Software Power Optimization via Post-Link-Time Binary Rewriting*

Saumya Debray Robert Muth Scott Watterson

Abstract

It is well known that compiler optimizations can significantly reduce the energy usage of a program. However, the traditional model of compilation imposes inherent limits on the extent of code optimization possible at compile time. In particular, analyses and optimizations are typically limited to individual procedures, and hence cannot cross procedural and module boundaries as well as the boundaries between application and library code. These limitations can be overcome by carrying out additional code optimization on the object file obtained after linking has been carried out. These optimizations are complementary to those carried out by the compiler. Our experiments indicate that significant improvements in energy usage can be obtained via post-link-time code optimization, even for programs that have been subjected to extensive compile-time optimization.

1 Motivation

The amount of energy consumed during program execution is becoming an increasingly important concern for a wide spectrum of computing systems, influencing issues ranging from battery lifetime to the amount of heat generated. It is well known that the energy usage of a program can be significantly reduced via compiler optimizations (see, for example, [5, 7, 9, 14, 15]). The reason is that such optimizations have the effect of eliminating instructions that tend to be "power-hungry," such as memory operations and branch instructions, and organize other instructions in ways that reduce the amount of energy consumed. For example, optimizations such as register allocation and invariant code motion out of loops have the effect of reducing memory operations; procedure inlining and loop unrolling result in the execution of fewer branch operations; and instruction packing and scheduling have the effect of reorganizing the instruction stream so as to reduce their overall power consumption.

However, the traditional model of compilation suffers from a number of deficiencies that limit the scope and extent of these optimizations. Programs are typically compiled a procedure at a time, which means that optimizations do not cross procedure and module boundaries. While there has been some research on inter-procedural code optimization, most compilers do not currently support such optimizations; even inter-procedural optimizations, where proposed, are generally limited in scope to individual modules. Moreover, compilers do not have access to the code for library routines at compile time, and hence are unable to optimize library calls. The latter problem cannot be readily solved within the traditional compilation model, for three reasons: first, library code is not available until link time; second, library routines often contain hand-written assembly code that cannot be processed by the compiler; and finally, even if we could modify the compilation process to work with libraries, the source code for third-party libraries may not be available. This is especially problematic because the trend towards accelerating product development cycles encourages modern software engineering techniques that aim at the use of components or code libraries that are developed with reusability and generality in mind.

These problems can be addressed by carrying out post-link-time code optimization on binaries, i.e., executable machine code files. The issues that arise, and the benefits that are obtained, in this context are complementary to those for compilers for high-level languages. This paper describes our experiences with link-time code optimization, with an eye towards energy usage reduction, using *alto*, a link-time optimizer we have developed for the Compaq Alpha architecture. The main contributions of this paper are to show that: (1) the overheads remaining in code after compile-time optimizations—even for programs that have been extensively optimized (in our experiments, at level `-O4`) by high-quality compilers—can be quite substantial; (2) much of this overhead can be eliminated by subsequent post-link-time optimization of the executable binaries; and (3) this can yield significant overall improvements in performance.

2 Link-Time Code Optimization and *alto*

The issues involved in "parsing" (i.e., decompiling), analyzing, and optimizing an executable file are very different from the corresponding source level actions in a compiler.

*This work was supported in part by the National Science Foundation under grants CCR-0073394, EIA-0080123, and ASC-9720738.
Authors' addresses: S. Debray and S. Watterson: Department of Computer Science, The University of Arizona, Tucson, AZ 85721; e-mail: `{debray, saw}@cs.arizona.edu`; R. Muth: Alpha Development Group, Compaq Computer Corporation, Shrewsbury, MA 01749; email: `Robert.Muth@compaq.com`.

Binary files typically have much less semantic information than source programs, which makes it harder to recover information about the program. As a result, tasks that are straightforward at the source level—e.g., determining the targets of a *switch* statement (i.e., an indirect jump through memory)—can turn out to be very difficult at the machine code level. In general it is also difficult to distinguish between code and data, and between addresses and non-address constants (the latter distinction is essential because code addresses must be updated to reflect the results of optimizations, while the values of constants cannot be changed).[1] Moreover, executable files can contain code generated from hand-written assembly routines that violate high-level invariants assumed by the compiler. For example, our experience with the standard C and mathematical libraries on a number of different platforms indicates that it is not uncommon for control to branch from one function into the middle of another, or fall through from one function into another, instead of using a function call. This kind of code complicates program analysis and optimization considerably. On the other hand, it is easier to carry out various low-level cost-benefit analyses, which are necessary to determine the profitability of optimizations, at the level of binaries than at the source level. Another benefit of working with binaries is that the entire program is available for analysis and optimization.

We have implemented a link-time optimizer, called *alto* [11], for the Compaq Alpha architecture; this system is freely available at `www.cs.arizona.edu/alto`. While a detailed description of *alto* is beyond the scope of this paper, we give a brief overview of the system to provide context for the discussion that follows. The execution of *alto* consists of five phases. First, an executable file is read in, and an inter-procedural control flow graph is constructed. A suite of analyses and optimizations is then applied iteratively to the program. This is followed by a function inlining phase. The fourth phase repeats the optimizations carried out in the second phase on the resulting code. The final phase carries out profile-directed code layout [12], code alignment and instruction scheduling, after which the code is written out.

3 Optimizations

There are two main classes of optimizations within *alto* that lead to energy usage improvements: memory operation elimination and value-based code specialization. This section discusses these in more detail.

3.1 Memory Operation Elimination

A number of optimizations contribute to the reduction of memory operations. These include *procedure call optimization*, *constant computation optimization*, and *load/store forwarding* (in addition, other classical optimizations, such as invariant code motion out of loops and shrink-wrapping, are also enabled by link-time code transformations and are carried out by *alto*, and permit the reduction of the number of memory operations executed beyond what is achievable at compile time).

3.1.1 Procedure Call Optimization

On most architectures, a procedure call can be implemented in two ways: we can use a PC-relative "branch-to-subroutine" (`bsr`) instruction, or we can load the address of the callee into a register and then use an indirect jump through the register using a "jump-to-subroutine" (`jsr`) instruction.[2] The former is more efficient—it does not have to load the target address from memory in order to effect the branch—but can reach only a limited number of addresses from the call site; the latter can reach any address, but is more expensive.

Compilers typically process programs a procedure at a time. Moreover, compilers do not know the order in which different object files will be linked together to produce the final executable for a program. For these reasons, when generating code for a procedure call, a compiler typically does not know how far away the callee is from the call site. This means that the compiler cannot, in general, assume that the call can be realized using a PC-relative `bsr` instruction, and therefore is forced to use a more expensive `jsr` instruction. The result is that procedure calls incur extra load operations because of the need to load the callee's address from memory.

During link-time optimization, *alto* uses global and inter-procedural constant propagation to compute the target address for every `jsr` instruction. If this address can be guaranteed to be a fixed constant that is within the range of a `bsr` instruction, the indirect procedure call can be replaced by a more efficient direct call. (The replacement of the indirect procedure call by a direct call also makes it possible to have a more precise representation of interprocedural control flow in the program, which turns out to be very helpful for assisting other optimizations.) This transformation is assisted by profile-directed code layout [12], which causes frequently executed code fragments to be placed close together in memory. While the primary benefit of such code layout is to improve instruction cache utilization, it has the effect of placing functions close to their frequently executed call sites. This has the result that the distance between the call site and the callee's entry point typically come close enough to allow the use of the cheaper `bsr` instruction.

[1] *Alto* uses relocation information, which must be supplied by the linker, to distinguish addresses from data. This is done by invoking the linker with additional options that instruct it to retain relocation information in the executable file.

[2] On a CISC architecture such as the Pentium, we can also provide the address of the callee as an immediate operand of a `call` instruction: this is essentially equivalent to the `jsr` instruction in the sense that we have to load the address of the target from memory before we can branch to it.

An example of the utility of this optimization is in the Mediabench benchmark *mpeg2_decode*: one of the most frequently executed loops in this program, in the function `Reference_ICDT()`, contains a call to the library routine `floor()`. Optimizing this call from a `jsr` to a `bsr` instruction, as described above, removes a load instruction from one of the busiest loops in the program, and has a significant impact on performance.

3.1.2 Constant Computation Optimization

In the Alpha architecture, 64-bit constants are placed in data areas called *global address tables*, which are accessed via a *global pointer* register `$gp`. Accessing a global object involves two steps: first, the address of the object is loaded from the global address table; this is then used to access the object referred to, e.g., to load from or store to a global variable, or jump to a procedure.

If it is possible to determine, from constant propagation/folding, that a value being computed or loaded into a register is a constant, *alto* attempts to find a cheaper instruction to compute the constant into that register. This is done as follows. Suppose that a register r_0 is known to contain a constant a_0 at a program point where the program loads a constant a_1 into a register r_1. The code generated by the compiler will load these constants from the global address table. However, if the difference between a_0 and a_1 is not too big (fits within 16 bits), *alto* can eliminate the load from memory into r_1, and instead use arithmetic to compute the value a_1 from the value a_0 in r_0. The primary benefit of this optimization is to significantly reduce the number of loads from the global address table, replacing them by register operations that are significantly cheaper in terms of both time and energy usage.

Alto also tries to optimize the use of constants. Most integer arithmetic instructions on the Alpha processor allow one of the operands to be a small immediate constant rather than a register. *Alto* attempts to exploit this feature: whenever it can guarantee that a register contains a small constant, *alto* attempts to modify arithmetic instructions involving that register to use an immediate value instead. This has the benefit that it frees up the register holding the constant; this register can now be used for other optimizations, such as the load/store forwarding optimization discussed below. The applicability of this optimization arises from the fact that, since at link time *alto* has access to the entire program, it is able to propagate constant arguments from procedure calls into the callee.

3.1.3 Load/Store Forwarding

It is sometimes possible to identify load (and, less frequently, store) operations as unnecessary at link time, and eliminate such operations. Unnecessary loads and stores can arise for a variety of reasons: a variable may not have been kept in a register by the compiler because it is a global, or because the compiler was unable to resolve aliasing adequately, or because there were not enough free registers available to the compiler. At link time, accesses to globals from different modules become evident, making it possible to keep them in registers. *Alto* is also able to take advantage of inlining across module boundaries, and of library routines, as well as value-based code specialization (see Section 3.2), to resolve aliasing beyond what can be done at compile time. Finally, *alto* scavenges registers–e.g., as discussed in Section 3.1.2, and also as a result of other optimizations such as the elimination of code that becomes unreachable due to the propagation of constant arguments across function boundaries—and these registers can then be used to hold values that were spilled to memory by the compiler.

Suppose that an instruction I_1 stores register r_1 to memory location l (or loads from l), and is followed soon after by an instruction I_2 that loads from location l into register r_2. If it can be shown that that location l is not modified between these two instructions, then *load forwarding* attempts to delete instruction I_2 and replace it with a register move from r_1 to r_2. It may happen that register r_1 is overwritten between instructions I_1 and I_2: in this case, *alto* tries to find a free register r_3 that can be used to hold the value in r_1. If the instruction I_1 can now be shown to be dead, it can also be deleted.

3.2 Value-Based Code Specialization

Since *alto* is able to modify executable files, in addition to carrying out optimizations, it is also able to insert instrumentation code to gather low-level execution profiles. One form of profiling it is able to carry out is *value profiling*. A value profile for a variable at a particular program point is information about the distribution of the values taken on by that variable at that point. Since gathering value profiles is a relatively expensive operation, we use a cost-benefit analysis, guided by the execution profile of the program and a cost model for the architecture under consideration, to prune the set of candidate variable/program point pairs to those where value profiling may be profitable [16].

Suppose that the value profile for a variable x at a particular program point indicates that it takes on the value 0 most of the time. If knowing that x has the value 0 allows the code to be simplified significantly, *alto* transforms the program so that it tests whether $x = 0$ at that point and branches to an optimized version of the code, specialized to the value 0 for x, if this is the case. The effect of this optimization is to allow optimized code for common-case values without sacrificing correctness for other values [10]. A low-level cost model is used to weigh the costs and benefits of such value-based code specialization and guide the decision about what code fragments are worth

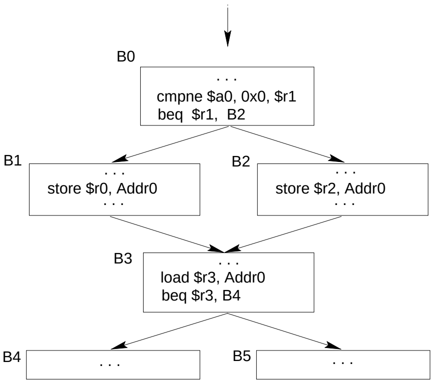

Figure 1: Example control flow graph

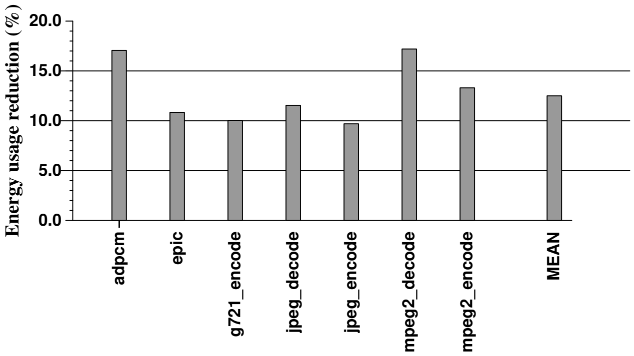

Figure 2: Energy usage reduction

specializing, and for which values. Note that this optimization is not possible using existing compiler technology because compilers do not specialize code for a value unless that value is guaranteed to be a fixed compile-time constant.

We can generalize this idea further and allow arbitrary expressions to be profiled. For example, given a pair of pointers p and q that are used for indirect loads and stores in a frequently executed code fragment, we can profile the expression '$p \neq q$' to determine whether these pointers are aliases in the common case.[3] We can then generate optimized code for situations where two pointers are unaliased most of the time, though possibly not all of the time; moreover, we can do this without having to resort to expensive and potentially conservative compile-time alias analysis. On the SPEC-95 benchmark *m88ksim*, such specialization yields a speed improvement of over 13% [10].

Overall, the effect of such code specialization is to eliminate memory and branch operations—which typically have relatively high latencies and consume a lot of energy—along commonly executed execution paths, thereby improving execution speed. On *m88ksim*, for example, we see a 17% reduction in the number of memory operations and a 5% reduction in the number of branch operations.

3.3 Interactions Between Optimizations

Much of the benefits of link-time code optimization are obtained from synergistic interactions between different optimizations. This is illustrated in Figure 1, which shows the control flow graph of a simple function. The function shown checks whether the first argument, passed to it in register `$a0`, is NULL, and branches to block B2 if it is (if `$a0` has a non-zero value, the `cmpne` instruction sets register `$r1` to 1; in this case the conditional branch `beq`—which branches to B2 if the operand register `$r1` is equal to 0—is not taken).

Suppose that all of the call sites for this function pass a non-NULL argument to this function. Context-sensitive inter-procedural constant propagation is then able to infer, from these argument values, that register `$r1` always has the value 1 after the `cmpne` instruction in block B0, and hence that the conditional branch in B0 is never taken. The optimizer accordingly deletes the `cmpne` and `beq` instructions at the end of B0 and eliminates the edge B0 $\rightarrow$ B2 from the control flow graph.

When unreachable code elimination is applied to the resulting graph, it identifies block B2 as unreachable, deletes it from the control flow graph, and adjusts the graph accordingly.

Suppose that the location `Addr0` is not modified between the `store` instruction in block B1 and the `load` in B3: load/store forwarding (Section 3.1.3) is then able to eliminate the `load` instruction in B3, replacing it by a register move from `$r0` to `$r3`.

If there are no other references to the location `Addr0` in the function, and it can be shown that this location is not live after the return from the function (e.g., if `Addr0` is a variable that is local to this function), then the `store` instruction in block B1 can also be eliminated. If the contents of register `$r0` at the `store` instruction in block B1 are known, it also becomes possible to statically determine the outcome of the conditional branch at the end of block B3, and eliminate this branch instruction as well.

4 Experimental Results

To evaluate the improvements resulting from link-time optimization using *alto*, we used a set of seven benchmarks from the Mediabench suite: *adpcm*, *epic*, *g721_encode*, *jpeg_encode*, *jpeg_decode* *mpeg2_decode*, and *mpeg2_encode*. The low-level performance characteristics of these programs were measured using the Wattch

[3] Our implementation also takes into account the width of the memory references via p and q to determine whether the addresses so referenced can overlap.

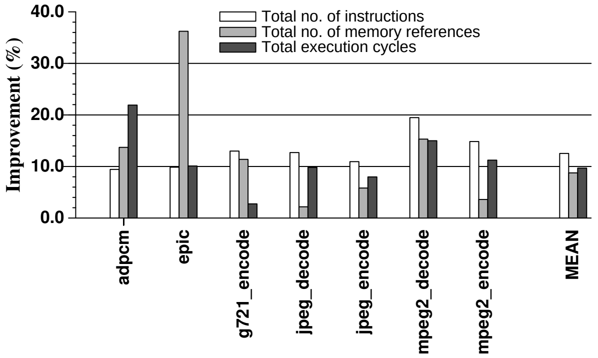

Figure 3: Low-level runtime performance characteristics

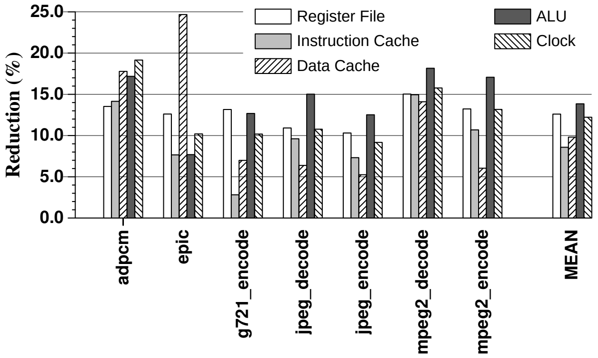

Figure 4: Breakdown of energy usage improvements

simulator [2], configured to simulate an Alpha 21264 processor [8]. Validation studies indicate that the simulation results are quite close to published energy usage data for this processor [2].

To ensure a balanced and reasonable evaluation of the benefits resulting from link-time optimization, we should ensure that the programs under consideration are subjected to as much compiler optimization as possible, preferably using a compiler with good optimization capabilities. To this end, we compiled our programs with the vendor-supplied C compiler (the highly optimizing GEM compiler system [1], which generates better code than current versions of `gcc`) V5.2-036, invoked as `cc -O4`, with additional linker options to retain relocation information and produce statically linked executables.[4]

The effect of link-time optimization on the energy usage of our programs are shown in Figure 2. The data shown refer to aggressive but non-ideal conditional clocking (i.e., some small amount of power is still consumed when a portion of the processor is not clocked), corresponding to the implementation of the Alpha 21264 processor [6]. It can be seen that most of the programs tested experience energy usage reductions in excess of 10%, with *adpcm* and *mpeg2_decode* obtaining improvements of over 15%. The smallest gain is for *jpeg_encode*, which improves by 9.7%. On average, the energy usage of this set of programs is reduced by about 12.5%.

Figure 3 shows the effect of link-time code optimization on several aspects of the low-level behavior of the programs. The total number of instructions executed by the programs generally drops by about 10–18%. Particularly significant is the effect of link-time optimization on the number of memory references (i.e., load and store instructions): several of our programs (*adpcm*, *epic*, *g721_encode*, *mpeg2_decode*) experience memory operation reductions in excess of 10%, with *epic* achieving the largest improvement of over 36%. These reductions lead to significant improvements in the total number of execution cycles as well, with reductions in the 10–20% range for several programs. Overall, we get an average reduction of 12.5% in the total number of instructions, 8.8% in the number of memory operations, and 9.7% in the number of cycles.

[4] We use statically linked executables because *alto* relies on the presence of relocation information for its control flow analysis. The Digital Unix linker `ld` refuses to retain relocation information for non-statically-linked executables.

Figure 4 gives a detailed breakdown of the effects of link-time optimization on different components of the processor during program execution. The energy usage of the register file is reduced by 10–15% (average: 12.6%). The instruction cache sees an improvement of 7–10% for most programs (average: 8.6%). Data cache energy usage is reduced by 5–25% (average: 9.8%). The energy usage of the ALU is reduced by 7–18% (average: 13.8%), and clock energy by 9–19% (average: 12.2%). The synergistic effects of the the various optimizations carried out by *alto* can be seen, for example, by focusing on the improvements in cache behavior. First, the elimination of memory (as well as other types of) instructions translates to a reduction in the number of instruction cache accesses, with a concomitant reduction in its energy consumption. Second, the elimination of load operations by keeping data values in registers, or computing them via register operations, means that less energy is used in the data cache. Finally, both these phenomena have a beneficial effect on the level-2 unified instruction and data cache. The reductions in misses in the primary instruction and data caches mean that there are fewer L2 cache accesses, and since there are fewer instructions and data values competing for space in the L2 cache there are fewer L2 cache misses as well. In either case, the result is a decrease in the L2 cache energy usage: detailed figures for this are omitted from Figure 4 due to space constraints, but we find that on average there is a 10.2% improvement in the energy usage of the level-2 cache.

It is important to note that these performance improvements are achieved on executables generated using a very high degree of compile-time optimization (at level

-O4). Moreover, these performance benefits generally result from information that is unavailable at compile time, namely, addresses of globals and functions, and by the propagation of information across boundaries that are typically not crossed in the traditional model of compilation, e.g., across procedure and module boundaries, and between application and library code. This indicates that, even though the compiler has done everything it can to eliminate inefficiencies in the code at compile time, there are nevertheless a considerable amount of overheads that remain. Link-time optimization can be used to eliminate some of these residual overheads, and this can lead to significant performance improvements overall.

5 Related Work

There has been a great deal of work on the application of compile-time code optimization to reduce the energy usage of programs [5, 9, 14, 15]. Most of these works investigate the effects of specific individual optimizations on energy usage. Our work, by contrast, examines the synergies and effects of a suite of different optimizations on energy usage.

Like us, Kandemir *et al.* also examine the combined effect of different optimizations [7]. However, their work is focused purely on compile-time optimization, while we focus on the additional energy savings that can be achieved via link-time optimization after the compiler has already optimized the program as much as it can. Other authors have looked into link-time code optimization [4, 13], but have not investigated the effects of such optimizations on energy usage.

The application of value-profile-based specialization to energy optimization is also discussed by Chung *et al.* [3]. This work carries out specialization at the source level at the granularity of procedures, while we use a low-level cost model—which can be more sensitive to machine-level cost/benefit tradeoffs—to guide specialization at the machine code level, and can specialize smaller code regions, e.g., basic blocks or loops, within procedures. We are also able to carry out expression-based specialization, which allows us to optimize code based on more general properties than simply the values of variables, e.g., non-aliasing between pointers.

6 Conclusions

While compiler optimizations can be very useful in reducing the energy usage of programs, the traditional model of compilation imposes inherent limits on the extent of optimization that can be achieved at compile time. This paper shows that the residual overheads, which can be quite substantial, can be greatly reduced via post-link-time code optimization applied to executable files. We describe our experiences with *alto*, a link-time optimizer we have developed for the Compaq Alpha architecture. Experiments with a collection of programs from the Mediabench suite indicate that energy usage reductions of over 10%, on the average, can be achieved even for programs that have been heavily optimized by a high-quality compiler.

References

[1] D. Blickstein *et al.*, "The GEM Optimizing Compiler System", *Digital Technical Journal*, 4(4):121–136.

[2] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A Framework for Architectural-Level Power Analysis and Optimizations", *Proc. 27th. International Symposium on Computer Architecture*, June 2000, pp. 83–94.

[3] E.-Y. Chung, L. Benini, and G. De Micheli, "Energy Efficient Source Code Transformation based on Value Profiling", *Proc. International Workshop on Compilers and Operating Systems for Low Power*, Philadelphia, Oct. 2000.

[4] R. Cohn, D. Goodwin, P. G. Lowney, and N. Rubin, "Optimizing Alpha Executables on Windows NT with Spike", *Digital Technical Journal* vol. 9 no. 4, 1997, pp. 3–20.

[5] G. C. S. de Araújo, *Code Generation Algorithms for Digital Signal Processors*, PhD. Dissertation, Princeton University, June 1997.

[6] M. K. Gowan, L. L. Biro, and D. B. Jackson, "Power Considerations in the Design of the Alpha 21264 Microprocessor", *Proc. 35th. ACM Conference on Design Automation* (DAC '98), June 1998, pp. 726–731.

[7] M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and W. Ye, "Influence of Compiler Optimizations on System Power", *Proc. 37th. ACM Conference on Design Automation* (DAC '00), pp. 304–307.

[8] R. E. Kessler, "The Alpha 21264 Microprocessor", *IEEE Micro*, Vol. 19, No. 2, March/April 1999.

[9] M. T.-C. Lee and V. Tiwari, "A Memory Allocation Technique for Low-Energy Embedded DSP Software", *Proc. 1995 IEEE Symposium on Low Power Electronics*, Oct. 1995.

[10] R. Muth, S. Watterson, and S. K. Debray, "Code Specialization based on Value Profiles", *Proc. 7th. International Static Analysis Symposium* (SAS 2000), June 2000, pp. 340–359. Springer LNCS vol. 1824.

[11] R. Muth, S. K. Debray, S. Watterson, and K. De Bosschere, "alto : A Link-Time Optimizer for the Compaq Alpha", *Software Practice and Experience* **31**:67–101, Jan. 2001.

[12] K. Pettis and R. C. Hansen, "Profile-Guided Code Positioning", *Proc. SIGPLAN '90 Conference on Programming Language Design and Implementation*, June 1990, pp. 16–27.

[13] A. Srivastava and D. W. Wall, "A Practical System for Intermodule Code Optimization at Link-Time", *Journal of Programming Languages*, pp. 1–18, March 1993.

[14] V. Tiwari, S. Malik and A. Wolfe, "Compilation Techniques for Low Energy: An Overview", *Proc. IEEE 1994 Symposium on Low Power Electronics*, Oct. 1994.

[15] S. Udayanarayanan and C. Chakrabarti, "Energy-efficient Code Generation for DSP56000 Family", *Proc. International Symposium on Low Power Electronics and Design* (ISLPED), July 2000, pp. 247–249.

[16] S. A. Watterson and S. K. Debray, "Goal-Directed Value Profiling", *Proc. 2001 International Conference on Compiler Construction* (CC 2001), April 2001 (to appear).