# Global Variable Promotion: Using Registers to Reduce Cache Power Dissipation

Andrea G. M. Cilio[1] and Henk Corporaal[2]

[1] Delft University of Technology, Computer Engineering Dept.
Mekelweg 4, 2628CD Delft, The Netherlands
A.Cilio@et.tudelft.nl
[2] IMEC, DESICS division
Leuven, Belgium
H.Corporaal@et.tudelft.nl

**Abstract.** Global variable promotion, i.e. allocating unaliased globals to registers, can significantly reduce the number of memory operations. This results in reduced cache activity and less power consumption. The purpose of this paper is to evaluate global variable promotion in the context of ILP scheduling and estimate its potential as a software technique for reducing cache power consumption. We measured the frequency and distribution of accesses to global variables and found that few registers are sufficient to replace the most frequently referenced variables and capture most of the benefits. In our tests, up to 22% of memory operations are removed. Four registers, for example, are sufficient to reduce the energy-delay product by 7 to 26%. Our results suggest that global variable promotion should be included as a standard optimization technique in power-conscious compilers.

## 1 Introduction

Certain code optimizations, like register allocation, offer increased potential for code improvement when applied to whole programs. Several research works, some of which resulting in a production compiler [15], have explored the potential of inter-module register allocation and *global variable promotion*. The latter technique allocates global variables in registers for a part of the lifetime crossing procedure and module boundaries (possibly for the entire lifetime). These works have always considered execution time the primary metric of evaluation. However, as we show in this paper, in the context of instruction scheduling for ILP processors performance is not so sensitive to inter-module register allocation; in this context, earlier results do not apply anymore.. With the increasing importance of low-power designs, due to the rapidly growing portable electronics market, we believe that metrics like energy and energy-delay product should be used to evaluate these and other software techniques.

From the point of view of execution cycle count, reserving a register to a global variable throughout the program lifetime is advantageous when the target architecture offers enough registers with respect to the number of interfering live

R. N. Horspool (Ed.): CC 2002, LNCS 2304, pp. 247–261, 2002.

ranges, which may be limited by, e.g., the lack of instruction-level parallelism. In these situations, a number of registers may be left underutilized. Modern multimedia, general-purpose and DSP processors, like Trimedia TM1000 [7], Intel's IA-64 and Analog Devices' ADSP-TS001M, offer large register files. Although this large number of registers is necessary to sustain high levels of ILP, the compiler ILP-enhancing techniques may not always succeed in utilizing them all effectively. By assigning underutilized registers to global scalar variables, the compiler can eliminate all the load and store operations that access those variables, thereby reducing the dynamic operation count and the cache-processor traffic. From the point of view of power consumption, this is advantageous, because a large fraction of the overall power consumption in modern processors is due to cache activity [12].

The purpose of this paper is to evaluate global variable promotion in the context of instruction-level parallel (ILP) scheduling and to estimate its potential as a software technique for reducing cache power consumption. Also, we investigate possible trade-offs points between execution time and energy consumption for different caches and CPU configurations with varying degrees of ILP.

The rest of this paper is organized as follows. Section 2 analyses the potential of global variable promotion and inter-module register allocation and presents the algorithm used to promote global scalar variables. Using the power dissipation model presented in section 3, section 4 evaluates the effect of global variable promotion on performance and two energy-related metrics. Section 5 reviews related work. Finally, section 6 summarizes the results obtained.

## 2 Global Register Allocation

A number of code generation systems extend the program analyses and optimizations to the inter-module (or whole-program) scope. Among these optimizations, inter-module register allocation and global variable promotion have received some attention [18] [17] [2] [15]. In this section we first evaluate the potential of these two optimizations techniques on our compiler. After concluding that only global variable promotion seems promising, we present an algorithm for global variable promotion.

### 2.1 Potential of Inter-module Register Allocation and Global Variable Promotion

Inter-module Register Allocation (and its restricted inter-procedural variant) aims at reducing the execution overhead due to save and restore code around function calls. While this can be effective when compiling for languages with frequent function calls, like LISP [17], the potential measured in other works, even though using more sophisticated approaches, seems low for languages like C and Pascal; the speedup ranges from 1 to 3% [2] [15].

To verify that the potential of inter-module register allocation is scarce in our C compiler (based on *gcc*), we performed a number of tests with and without

**Table 1.** Effect of function inlining on a set of benchmarks

| benchmark | % call operations original | % call operations inline | % size increase | % cycles reduction |
|---|---|---|---|---|
| compress | 0.368 | 0.005 | 18.143 | 1.557 |
| cjpeg | 1.263 | 0.063 | 6.682 | 11.306 |
| djpeg | 0.209 | 0.018 | 3.089 | 2.119 |
| mpeg2dec | 1.395 | 0.132 | 24.472 | 30.802 |
| average | 0.809 | 0.055 | 13.097 | 11.446 |

**Table 2.** Potential speedup of inter-module allocation of local variables: upper bounds

| benchmark | % reduction cycles original | cycles inline | mops original | mops inline |
|---|---|---|---|---|
| compress | 2.305 | 0.010 | 6.499 | 0.044 |
| cjpeg | 1.124 | 0.748 | 8.919 | 3.906 |
| djpeg | 0.334 | 0.198 | 2.844 | 0.721 |
| mpeg2dec | 15.731 | 1.809 | 36.441 | 4.571 |
| average | 4.873 | 0.691 | 13.676 | 2.311 |

function inlining (see table 1). Details about the target machine can be found in section 4.3, while the benchmarks are presented in section 4.2. Columns 4 and 5 of Table 1 show, respectively, the code size increase and the speedup of the inlined program with respect to the original program. Function inlining drastically reduces the number of function calls at the cost of a modest code size increase. The very low fraction of call operations after inlining (column 3) suggests that save and restore code does not constitute a large overhead.

A good upper bound to the speedup that could be achieved by means of inter-module register allocation is obtained by totally disabling the generation of save and restore code around calls. The performance is correctly measured by our cycle-accurate simulator, which takes care of saving and restoring the used registers "on behalf" of the program. Columns 2 and 3 of table 2 show the speedup obtained when the original and the inlined versions of the programs are compiled without generating save/restore code, while the last two columns show the reduction in memory operations. From these data we can conclude that the potential of inter-module register allocation is negligible after function inlining has been applied. Also, notice that this upper bound is not always achievable: recursive functions, for example, still require some save and restore code. In addition to the low fraction of function calls, another reason contributes to these very low upper bounds: the caller- and callee-saved register conventions [2] are effectively used in our compiler [8] to minimize the unnecessary save and restore code for registers that are not live around a function call.

**Table 3.** Memory operations and accesses to global scalar variables as fractions of all operations and of memory operations executed, respectively

| | | % globals | |
|---|---|---|---|
| **benchmark** | **mops %** | **unscheduled** | **scheduled** |
| compress | 31.4 | 33.0 | 25.9 |
| cjpeg | 24.8 | 26.5 | 16.0 |
| djpeg | 22.8 | 18.5 | 8.6 |
| mpeg2dec | 31.3 | 20.4 | 12.7 |
| average | 25.58 | 24.6 | 15.8 |

Promoting global scalar variables appears to be more promising than inter-module register allocation. Previous works reported speedups ranging from 7% [15] to 10–20%, for a set of small benchmarks [18] and found that global variable promotion is of greater benefit than inter-procedural register allocation. These works have also shown that scalar variable accesses represent a substantial fraction of the total number of memory operations that access global (static) data. Our measurements, however, do not fully confirm this fact, as shown in table 3. Columns 3 and 4 contain the total accesses to global scalar variables as a fraction of the total memory operations. These values have been measured in unscheduled (and only partially optimized) and scheduled code, respectively. The measured difference can be ascribed to function inlining (which is not applied to unscheduled code) and the additional optimizations performed during scheduling. The difference with previously reported results can be partially explained by the fact that, while we only count variables residing in memory, the baseline register allocator used by Wall [18] considers also constants and link-time constant addresses 'globals', and stores them in memory. These amount to a substantial portion of the overall memory references. In fact, Wall reports that the most important globals are few, frequently used numeric constants, and that keeping them in global registers captures much of the link-time allocation advantage. Since our compiler encodes all constant values (including link-time constant addresses) in immediate fields, it is not surprising that we find fewer globals.

### 2.2 Algorithm for Global Variable Promotion

The scarce potential shown by inter-module optimization, discussed in previous section, lead us to focus on variable promotion. The results reported by Santhanam [15], suggest that a simple algorithm for global variable promotion performs almost as well as the most sophisticated. For this reason, we chose *blanket* promotion, a simple algorithm which replaces a set of selected global variables with registers throughout the program.

To obtain alias information on global-scope scalar variables, we added a post-linkage analysis pass. This pass determines which variables have their address taken in at least one of the modules and are thus not eligible for promotion. All

unaliased global variables are candidates for assignment to registers. The decision of which global variable to select, given a budget of registers for promoted variables, is taken based on the number of load and store operations that would be eliminated. The frequencies are obtained with profiling. Variable promotion is applied after all modules and library functions have been linked together, before instruction scheduling [3].

## 3 Cache Power Consumption

The power dissipation due to on-chip caches is a significant portion of the overall power dissipated by a modern microprocessor. For example, the on-chip D-cache of a low-power microprocessor, the StrongARM 110, consumes 16% of its total power [12]. The current trend towards larger on-chip L1 caches emphasizes the importance of reducing their power dissipation for two reasons: first, larger caches require larger capacitances to be driven; second, larger L1 caches have higher hit rate and therefore reduce the relative power spent in L2 caches or in off-chip memory communication.

### 3.1 Cache Power Model

To evaluate the reduction of cache power dissipation we used the analytical model for cache timing and power consumption found in CACTI 2.0 [14], which is based on the cache model proposed by Wilton and Jouppi [19]. The source of power dissipation considered in this model is the charging and discharging of capacitative loads caused by signal transitions. The energy dissipated for a voltage transition $0 \rightarrow V$ or $V \rightarrow 0$ is approximated with:

$$E = \frac{1}{2} C V^2 \tag{1}$$

where $C$ is the capacitance driven. An analytical model of the cache power consumption includes the equivalent capacitance of the relevant cache components. The power consumption is estimated by combining (1) and the transition count at the inputs and outputs of each modeled component. The cache components fully modeled are: address decoder, wordline, bitline, sense amplifiers, data output driver. In addition, the address lines going off-chip and the data lines (both going off-chip and going to the CPU, are taken into account. Our model does not consider the power dissipated by comparators, data steering logic, and cache control logic.

This model is quite accurate; Kamble and Ghose [10] have shown that their model, which is very similar to this one, if coupled with exact transition counts, predicts the power dissipation of conventional caches (i.e., caches whose organization does not use power-reducing techniques like *sub-banking* and *block buffering*) with an error within 2%. In our estimations we use accurate counts for cache accesses and address bit transitions to and from memory. The average width of a piece of data written to memory is estimated assuming equal distribution of

bytes, half-words and (32-bit) words, like in [12]. Also, we estimate that the transition counts of address and data bits are evenly distributed between accesses that hit and miss the cache.

### 3.2 Energy-Related Metrics

To evaluate the efficiency of global variable promotion we measure the energy-delay (E-D) product. This metric was proposed by Gonzales and Horowitz [5], who argue it is superior to the commonly used power or energy metrics because it combines energy dissipation and performance.

To compute the delay $D$ we assumed a clock frequency compatible with the access times estimated by CACTI. The E-D product is given by:

$$ED = E \cdot D = P \cdot D^2 = P \cdot (N_{cycles} \cdot T_{clock})^2. \quad (2)$$

Although the E-D metric presents important advantages, like the reduced dependence from technology, clock speed and implementations, the energy consumption is an important metric for battery-operated processors in portable devices, because it determines their battery duration [1]. In our experiments, the energy reduction closely follows the reduction of energy-delay. Nevertheless, we do show these results.

## 4 Experimental Results

We present the results of our simulations in this section. First, we briefly introduce our code generation infrastructure, our benchmarks and the target machines used for the simulations. The results are presented in three parts: the frequency distribution of global variables, the performance results and the energy efficiency of the data cache.

### 4.1 Code Generation Infrastructure

Figure 1 shows our code generation path. It generates code for a templated architecture especially suited for Application Specific Instruction-Set Processors, called *Move*. This architecture offers explicitly programmed instruction-level parallelism, in a fashion similar to that of VLIW architectures [4]. For the purpose of this paper, the details of the architecture used for the evaluation are unimportant. Its inherently low-power characteristics, however, make the contribution of caches to the overall chip power consumption even larger than in a conventional architecture.

The code generation is coarsely split in two phases: (1) compilation to a generic-machine instruction set and (2) target-specific instruction scheduler, which integrates also register allocation [9]. Simulation of the generic, unscheduled code is used to generate profiling data. The intermediate representation used in the first phase of code generation is *SUIF*, the Stanford University Intermediate Format [6].

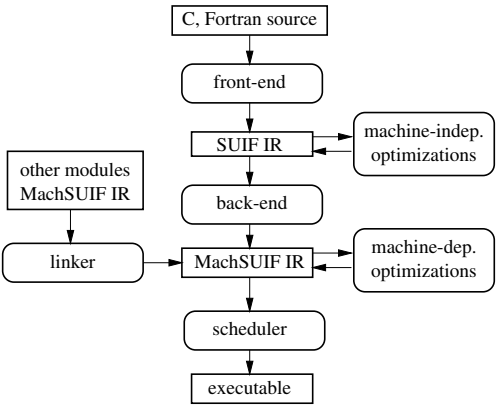


**Fig. 1.** The adapted code generation trajectory

**Table 4.** Benchmarks used for evaluation

| benchmark | instr. | cycles | description |
|---|---|---|---|
| compress | 4855 | 2.0M | Unix utility for file compression. |
| djpeg | 16421 | 19.7M | JPEG image decompression. |
| cjpeg | 16526 | 29.8M | JPEG image compression. |
| mpeg2decode | 12935 | 30.3M | Standard MPEG-2 format decoder. |

Instead of generating the traditional assembly textual output, the compiler generates and maintains a structured representation of the machine code in *MachSUIF* [16], a format derived from SUIF. MachSUIF maintains all source-level information, as well as any other piece of information gathered during analysis passes. This format allows to perform sophisticated code analysis on whole programs and makes the related code transformation much easier to apply than on a binary format [3].

### 4.2 Benchmark Characteristics

Four benchmarks have been used for our experimental evaluations. Their static code size and dynamic operation count (test set) are summarized in table 4. All benchmarks have been profiled with a training input data set and tested with a different data set. We selected multi-module programs of a sufficient level of complexity, such that the use of global (scalar) variables is almost unavoidable. Small benchmarks, on the other hand, are often coded without using global scalar variables. *Compress*, with its relatively small size, is an exception, in that it is a single-source, simple program with frequent accesses to global scalar variables.

**Table 5.** Machine configurations used for the evaluation

| | quantity | | | | quantity | |
|---|---|---|---|---|---|---|
| **resource** | **M1** | **M2** | **unit** | **latency** | **M1** | **M2** |
| transport busses | 3 | 8 | LSU | 2 | 1 | 2 |
| long immediates | 1 | 2 | IALU | 1 | 2 | 4 |
| # integer regs. | varies | varies | multiply | 3 | 1 | 1 |
| # FP regs. | 16 | 48 | divide | 8 | 1 | 1 |
| # boolean regs. | 2 | 4 | FPU | 3 | 1 | 1 |
| cache size | 16KB | 32KB | | | | |

### 4.3 Target Machines

We performed our evaluation on two Move target machines with different cost and capabilities. The two machine configurations were selected in order to evaluate how ILP affects the results of global variable promotion. Our Move architecture is kind of VLIW machine with streamlined reduced instruction set. The smaller machine, *M1*, is slightly more powerful than a simple single-issue RISC processor.[1] The average IPC measured for our benchmarks ranges between 1.2 and 1.3. We selected this configuration in order to estimate the effect of global promotion on a single-issue machine. The larger machine, *M2*, is capable to perform about 4 operations per cycle, two of which can be data memory accesses. In this case, the average IPC measured for our benchmarks is 1.7–2.3.

Table 5 summarizes the characteristics of the machine configurations. The *busses* are explicitly programmed to transport data between execution units and register files. The boolean registers allow to guard operations and predicate their execution. We assumed that the CPU is attached to a 2-way set-associative, write-through, on-chip data cache with LRU replacement policy. The cache line size is 32 bytes. Although the results shown in the following sections were obtained with 16KB and 32KB caches, other cache sizes have been tried. For all configurations the *relative* energy reduction is very similar.

### 4.4 Distribution of Global Variable Uses

The number of accesses to the memory segment dedicated to global data varies widely from benchmark to benchmark [13]. The relative frequency of memory operations that access global scalar variables poses a clear upper bound on the improvement of energy efficiency achievable via global variable promotion. Fortunately, the accesses to global scalar variables have a desirable characteristic. As shown in figure 2(a), only a few variables are sufficient to cover most memory operations due to accesses to global scalar variables. The values on the Y axis are

[1] Due to limitations in the current implementation, the integrated instruction scheduler/register allocator cannot generate code for a machine configuration with only one integer ALU.

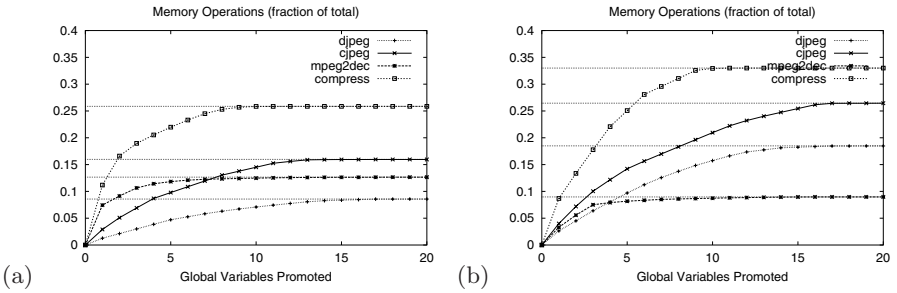


**Fig. 2.** Dynamic memory operation count covered by global scalar variables (a) on scheduled and optimized code, (b) on unscheduled code

the number of memory operations (as a fraction of the total memory operation count) due to the $N$ most used global scalar variables, where $N$ is reported on the X axis. This indicates that it is sufficient to dedicate only few registers to global variables to capture most of the benefit of global variable promotion.

The results shown in figure 2(a) refer to scheduled and highly optimized code on M1, for which most of intra-procedural unaliased accesses to global variables have been optimized away. Code optimizations considerably reduce the relative frequency of accesses to global variables, as confirmed by figure 2(b), which depicts the same frequency distribution obtained from unscheduled code. Part of this reduction is accounted for by function inlining, which opens new opportunities for intra-procedural optimizations.

### 4.5 Performance

We compiled 9 different versions of each benchmark, with a budget dedicated to global variables ranging from 0 to 8 registers. For a given register budget $n$, the $n$ most frequent global variables were promoted, resulting in the same number of registers not being available for general register allocation.

The first series of tests measures the effect of global variable promotion on performance. Figure 3 shows the cycle count of the four benchmarks for different sizes of the integer register file. The modest speedup can be explained by the fact that load operations associated with global variables have a constant address and do not have flow dependencies with preceding operations, therefore can be scheduled with considerable freedom. Thanks to this freedom, our instruction scheduler is capable of hiding the latency of most load operations associated with global variables. The results in figure 3 confirm that the effect of scheduling freedom prevails, thus making the performance improvement modest to negligible depending on the benchmark.

Figure 4 shows the dynamic count of load and store operations for the same series of tests. The reduction in memory operations executed is in good accordance with the usage distributions in figure 2, except when the most register-

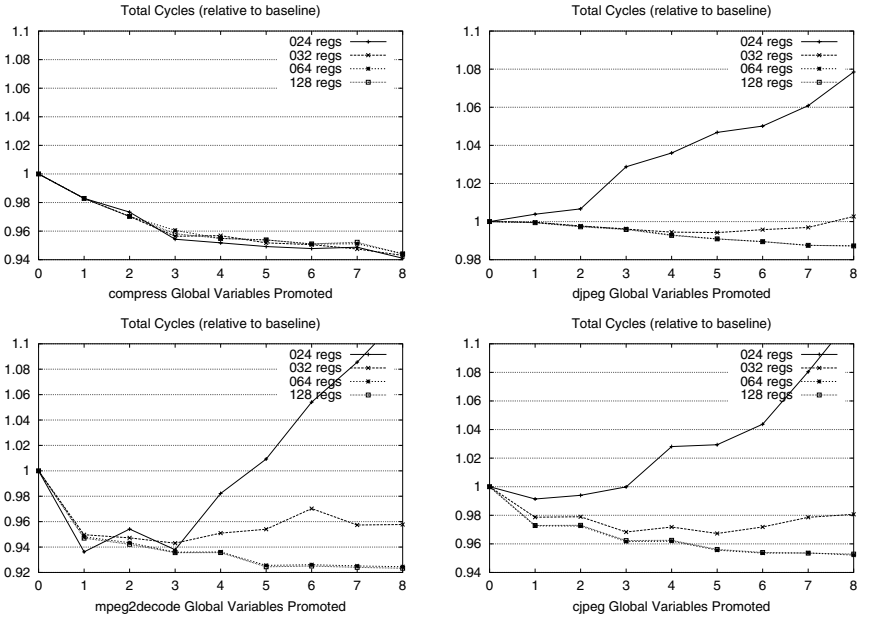


**Fig. 3.** Performance results: dynamic cycle counts on 'M1'

hungry benchmarks are run on machine configurations with a small integer register file. In such cases, the reduced number of registers available to general register allocation quickly offsets the gains of variable promotion. This is due to the introduction of false dependencies, which pose a tighter constraint on scheduling freedom. A further increase in register pressure results in a large number of spill operations.

We also measured the miss rate of the data cache and found that it increases as more global variables are promoted. Obviously, this is only a *relative* increase, due to the fact that the number of memory accesses decreases more than the number of cache misses. This result confirms that global variables show high temporal locality [13]. We can therefore conclude that global promotion reduces cache activity but does not significantly affect the CPU-memory traffic.

### 4.6 Energy and Energy-Delay Product

A reduction of energy and energy-delay product, consistent with the reduction of memory operations, has been measured for configurations of M1 and M2 with varying number of registers. Figure 5 shows the results for M1 with 64 registers relative to the original program without variable promotion. Very similar reductions are found for the M2 configuration with 64 registers, as can be seen from figure 6. In this case a 32KB cache was measured.

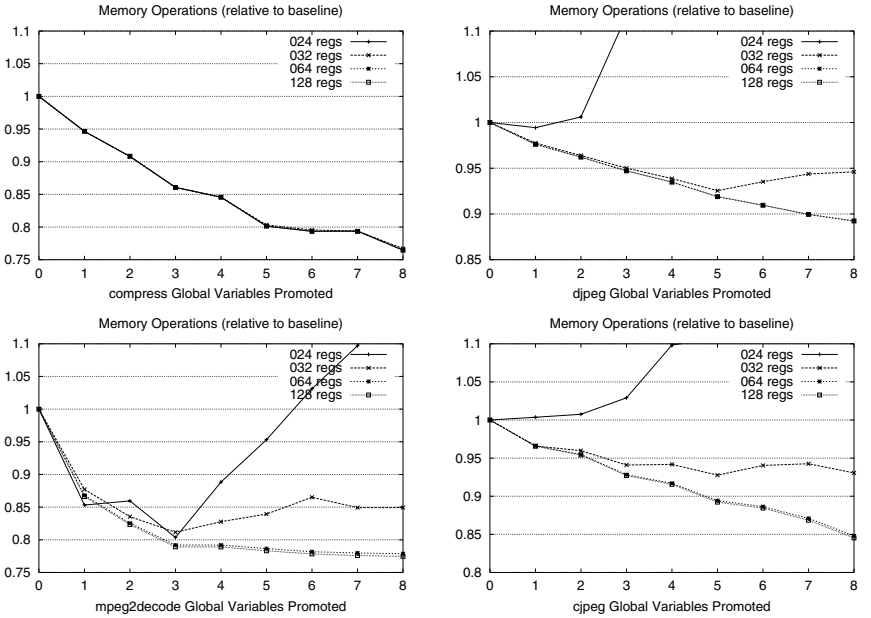


**Fig. 4.** Performance results: dynamic memory operation counts on 'M1'

While the level of ILP seems not to have a significant impact on the effect of global variable promotion, the number of available registers is critical. Figure 7 shows the energy-delay product on M1 and M2 when only 32 registers are available. Only *compress* shows consistent improvement, owing to its low register pressure; in all other benchmarks, the register pressure results in more spill code and cache activity when promoting too many globals.

The reduction in energy consumption is paired with reduced execution times, as can be seen by comparing figure 3 with figures 5, 6, and 7, therefore we cannot speak of a clear trade-off between performance and energy consumption for this software technique. This is easy to explain, since the primary source of performance degradation caused by global variable promotion is register pressure, which often results in register spilling and therefore additional memory operation and increased cache activity.

## 5 Related Work

In this section we review previous work on architectural/software techniques for reducing data cache power consumption. Work on whole-program register allocation has been briefly discussed in section 2.

It has recently been demonstrated that memory traffic due to references to the global section of a program (which includes scalar global variables) shows

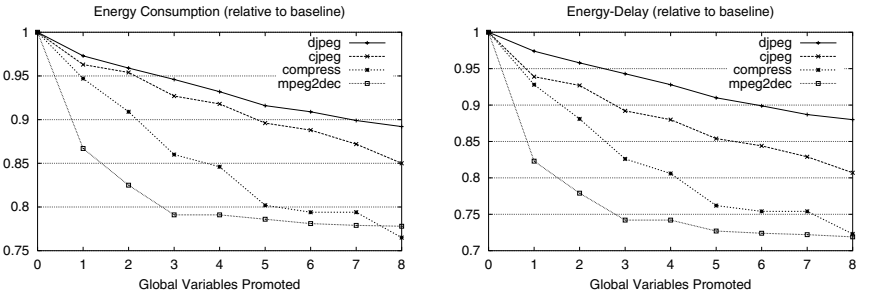


**Fig. 5.** Relative energy consumption (right) and energy-delay product (left) for a configuration of 'M1' with 64 integer registers

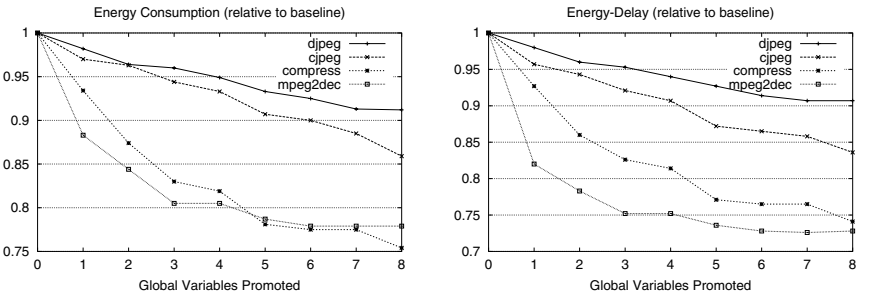


**Fig. 6.** Relative energy consumption (right) and energy-delay product (left) for a configuration of 'M2' with 64 integer registers

very high *temporal* locality, with an average life span of cache lines up to almost one order of magnitude higher than that of accesses to heap region [13]. For this reason, most traffic due to accesses to global variables can be captured by a small dedicated cache. Since stack accesses show even better cacheability, the authors subdivide the data cache into three *region caches* which cover global data, stack and heap. This three-component on-chip cache system is much more power-efficient than the conventional single data cache: 37% to 56% less power is dissipated, depending on the cache configuration.

Another recent work on architectural-level low-power cache design is presented by Kin and others [12], who propose to insert an unusually small cache before what normally is the L1 on-chip cache. This small cache, called *filter cache*, reduces the access cost by roughly a factor 6 at the cost of increased cache miss rate and increased miss latency. This allows to trade-off power efficiency with performance. The authors show that a clear optimal point exists between no filter cache at all and a filter cache of the same size of the conventional L1 cache. With an optimal filter cache size (512 bytes) the energy-delay product is reduced by 50% at the expense of a 21% increase in cycle count.

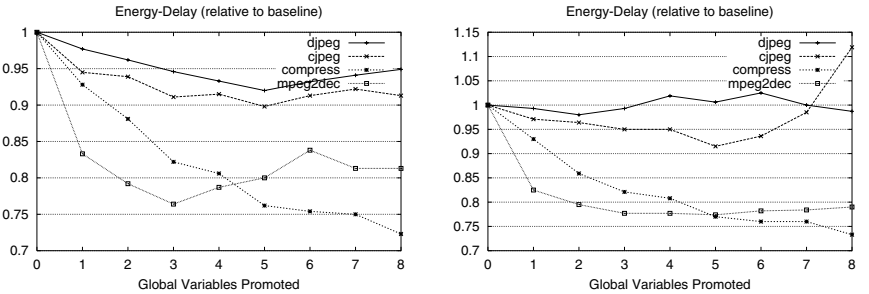


**Fig. 7.** Relative energy-delay product for a configuration of 'M1' (right) and 'M2' (left) with 32 integer registers

The use of global variable promotion to reduce power consumption proposed in this paper exploits the same principles used in the Filter Cache [12] and the Region-Based Cache [13]. While the former exploits the locality principle to decrease power consumption by introducing a new level in the memory hierarchy, our approach achieves a similar result by using the register file. The registers allocated to frequently accessed global scalar variables can be also compared to the Region-Based cache partition dedicated to global data references. In this case, the use is further limited to a selected subset of *scalar* global variables.

Many other architectural techniques for improving the energy efficiency of caches have been proposed. Kamble and Ghose, for example, evaluate the effectiveness of two such techniques: *block buffering* and *sub-banking*. The interested reader is referred to their paper [11] and to the section on previous work in [13] for further references about this important research area.

## 6 Conclusions

Power and energy consumption have become a critical issue in high-performance and portable/embedded processors, respectively. As a consequence, new microarchitectural and code generation techniques for power reduction are researched with increasing interest. At the same time, traditional software techniques, like loop unrolling take on a new light when energy-related metrics are considered [1]. Global variable promotion is in our opinion one of those software techniques that deserves new attention in the context of power reduction.

In this paper we evaluated the effect of global variable promotion on performance and cache energy consumption, and found that significant savings, up to 26%, are achieved by promoting a few (4–8) critical global variables. In summary, the results suggest that on ILP architectures the effect of global variable promotion on performance is rather limited. However, this techniques can significantly reduce data cache power consumption, and should be included as a standard optimization technique in power-conscious compilers.

## References


1. David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 83–94, Vancouver, British Columbia, June 12–14, 2000. 252, 259
2. Fred C. Chow. Minimizing register usage penalty at procedure calls. In *SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 85–94, 1988. 248, 249
3. Andrea G. M. Cilio and Henk Corporaal. A linker for effective whole-program optimizations. In *Proceedings of HPCN*, Amsterdam, The Netherlands, April 1999. 251, 253
4. Henk Corporaal. *Microprocessor Architectures; from VLIW to TTA*. John Wiley, 1997. ISBN 0-471-97157-X. 252
5. R. Gonzalez and M. Horowitz. Energy dissipation in general purpose microprocessors. *IEEE Journal of Solid-State Circuits*, 31(9):1258–66, September 1996. 252
6. Stanford Compiler Group. *The SUIF Library*. Stanford University, 1994. 252
7. Jan Hoogerbrugge. Instruction scheduling for trimedia. *Journal of Instruction-Level Parallelism*, 1(1–2), 1999. 248
8. J. Janssen. *Compilation Strategies for Transport Triggered Architectures*. PhD thesis, Delft University of Technology, 2001. 249
9. Johan Janssen and Henk Corporaal. Registers on demand: an integrated region scheduler and register allocator. In *Conference on Compiler Construction*, April 1998. 252
10. M. B. Kamble and K. Ghose. Analytical energy dissipation models for low-power caches. In *Proceedings of the 1996 international symposium on Low power electronics and design*, Monterey, CA USA, August 12–14, 1997. ACM. 251
11. M. B. Kamble and K. Ghose. Energy-efficiency of vlsi caches: a comparative study. In *Proceedings Tenth International Conference on VLSI Design*, pages 261–7. IEEE, January 1997. 259
12. Johnson Kin, Munish Gupta, and William H. Mangione-Smith. Filtering memory references to increase energy efficiency. *IEEE Transactions on Computers*, 49(1), January 2000. 248, 251, 252, 258, 259
13. Hsien-Hsien S. Lee and Gary S. Tyson. Region-based caching: An efficient memory architecture for embedded processors. In *CASES*, San Jose, CA, November 2000. 254, 256, 258, 259
14. G. Reinman and N. P. Jouppi. An integrated cache timing and power model. Technical report, COMPAQ Western Research Lab, Palo Alto, California, 1999. 251
15. Vatsa Santhanam and Daryl Odnert. Register allocation across procedure and module boundaries. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 28–39, 1990. 247, 248, 250
16. Michael D. Smith. Extending SUIF for Machine-dependent Optimizations. In *Proceedings of the First SUIF Workshop*, January 1996. 253
17. Peter A. Steenkiste and John L. Hennessy. A simple interprocedural register allocation algorithm and its effectiveness for lisp. *TOPLAS*, 11(1), 1989. 248
18. David W. Wall. Register windows vs. register allocation. Technical Report 7, Western Research Laboratory, Digital Equipment Corporation, December 1987. 248, 250

19. S. J. E. Wilton and N. P. Jouppi. An enhanced access and cycle time model. Technical Report 5, Digital Western Research laboratory, Palo Alto, California, July 1994. 251