture, a framework has been developed to allow for useful prediction of the true potential of an architecture for meeting performance constraints. In addition, this technique provides much needed guidance toward improved use of a DSP architecture's features. What differentiates this approach from the main efforts in architecture exploration is that there is no need for generating assembly code and simulating on each architecture considered, which is usually iterative and time consuming, and requires in-depth expertise in programming the different processors being considered. On the other hand, retargetable estimation does not rely on partial representation of the application by a set of kernels with published results on a set of processor choices.

In the future, we plan to apply this method to more architectures and investigate the inclusion of other design metrics such as power consumption in this early estimation scheme, to enable the evaluation of different aspects of the considered architectural choices.

REFERENCES

[1] A. Sudarsanam, *Code optimization libraries for retargetable compilation for embedded digital signal processors.* PhD. thesis. Princeton University, Department of EE, May 15, 1998.

[2] S. Hanono and S. Devadas, "Instruction selection, resource allocation, and scheduling in the Aviv Retargetable Code Generator." Proceedings of the ACM/IEEE Design Automation Conference, June 1997.

[3] V. Zivojnovic, S. Pees, and C. Schlager, "DSP processor/compiler co-design: a quantitative approach." Proceedings. 9th International Symposium on System Synthesis, pp.108-13, 1996.

[4] Bier, J. "Processors for DSP--the options multiply," The Embedded Processor Forum. MicroDesign Resources, 1998.

[5] Bier, J., et. al., Evaluating DSP processor performance," white paper, Berkeley Design Technologies Inc. (BDTI), 1996.

[6] M. Yamguchi, et. al., "Architecture evaluation based on datapath structure and parallel constraint." Proceedings of IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC), pp. 503-08, January 1997.

[7] J. Gong, D. Gajski, A. Nicolau, "Performance evaluation for application-specific architectures," IEEE Trans. on VLSI, vol. 3, pp. 483-90, December 1995.

[8] A. Fauth, J. Van Praet, and M. Freericks. "Describing instruction set processors using nML**."** Proceedings of the European Design and Test Conference. ED&TC, p503-7, 1995.

[9] G. Hadjiyiannis, S. Hanono, and S. Devadas, " ISDL: an instruction set description language for retargetability." Proceedings of the ACM/IEEE Design Automation Conference, 34th DAC, pp. 299-302, 1997.

[10] V. Zivojnovic, S. Pees, and H. Meyr, "LISA-machine description language and generic machine model for HW/SW co-design." VLSI Signal Processing, IX, pp. 127-36, 1996.

[11] LSI Logic Corporation (formerly ZSP), "LSI401Z DSP," technical note. http://www.zsp.com/pdf/LSI401.pdf, 1998

[12] Stanford Compiler Group. *The SUIF Library*, version 1.0, http://suif.stanford.edu. 1994

[13] B. Rau, "Iterative modulo scheduling." International Journal of Parallel Programming, vol.24, pp.3-64, Feb. 1996.

[14] LSI Logic Corporation (formerly ZSP Corporation), *The ZSP164xx Programmer's Reference Manual.* 1997.

[15] Texas Instruements. *TMS320C6000 CPU and instruction set reference guide* http://www.ti.com/sc/psheets/spru189d/spru189d.pdf, 1997.