Compiler Analysis and Optimizations: What is New?

Uday Khedker[1] and R Govindrajan[2]

[1] Dept. of Computer Science and Engineering,
Indian Institute of Technology Bombay.
(uday@cse.iitb.ac.in)
[2] Dept. of Computer Science and Automation,
Indian Institute of Science, Bangalore.
(govind@csa.iisc.ernet.in)

Abstract. Traditional compiler analyses and back-end optimizations, which play an important role in generating efficient code for modern high-performance processors, are quite mature, well understood, and have been widely used in production compilers. However, recent advances in high-performance (general purpose) processor architecture, emergence of novel architectural paradigms, emphasis on application-specific processors and embedded systems, and the increasing trend on compiling applications directly onto silicon present several interesting challenges and opportunities in high performance compilation techniques. In this paper we discuss the trends that are emerging to meet the above challenges. In particular, we discuss recent advances in data flow analyses, compiling techniques for embedded and DSP processors, and compiling techniques that reduce power consumption.

1 Introduction

Traditional compiler analyses and back-end optimizations, which play an important role in generating efficient code, are quite mature, well understood and have been routinely employed in production compilers. They have been further advanced by constant research necessitated by recent advances in processor architecture and related areas. This paper concentrates on interesting challenges and opportunities in compiling for high performance architectures. We do so by observing the emerging trends.

High-performance (general purpose) processor architectures have witnessed many advances in the form of Very Large Instruction Word (VLIW) architecture, clustered VLIW architecture, and the emergence of novel architectural paradigms such as Explicitly Parallel Instruction Computing (EPIC) architecture. These advances have put tremendous stress on machine dependent compiler optimizations like instruction scheduling, software pipelining, and register allocation and have also put unconventional demands on machine independent optimizations like redundancy elimination. Developments in exploiting subword parallelism, e.g., those found in multimedia extensions (MMX) or visual instruction set (VIS) extensions, have not only demanded compiler technology to identify and expose available parallelism [16, 17, 24], but have also triggered a number of research activities on bitwidth-aware compilation [23, 25, 26]. The growing complexities of digital signal processors have also pushed compiler technology in several fronts, including address arithmetic optimizations, memory layout optimizations, and instruction scheduling [18]. Compiling for embedded systems raises issues similar to those that occur in DSP processors and additionally imposes constraints on power/energy consumption, code size, and real-time performance. These are only a few examples, but they illustrate well the demands placed on compilation techniques.

The above developments have also shifted the focus from a purely performance-oriented viewpoint. Reducing power/energy consumption and/or reducing code size are also important objectives. Further, traditional compiling (in production compilers) have placed stringent demand on the compilation time which often has enforced a trade off between the compile time and the quality of the code produced. Compiling for embedded systems, on the other hand, relaxes the constraints on compile time, enabling highly optimized code–optimized in terms of performance, power/energy, and/or code size.

Among the commonly used methods of analysis for optimizations, data flow analysis is the most systematic method of discovering relevant information. Most other methods are specific to optimizations being performed whereas data flow analysis is applicable to a large set of optimizations. Traditionally, data flow analysis has been used in the context of machine independent optimizations. However it is more general and can be exploited in other contexts too. This has led to some interesting advances in data flow analysis.

In this paper we discuss emerging compiling techniques with the above perspective. In particular, we discuss recent advances in dataflow analyses, compiling techniques for embedded and DSP processors, and compiling techniques that reduce power consumption or code size. We focus mainly on back-end optimizations and the required analyses. The primary intent of this paper is to observe trends rather than provide a comprehensive survey. For brevity, we include only the recent references and wherever possible, cite surveys and/or book chapters rather than individual papers.

Section 2 deals with traditional analyses and optimizations briefly. Section 3 discusses recent advances in data flow analyses and optimizations. We present concluding remarks in Section 4.

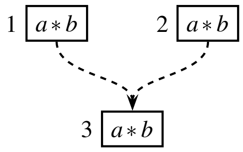
2 Traditional Compiler Analysis and Optimizations

2.1 Data Flow Analysis

Data flow analysis is almost always a static analysis[3] based on solving constraints which are usually expressed in the form of linear simultaneous equations called *data flow equations*. The domains of values computed by these equations represent the *data flow information* to be derived and are usually some kinds of finite sets (also viewed as bit vectors), although other kinds of values (eg. tuples, graphs etc.) have also been used. These equations are typically solved using *iterative* methods by repeatedly refining conservative initial values. An alternative approach based on *elimination* methods uses structural properties of programs to avoid repeated computations. Both the approaches compute the *Maximum Fixed Point* solution of data flow equations.

Figure 1 illustrates a well known optimization called Common Subexpression Elimination (CSE) which uses a classical data flow analysis called Available Expressions

[3] Analyses used in Dynamic Compilation, interactive debugging etc. are dynamic.



- Along path $1 \rightsquigarrow 3$: Neither a, nor b is modified. Hence the value of $a*b$ in block 3 is same the value of $a*b$ in block 1.
- Along path $2 \rightsquigarrow 3$: Neither a, nor b is modified. Hence the value of $a*b$ in block 3 is same as the value of $a*b$ in block 2.
  $\Rightarrow a*b$ in block 3 is a redundant recomputation. Previously computed value of $a*b$ can be used in block 3.

Fig. 1. Common Subexpression Elimination

Analysis (AvEA). Intuitively, an expression e is available at a program point u, if *every* path *from program entry to u* contains an evaluation of e which is not followed by a redefinition of any operand of e. For each program point, the maximal set of expressions which are available is determined; If any expression is computed at a program point and is also available at that point, its computation at that point is eliminated.

Let b and $Pred(b)$ denote a basic block[4] and its (immediate) control flow predecessors. Let Gen_b denote the set of expressions which are computed in b whose operands are not subsequently modified in b. Let Kill_b denote the set of expressions whose operands are modified in b. The equations defining IN_b and OUT_b (i.e. the set of expressions available at the entry and exit of b) are :

$$\mathsf{IN}_b = \begin{cases} \emptyset & \text{if } b \text{ is the entry block} \\ \bigcap\limits_{p \in Pred(b)} \mathsf{OUT}_p & \text{otherwise} \end{cases} \tag{1}$$

$$\mathsf{OUT}_b = \mathsf{Gen}_b \cup (\mathsf{IN}_b - \mathsf{Kill}_b) \tag{2}$$

Equation (2) can also be written in terms of a *flow function* f_b :

$$\mathsf{OUT}_b = f_b(\mathsf{IN}_b), \text{ where } f_b(X) = \mathsf{Gen}_b \cup (X - \mathsf{Kill}_b) \tag{3}$$

We use the above description of AvEA to motivate the following generic abstractions :

Data Flow Values. Traditional values have been sets of facts asserting specific properties of data items in a program. These sets can also be viewed as bit vectors containing one bit for every data item indicating one of the two possible values (*must-be-available*, *is-not-available* in the case of AvEA). Among common data flow problems, the only exception has been Constant Propagation Analysis (CPA) in which a data flow property for a variable can range through three values (*is-undefined*, *is-not-constant*, *is-a-constant-with-value-k*).

Flow Functions. Most traditional data flow problems have constant Gen_b and Kill_b for a block implying that data flow information generated and invalidated in a block is fixed. For CPA fixed Gen and Kill cannot be defined.

Direction of Data Flow. In a *forward* data flow problem the data flow information of a block is influenced by its control flow predecessors (as in AvEA). In a *backward* data flow problem, data flow information is influenced by successors. In *bidirectional* data flow problems like Partial Redundancy Elimination (PRE), data flow information is simultaneously influenced by predecessors and successors. However,

[4] A (single entry, single exit) sequence of statements.

PRE is not truly bidirectional and can be solved by a sequence of unidirectional data flows.

Confluence Operation. Data flow information is merged using set union (or boolean OR) operation if the relevant information *may* hold along any path reaching a block. It is merged using set intersection (or boolean AND) operation if the relevant information *must* hold along all paths reaching a block. Non-boolean data flow values (eg. data flow values in CPA) require special confluence operations depending upon the properties of values. In any case, all traditional data flow problems have a single kind of confluence.

More details about data flow analysis can be found in [7, 9].

2.2 Traditional Optimizations

Traditional machine independent optimizations such as copy propagation, constant folding, common subexpression elimination, loop invariant code motion, dead-code elimination, strength reduction, partial redundancy elimination, and other control flow optimizations are typically performed in a low-level intermediate code. The resulting optimized low-level intermediate code is then translated to target machine code. Instruction scheduling (both local and global) and register allocation are performed on the machine code. In this section we restrict ourselves to instruction scheduling, register allocation, and software pipelining.

Instruction scheduling is performed to reduce the pipeline delay stalls and to expose (instruction-level) parallelism for superscalar and VLIW architectures [6]. Basic block instruction scheduling reorders instructions within a basic block obeying data dependences and is based typically on a list scheduling approach; a number of heuristics have been proposed to select the next instruction to be scheduled. Global scheduling deals with moving instruction across basic blocks and can be classified as structure-driven or profile-driven. The profile-driven approach identifies the most frequently executed paths using profile information and coalesces them into an extended basic block. Trace scheduling, superblock scheduling, region-based scheduling are a few examples of profile-driven global scheduling methods. In contrast, the structure-driven approach identifies and attempts to improve performance along all execution paths by moving operations between basic blocks, considering program structure and without using profile information. Examples of the structure-driven approach are treegion scheduling and percolation scheduling.

Register allocation is another important compiler optimization that decides which variables are allocated to registers. The problem of global register allocation can be viewed as a graph coloring problem [3]. Starting from the simple heuristic method, a number of register allocation methods have been proposed including the priority-based register allocator, a coalescing register allocator, a clique-based register allocator, a linear-scan register allocator, and a register allocator based on program dependence graph. When the number of live ranges at a program point exceeds the number of available registers, some of the variables are *spilled*. Live range *splitting* is an optimization over live range spilling, and register allocation methods based on live range splitting have been proposed. Refer to [5] for a recent and exhaustive survey of register allocation methods.

Register allocation and instruction scheduling phases have a tight phase coupling in that they are influenced by the order in which they are performed. When instruction scheduling is performed first, it increases the register pressure causing a higher number of register spills. On the other hand, if register allocation is performed first, it introduces data dependences in the form of anti- and output-dependences which constrain the instruction-level parallelism which can otherwise be exploited. A number of studies on the phase ordering of register allocation and instruction scheduling have been reported including a few integrated register allocation and instruction scheduling methods. Refer to [6] for a recent discussion on the phase ordering issues.

Software pipelining schedules instructions in iterative computations such that execution of instructions from different iterations is overlapped. Modulo scheduling is a software pipelining method in which instructions from successive iterations are scheduled at a fixed initiation interval. A number of heuristics have been proposed to construct a modulo schedule. Rate-optimal resource-constrained software pipelining problem has been formulated as an integer linear programming problem. Similar to instruction scheduling, software pipelining also increases the register pressure. Hence a number of methods that are sensitive to registers have also been proposed. Software pipelining methods are reviewed in [1].

3 Recent Advances in Analysis and Optimizations

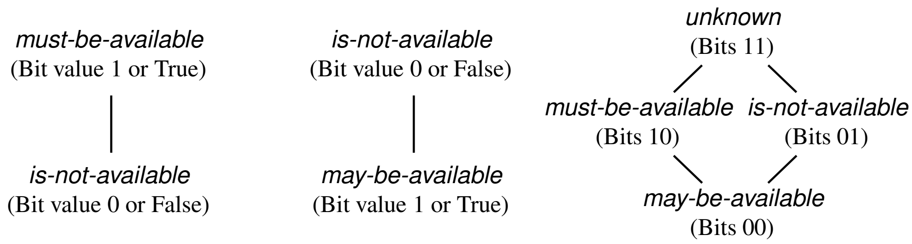
3.1 Data Flow Analysis

Traditionally data flow analysis is viewed as working mostly on "bit vectors" (or sets). Much more interesting information can be derived from programs by a combination of the following advances for each of the abstractions listed in Section 2.1.

Data Flow Values. Traditional *two-valued* (i.e. True-False kind of) data flow information is more definite in one value than the other. For example, when AvEA concludes that an expression *must-be-available*, it is guaranteed to be available along all paths. However, when an expression *is-not-available*, it includes two possibilities : either the expression is not available along any path or it is available along some path(s) but not along all. More precise analysis should have one more value *may-be-available* to allow a distinction between the two. Traditionally this has been achieved by performing two data flow analyses with the lattices shown in Figure 2(a) and (b) :

- *Partial* availability which computes values *may-be-available* (i.e. "available along some path") and *is-not-available* (i.e. "not available along any path").
- *Total* availability which computes *must-be-available* (i.e. "available along all paths") and *is-not-available* (i.e. "not available along some path").

Recent data flow analyses [2, 12] combine the *may/must* values into a single data flow analysis using the lattice shown in Figure 2(c). Note that this is a general lattice and can be used for any kind of *may/must* data flow information (eg. alias information). Unless the data flow information requires special representation, this lattice can be implemented using bit vectors. Although Figure 2(c) implements it using bitwise



(a) Total availability
Confluence: ∩ or AND

(b) Partial availability
Confluence : ∪ or OR

(c) Precise availability
Confluence: ∩ or AND

Fig. 2. Lattices in available expressions analyses.

AND, bitwise OR can be used by interchanging the bit representations of *unknown* and *may-be-available*.

Data flow values other than bit vectors are also becoming common. [12] proposes a data flow analysis for static type inferencing for dynamically checked languages. This analysis uses sets of tuples for data flow values to distinguish between (a) *may* and *must* information, and (b) between type information flowing from ancestors and descendants. [14] uses a graph to represent a set of sequences of names (eg. *x.next.left.next*) representing reference of a heap address. In the presence of loops, neither the length nor the number of such possible sequences is bounded. A graph representation bounds it leading to a complete lattice making it amenable to data flow analysis.

Flow Functions. Traditional Gen and Kill terms used to define common flow functions have been constant (i.e. fixed for a basic block). However, data flow analysis is not restricted to such flow functions. The Gen and Kill used in the heap reference analysis [14] for determining liveness of heap references, the references generated as live in a block include aliases of the references live at the exit of the block. Hence the Gen_b is a function of OUT_b.

Direction of Data Flow. The original formulation of PRE [20] involves bidirectional dependencies which cannot be characterised by the classical theory of data flow analysis. Hence many efforts were directed towards devising variants of PRE based on purely unidirectional flows [10]. However, formal theory and practical methods for bidirectional data flow analysis have been known for some time and it is now possible to devise data flow analyses which involve truly bidirectional dependencies (i.e. dependencies involving simultaneous forward and backward flows) [9]. Static type inferencing for dynamically checked languages [12] uses bidirectional dependencies. An optimizing compiler for Intel's IXS 1000 Media Signal Processor uses bidirectional dependencies for achieving a tradeoff between redundancy elimination and generating MAC-like instructions [13]. The former requires dividing computations into sub-computations so that redundant computations can be eliminated. The latter requires combining computations to form a composite computation for which a MAC-like instruction can be generated.

Confluence Operation. All traditional data flow problems have a single kind of confluence (*Any paths*, *All paths*, or a special confluence which cannot be categorised as any/all paths). This implies that the data flow information being derived depends on the information along either all paths or some paths. For a given data flow problem :

- The data flow information derived at all points may depend on either all paths or on some paths. For total availability analysis, it is the former while for the partial availability analysis, it is the latter.
- The data flow information derived at some point may depend on all paths while at some other point it may depend on some paths; eg. in precise availability analysis, an expression may be partially available at a program point (*may-be-available*) while at some other program point, the same expression may be totally available (*must-be-available*).
- The data flow information derived at all points may have a component which depends on *all* paths and another component which depends on *some* paths. The placement possibility of an expression in a variant of PRE [11] involves both AND and OR. Theoretically, the desired fixed point solution of such data flows is different from (and lower than) the maximum fixed point solution [4] and characterising them is still an open problem [9]. Practically, they are solved in two steps : an approximation step which (conservatively) ignores some confluence and a refinement step which incorporates the effect of the previously ignored confluence.

3.2 Optimizations

This section discusses recent trends in three specific areas which are driven by recent advances in both general purpose and application specific processor architectures.

Power-Aware Compilation. Reducing power and energy consumption is becoming increasingly important especially in embedded systems and application-specific processors. In this discussion we concentrate on instruction scheduling and software pipelining methods for reducing power/energy consumption.

Dynamic energy consumed can be reduced by reducing the switching activities on the system bus — toggling of signals from zero to one or vice-versa — through careful reordering of instructions. For architectures that employ multiple functional units which operate under different supply voltage and clock frequency, an alternative approach exploits the slacks in instructions that are not in the critical path by scheduling them on slow functional units [28]. The leakage energy consumed can be reduced [15] by carefully limiting the instruction-level parallelism across loops by limiting the number of functional units used based on the available parallelism at loop-level granularity; the power dissipated by other functional units is controlled either through clock or voltage gating.

Slacks in scheduling non-critical instructions in a software pipelined schedule have been exploited to reduce energy consumption [27]. The problem of constructing software pipelined schedules with optimal energy has been formulated as an integer linear programming problem. This or a similar approach can be used to reduce dynamic energy as well as leakage energy or power variations, measured in terms of hot spots, peak power, and step power. In [8], compiler techniques to identify program regions that can be slowed down without significantly affecting the performance have been proposed. While executing these program regions, functional units can be slowed down by applying lower CPU voltages and frequencies (through dynamic voltage scaling).

Compilation Techniques for Application-Specific Computation. Challenges in compiling for application-specific processors arise because of the increasing complexity of the processors in terms of their irregular data paths, heterogeneous registers, dedicated address generation units, chained instructions such as multiply-and-accumulate, and parallel memory banks. Instruction scheduling methods for DSP processors starting from simple list scheduling based heuristics to code compaction techniques, integer programming methods, combined register allocation and instruction scheduling, have been proposed in the literature [18].

DSP processors contain multiple (typically two) memory banks. A proper partitioning and layout of program variables (especially arrays) on these banks is important for exploiting parallel access of variables in a single cycle. For exploiting such memory organizations, these techniques should either be integrated with or at least communicate with instruction scheduling. Refer to [18] for a detailed survey on code generation and optimization for embedded and DSP processors.

Efficient usage of address generation units in DSP processors is another optimization which has attracted significant research attention. Address generation units typically have address register and modify register files. Address registers are updated either through auto-increment/decrement addressing modes, or by adding the contents of a modify register. This optimization also requires a careful layout of automatic variables in the stack frame. Liao reduced this problem to one of finding the maximum weight path cover in a graph for a given (access) sequence of automatic variables [19]. Several heuristic methods as well as a genetic algorithm based approach have subsequently been proposed. In [21], transformations exploiting associativity and commutativity of operations are used to change the access order of variables (but retaining the instruction order) to obtain improved solutions. A unified approach combining instruction reordering and algebraic transformation has been proposed in [22].

Subword Parallelism and Bitwidth-Aware Compilation. Many modern processors support instruction set extensions, e.g., multimedia extensions (MMX) and visual instruction set (VIS) for supporting subword parallelism which is exposed and exploited by many techniques. Subword parallelism is exposed through vectorization methods in [16, 24]. Superword Level Parallelism packs isomorphic instructions from the same basic block as well as instructions across loop iterations by creating (short) vectors of their data in superword registers and exploits subword parallelism in the superword registers [17]. Bitwidth analysis and optimization techniques, which automatically analyze and minimize the bits used to represent operands, is useful to exploit subword parallelism, as well as reducing silicon area in custom hardware [25]. Analysis and optimization techniques to use superword registers as a compiler-controlled cache help in exploiting temporal and spatial data reuse [23]. Bitwidth analysis is efficiently employed to pack multiple subword variables in a single register in [26]. Bitwidth analysis requires both forward and backward data flow analyses.

4 Conclusions

Recent advances in processor architectures including emergence of novel architectural paradigms, application-specific processors and embedded systems, compiling applications directly onto silicon, present many interesting challenges and opportunities in high performance compilation techniques. In particular,

– These advances have put unconventional demands on traditional optimizations by creating complex situations which need to be analysed. This has influenced both machine dependent and machine independent optimizations.
– Traditionally, optimization metric was dominated by performance optimization in terms of execution time. The recent advances have enriched the optimization metrics by adding a new dimension of power/energy optimizations and by reviving the code space optimizations. Besides, newer opportunities have been created along the traditional dimension of time optimizations too.
– Data flow analysis has witnessed many advances, most notably in the kind and the range of data flow information which can be derived. Unlike traditional applications, it is now being used for machine dependent optimizations too.

We expect compiler analysis and optimization to mature still further in the days to come.

References

1. V.H. Allan and S. J. Allan. Software pipelining. In Y. N. Srikant and P. Shankar, eds., *The Compiler Design Handbook: Optimization and Machine Code Generation*. CRC Press, Boca Raton, FL, 2002.
2. R. Bodik, R. Gupta, and M.L. Soffa. Complete removal of redundant computations. In *Proc. of ACM SIGPLAN '98 Conf. on Programming Language Design and Implementation* pages 1–14, Montreal, Canada, June 1998.
3. G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, Jan. 1981.
4. V. Dhaneshwar and D.M. Dhamdhere. Strength reduction of large expressions. *Journal of Programming Languages*, **3**:95–120, 1995.
5. K. Gopinath. Register Allocation. In Y. N. Srikant and P. Shankar, eds., *The Compiler Design Handbook: Optimization and Machine Code Generation*. CRC Press, Boca Raton, FL, 2002.
6. R. Govindarajan. Instruction Scheduling. In Y. N. Srikant and P. Shankar, eds., *The Compiler Design Handbook: Optimization and Machine Code Generation*. CRC Press, Boca Raton, FL, 2002.
7. M.S. Hecht. Flow Analysis of Computer Programs. Elsevier North-Holland Inc. 1977.
8. C.-H. Hsu, U. Kremer, and M. Hsiao. Compiler-directed dynamic frequency and voltage scheduling. In *Workshop on Power-Aware Computer Systems (PACS'00)*, Cambridge, MA, Nov. 2000.
9. U.P. Khedker. Data flow analysis. In Y.N. Srikant and P. Shankar, eds., *The Compiler Design Handbook: Optimizations and Machine Code Generation*. CRC Press, Boca Raton, FL, 2002.
10. U.P. Khedker and D.M. Dhamdhere. Bidirectional data flow analysis : Myths and reality. *ACM SIGPLAN Notices*, **36**:47–57, 1999.
11. U.P. Khedker and D.M. Dhamdhere. A generalized theory of bit vector data flow analysis. *ACM Transactions on Programming Languages and Systems*, **16**:1472–1511, 1994.

12. U.P. Khedker, D.M. Dhamdhere, and A. Mycroft. Bidirectional data flow analysis for type inferencing. *Computer Languages, Systems and Structures*, **29**:15–44, 2003.
13. U. P. Khedker and S Pendharkar. Design and Implementation of VxCC. Internal Document. Tata Infotech Ltd., 2001.
14. U. P. Khedker and A. Sanyal. Improving garbage collection through static analysis. Technical report TR-CSE-004-02, Department of Computer Science and Engineering, Indian Institute of Technology, Bombay, 2002.
15. H. S. Kim, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin. Adapting instruction level parallelism for optimizing leakage in VLIW architectures. In *Proc. of ACM SIGPLAN Conf. on Language, Compiler, and Tool for Embedded Systems*, pages 275–283, San Diego, CA, June 2003.
16. A. Krall and S. Lelait. Compilation techniques for multimedia processors. *Intl. Journal of Parallel Programming*, 28(4):347-361, Aug. 2000.
17. S. Larsen and S. Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *Proc. of ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 145–156, Vancouver, Canada, June 2000.
18. R. Leupers. Code generation for embedded processors. In *Proc. of the Intl. System Synthesis Symp.*, Madrid, Spain, Sep. 2000.
19. S. Liao, S. Devadas, K. Keutzer, S. Tjiang, and A. Wang. Storage assignment to decrease code size. In *Proc. of ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 186–195, La Jolla, CA, June 1995.
20. E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Communications of ACM*, **22**:96–103, 1979.
21. A. Rao and S. Pande. Storage assignment to generate compact and efficient code on embedded DSPs. In *Proc. of ACM SIGPLAN Conf. on Programming Language Design and Implementation*, Montreal, Canada, June 1998.
22. V.V.N.S. Sarvani and R. Govindarajan. A unified instruction reordering and algebraic transformations for minimum cost offset assignment. In *Proceedings of the Intl. Workshop on Software and Compilers for Embedded Systems*, Vienna, Austria, Sept. 2003.
23. J. Shin, J. Chame, M. W. Hall. Compiler-controlled caching in superword register files for multimedia extension architectures. In *Proc. of the Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT 2002)*, pages 45–55, Charlottesville, VA, Sept. 2002.
24. N. Sreraman and R. Govindarajan. A vectorizing compiler for multimedia extension. *Intl. Journal of Parallel Programming*, 28(4):363–400, Aug. 2000.
25. M. Stephenson, J. Babb, and S. Amarasinghe. Bitwidth analysis with application to silicon compilation. In *Proc. of ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 108–120, Vancouver, Canada, June 2000.
26. S. Tallam and R. Gupta. Bitwidth aware global register allocation. In *Conf. Rec. of the 30th SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 85-96, New Orleans, LA, Jan. 2003.
27. H. Yang, R. Govindarajan, G.R. Gao, G. Cai, and Z. Hu. Exploiting schedule slacks for rate-optimal power-minimum software pipelining. In *Proc. of Workshop on Compilers and Operating Systems for Low Power (COLP-2002)*, Charlottesville, Virginia, Sep. 2002.
28. W. Zhang, N. Vijaykrishnan, M. Kandemir, M. J. Irwin, D. Duarte, and Y-F. Tsai. Exploiting VLIW schedule slacks for dynamic and leakage energy reduction. In *Proc. of 34th Intl. Symp. on Microprogramming*, pages 102–113, , Austin, TX, Dec. 2001.