On-chip Stack based Memory Organization for Low Power Embedded Architectures

Mahesh Mamidipaka
maheshmn@cecs.uci.edu

Nikil Dutt
dutt@cecs.uci.edu

Center for Embedded Computer Systems
University of California, Irvine, CA 92697, USA

Abstract

This paper presents a on-chip stack based memory organization that effectively reduces the energy dissipation in programmable embedded system architectures. Most embedded systems use the notion of stack for implementation of function calls. However, such stack data is stored in processor address space, typically in the main memory and accessed through caches. Our analysis of several benchmarks show that the callee saved registers and return addresses for function calls call constitute a significant portion of the total memory accesses. We propose a separate stack-based memory organization to store these registers and return addresses. Our experimental results show that effective use of such stack-based memories yield significant reductions in system power/energy, while simultaneously improving the system performance. Application of our approach to the SPECint95 and MediaBench benchmark suites show a reduction of up to 32.5% reduction in energy in L1 data cache, with marginal improvements in system performance.

1. Introduction

Current day embedded system designers are facing conflicting requirements of higher performance and lower power consumption. Power/Energy consumption in particular has received increased attention as it determines the battery life in portable battery operated devices. Also, reliability concerns and higher packaging costs for absorbing the heat dissipated in highly dense chips have made the power dissipation problem even more significant. Researchers have proposed various techniques from the algorithmic and system level[6, 15] to circuit and layout level[15, 16]. Also different dynamic voltage scaling algorithms[19, 17] and dynamic power management[13] strategies are proposed to reduce the power dissipation. While some of the power reduction techniques improve performance, some techniques degrade performance. In this paper, we propose a memory organization supporting function calls for programmable embedded systems that reduces the system level power/energy dissipation and at the same time gives marginal improvements in performance.

In the context of software development for embedded systems, Modular programming[8] has gained tremendous significance in the development of embedded system software. The efficient use of modules (called as functions) eases the tasks of understanding, re-usability of code, independent software development and assists code size reduction. However, in the context of a programmable embedded system architecture, this demands a focus towards efficient implementation of function calls. For every function call that occurs during program execution, the application saves and retrieves the context by saving a few registers before the execution of call and retrieving them once the function call execution is complete. Such register-saves contribute to a significant portion of the total memory accesses for many applications, resulting in a significant increase in the power dissipation. Since the register saves and restores can be implemented using a Last-In-First-Out (LIFO) policy, efficient stack memory organizations could be used for saving these registers. In this paper, we propose the use of a separate on-chip stack based memory organization for implementation of function calls that leads to improved performance, and more significantly, reduced power/energy consumption.

The paper is organized as follows. In Section 2, we present the motivation behind our proposal to use a separate stack memory organization for reduction in power consumption. Section 3 describes how the proposed stack memory organization can be integrated into existing programmable embedded system designs. Section 4 elaborates on the setup used for our experiments, the models used for estimation of energy dissipation, and presents results showing the efficacy of our technique in terms of reduction in energy dissipation. Related work is presented in Section 5 and Section 6 concludes with a discussion of future work.

1530-1591/03 $17.00

2. Motivation

In this section, we present different application statistics for the SPECint 95 and MediaBench benchmarks that motivate the necessity of a separate stack memory organization. First we describe the protocol typically used for the execution of a function call on a processor. Then we show some experimental results that form the basis of our proposal for a separate stack memory organization.

Although the machine code for implementing a function call depends on the source language, on the compiler, and on the architecture, compilers for most processors and popular source languages (e.g., C) generate code that follow a similar generic protocol. A typical assembly code for a function call follows the steps listed below. (Note that when function B is called from function A, A and B are referred to as caller and callee functions respectively.)

Caller initiated steps:

- Save caller-saved registers
- Pass arguments to callee
- Execute call instruction

Callee initiated steps:

- Allocate memory for local variables
- Save callee-saved registers and return address
- Execute the function call
- Retrieve the return address and callee-saved registers
- Exit from function call

Most processor architectures have designated registers for storing return address and registers that caller and callee need to save. Callee-saved registers and caller-saved registers are saved only if the callee function attempts to use the corresponding registers during its execution. Similarly, the return address register is saved only if there is a nested function call from the callee function. While the local variables and the arguments for the callee function are accessed randomly during the function execution, the callee saved-registers and the return address register are stored and retrieved only once. Also these registers can be saved and retrieved in a LIFO order. Traditionally these registers are saved in main memory accessed through different levels of caches. In this work, we propose to use a separate memory module, an on-chip stack, for storing the callee-saved registers and return address register instead of storing them in main memory. Henceforth we refer to the callee-save register and return address register stores and loads as stack accesses.

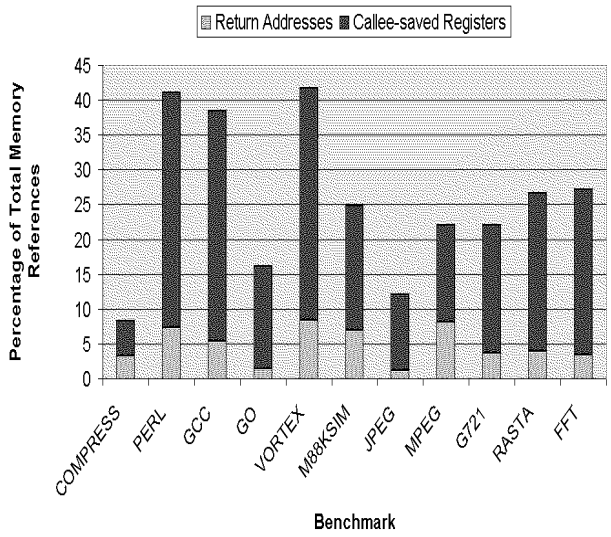

Figure 1. Plot showing percentage of stack accesses of the total memory references

Figure 1 shows the percentage of memory accesses caused due to stack accesses for several SPECint 95 and MediaBench benchmarks. The experiments were run using the sim-cache simulator within the SimpleScalar tool set [5]. The sim-cache simulator is modified to count the stack accesses and also for counting the maximum depth of call functions during the execution of an application. The SimpleScalar architecture consists of 9 callee-save registers and a designated return address register. So potentially there can be 20 stack accesses per function call (10 stores and 10 loads). However, some function calls need not have any stack accesses. Even saving of the return address can be avoided if there is no other function call from the current function. In such a case, the return address register would not be modified by the callee-function thereby eliminating the need for saving the return address register.

Table 1 shows different metrics for each benchmark [1] . Column 2 shows the benchmark program, Column 2 shows the total number of instructions executed for the corresponding benchmark, Column 3 indicates the total function calls in the benchmark execution, Column 4 indicates the maximum call depth during the execution, Column 5 indicates the total memory references in the benchmark execution, Column 6 indicates the number of return address accesses (stores+loads), and Column 7 indicates the callee-save register accesses in the program execution (stores+loads). We make the following observations based on data from Table 1 and Figure 1:

- Stack accesses (return address and saved register - reads and writes) contribute to an average of 25.5% of the total memory accesses.

[1] All values are in Millions, except for call depth values

Program	Total Instrs. (mil)	Total Calls (mil)	Call Depth	Memory Refs. (mil)	Return Addr. (mil)	Callee Reg. (mil)
compress	35.7	0.9	9	13.4	0.5	0.7
perl	2391.5	43.5	23	1044.1	77.4	354.7
gcc	1273.2	17.2	41	513.0	27.6	171.0
go	16389.6	178.2	33	4745.6	65.8	706.5
vortex	9051.6	188.7	29	4763.0	372.1	1473.5
m88ksim	493.0	7.4	13	127.3	8.9	22.9
jpeg	15.8	0.1	12	0.5	0.1	0.6
mpeg	171.2	1.5	17	32.9	2.7	4.6
g721	276.9	2.7	9	48.0	1.8	8.9
rasta	39.0	0.6	15	12.2	0.5	2.8
fft	137.2	2.3	10	31.7	1.1	7.6

Table 1. Various metrics for SPECint 95 and MediaBench for stack memory analysis

- The maximum call depth of 41 is observed for 'GCC' benchmark. Since for each call a maximum of 10 registers can be saved, the worst-case situation requires a stack of size 410 words.

It is to be noted that an access to stack (because of its smaller stack size and absence of tag match) would consume lesser power/energy than a cache access. Since as many as 42% (for VORTEX) of the total memory accesses are diverted from a high power consuming memory module to low power module, we believe that use of a stack memory organization will help reduce system level power dissipation significantly. Although the reductions are shown for the execution of various benchmarks on the SimpleScalar architecture, we believe that the same characteristics would be valid for most of the popular embedded processor architectures.

3. Stack Memory Organization

A traditional embedded processor memory architecture is shown in Figure 2. The processor core has separate instruction and data buses connecting to instruction and data caches respectively. The memory hierarchy consists of another level of unified cache (L2 Cache) before accesses are made to main memory. In this section we describe two design methods: In-processor stack and Memory-mapped stack for integrating the hardware stack into the system level design. We elaborate on each of these methods and give some critical analysis on the overheads involved in each of the proposed implementations.

In-processor stack: In this method, we propose to integrate the stack into the existing processor architecture. This necessitates two extra instructions: push and pop to store and load contents from the stack. So the compiler with the knowledge of stack hardware in the processor needs to associate the register-saves and restores to the push and pop instructions respectively. In addition to the compiler modifications, this method will require the processor pipeline to understand the new instructions. Although this might be a feasible implementation, the implementation cost would be significant considering the modifications needed for both compiler and processor hardware. However, in addition to the power savings in the L1 Data caches, this method will yield savings on the data address bus. This is because accesses to the stack within the processor requires that only the load/store signals be communicated unlike traditional cache accesses that require the whole address.

Memory-mapped stack: In this method, we propose to map the stack module to the processor data address space. In this case, the compiler needs to be modified so that it uses the address space corresponding to the stack for the register-saves and restores. The graphical view of this memory mapping is shown in Figure 3 and can be compared with the traditional implementation in Figure 2. In the memory-mapped method, the stack accesses go directly to a stack memory which shares the address space with the main memory. The other data reads/writes in the program execution go to the data cache (DCache). Note that although the stack is shown along side the main memory, the stack is still intended to be on-chip and only the address space is shared with the main memory.

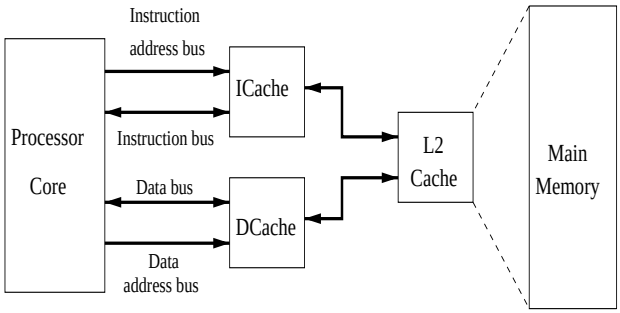

Figure 2. Traditional processor memory architecture

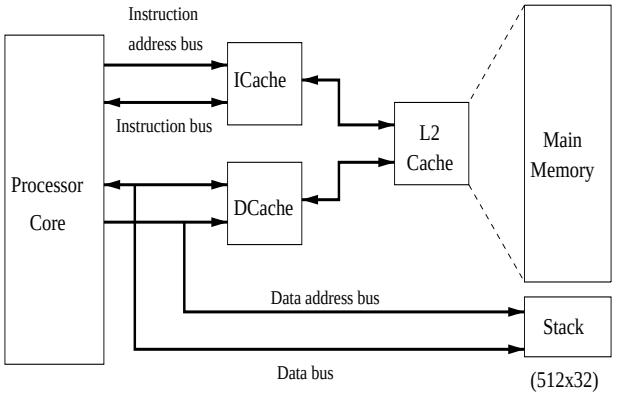

Figure 3. Stack based processor memory architecture

Table 2 shows the comparison of the design changes that would be needed for each of the proposed implementation methods. If the processor data address space is not critical,

we suggest the use of memory mapped stack implementation since it does not require any changes to the processor core.

Implementation type	Address Mapping	Compiler Changes	Processor Changes
In-processor	NO	YES	YES
Memory mapped	YES	YES	NO

Table 2. Comparison of implementation changes for proposed methodologies

In both the proposed methodologies, the critical design issue is to decide the size of the stack. In embedded system domain, since the application is often known, the maximum depth of function calls can be evaluated by running exhaustive simulations. The stack depth can then be calculated as the product of maximum function call depth and the maximum callee register saves per function call. However, in some applications, it may not be possible to determine the maximum function call depth. In such cases, an extrapolated value could be used to determine the stack size. To account for the eventuality of a stack overflow, the memory management unit could be modified to enable the saving and retrieval of stack registers from the L1 data cache when the stack is full. In our analysis, we assume that the stack size can be determined and hence the memory management unit would not incur any significant power or performance overhead.

4. Experimental Setup and Results

We consider a traditional processor memory architecture, with processor accessing memory through separate L1 instruction and data cache, and an unified L2 cache show in Figure 2 as the reference architecture. We assume that both L1 and L2 caches are on-chip. For showing the efficiency of the proposed technique we simulate the memory-mapped stack methodology. The graphical view of the proposed architecture is shown in Figure 3. This technique requires identifying the stack reads and writes at the compile phase and generating code by mapping these reads/writes to the stack address space. However, in our experiments, we use the simulator to emulate the changes required in the compiler by identifying the memory loads and stores corresponding to the stack. In our experiments we consider a 512x32 SRAM based stack. Although a more efficient and low power custom stack could be designed, for the sake of simplicity in modeling the power, we use a SRAM based stack in our experiments. The size of the stack is decided based on the maximum stack depth that can be reached during the program execution as explained in Section 2. The SPECint 95 and MediaBench benchmarks are used in the experiments to show the efficiency of our proposed technique.

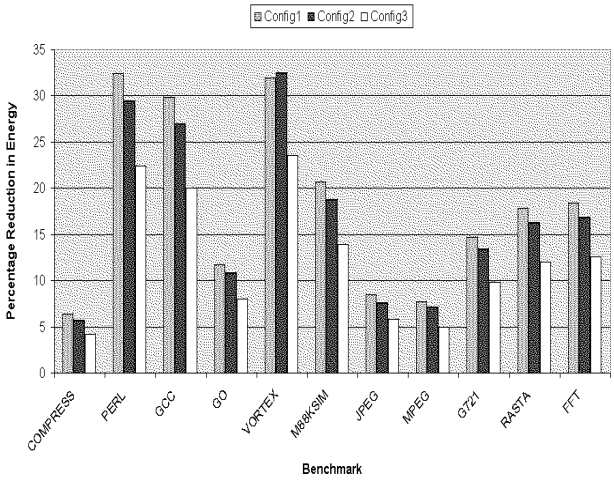

Figure 4. Plot showing % reduction in L1 data cache energy for different configurations

Note that compared to the traditional architectural implementation, the stack based architecture reduces the accesses to L1 data cache and L2 cache. The energy dissipation in the processor core would remain the same. Also using the stack based architecture, there is minimal variation in accesses to main memory. So we believe that the compared to traditional implementation, the stack based architecture will have reductions in L1 data cache and L2 cache. Therefore only these units are modeled to see the reduction in energy/power. To compute the energy/power dissipations we use the power models described in Wattch[4].

Figure 4 shows the percentage reductions in energy dissipation in data cache achieved using the stack based architecture. The percentage reduction is calculated as $\%red = 100 * [E_{DcacheTrad} - (E_{DcacheStack} + E_{Stack})]/E_{DcacheTrad}$ where, $E_{DcacheTrad}$ is the energy dissipation in data cache in traditional architecture, $E_{DcacheStack}$ is the energy dissipation in data cache in stack based architecture and E_{Stack} is the energy dissipation in stack. Since the energy dissipation in a cache depends on its size and configuration, the reductions are shown for three representative data cache configurations. Table 3 shows the size and parameters of each data cache configuration. The parameters are expressed in terms of *(No. of Lines x Block Size x Associativity x Word Size).*

	Size	Parameters
Config1	8 KB	256x32x1x1
Config2	4 KB	256x16x1x1
Config3	2 KB	128x16x1x1

Table 3. Different L1 data cache configurations used in experiments

Figure 4 shows a significant reduction in energy for all configurations of data cache for various benchmarks. It can

be observed from Figure 4 that the percentage reduction in energy for data caches using stack hardware, reduces with the size of data cache. This is because for smaller data cache sizes the energy dissipation per access would be more and more comparable to the energy dissipated in the stack per access. However, smaller sizes of data cache could increases the miss rate, requiring more accesses to L2 cache. This not only affects the performance (because the L2 cache has a higher latency) but also increases the energy dissipation in L2 cache. For VORTEX, it can be observed that percentage reduction in energy for larger data cache (config1) is less than that for a smaller data cache (config2). This is because for the smaller data cache configuration (config2) and stack, the miss rate in the cache reduced considerably compared to the implementation without a stack memory. Since a miss contributes to more dissipation in energy than a hit, for VORTEX, the percentage reductions in the cache for config2 is higher than that of config1. The maximum reduction in energy dissipation is achieved for PERL, 32.5% in config1. On average, the percentage reduction reduction in energy dissipation for data cache (config1) is 19.5% for various SPECint95 and MediaBench benchmarks. The average energy reductions in data cache for MediaBench benchmarks (JPEG, MPEG, G721, RASTA, and FFT) are seen to be less than that for SPECint95 benchmarks (COMPRESS, PERL, GO, GCC, VORTEX, and M88KSIM). This could be because of the smaller data set used for the MediaBench benchmarks which leads to reduced number of misses in data cache [2]. Nevertheless, we still observe significant energy reductions for the MediaBench benchmarks as well.

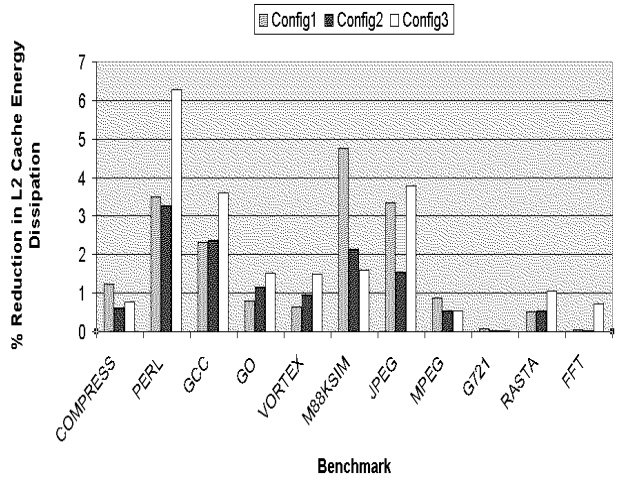

Figure 5. Plot showing % reduction in L2 cache energy for various configurations

Figure 5 shows the percentage reduction in L2 cache energy using the stack based architecture compared to the traditional architecture for the different data cache configurations listed in Table 3. As expected, the energy dissipation reductions were seen to be proportional to the reduction in accesses to L2 Cache. Because of the reduction in accesses to data cache with the use of stack hardware, the accesses to L2 cache are also reduced. However, since the accesses to L2 cache due to misses in L1 Instruction cache do not change significantly, the overall reduction in accesses to L2 cache is not significant. The percentage reduction in energy has been observed to be 1.9% on average for config1 over the benchmarks on which experiments are conducted. As stated earlier, we think the reductions in L2 cache energy for the MediaBench benchmarks have been minimal because of fewer misses at the L1 data cache.

To evaluate the total clock cycles for execution of the application in each configuration, we used a 1 cycle latency for accesses to L1 caches and stack, 6 cycle latency for accesses to L2 cache, and a 18 cycle latency for main memory accesses. However, the improvement in performance is not significant and varied from 2% to a maximum of 5%. The improvement in performance was expected to be marginal because both the stack and L1 data cache accesses are assumed to have the same latency. The improvements however are because of the reduced number of misses in L1 data cache when the stack based memory subsystem is used. Note that since stack accesses account for significant percentage of total accesses (apprx. 25.5% for SPECint 95 and MediaBench), if the stack access latency can be reduced, there would be a significant improvement in performance in addition to the reductions in energy.

5. Related work

The work related to this paper can broadly be classified under three categories: different memory organizational techniques for low power, optimizations related to function calls, and use of stacks in system designs.

The memory subsystem in programmable embedded systems has long been identified as a major contributor for the total system level power dissipation. Previous research has generated related work on several fronts, focusing mainly on various cache configurations and memory organizations for low power. Su and Despain evaluated different low power techniques like block buffering and subbanking[20]. Kin et al. proposed a filter cache[9] as another level of caching for reducing the power dissipation at the cost of performance. The Loop cache[11] is proposed to reduce the power dissipation in instruction cache. The loop cache exploits the locality in loops by storing the loop in a cache closer to the processor. The loop cache eliminates the need for tag comparison and because of its smaller size yields significant power reductions. Scratch pad memories(SPM)[14] are often used in embedded system to read/write the most often used data during program

[2]Note that a cache miss consumes more power than a hit because of an additional write in tag array and data array due to cache replacement.

execution. Effective exploitation of a SPM can improve program performance (by reducing effective data access time) and reduce energy consumption as on chip memories consume less per access energy than off chip DRAMs[3]. However, the designer still has to analyze the program to partition the data between the off-chip memories and SPMs.

Inline function expansion has been proposed in the literature for eliminating overhead involved in function call invocation and may lead to improved performance. This technique enlarges the scope of other optimizations which could lead to enhanced performance[18]. Although this technique eliminates the need for context save/restore, it could lead to increased code size. Increased code size in turn could have implications on instruction cache misses and hence the power dissipation in instruction cache.

A lot of work has been presented in the literature on stack computers in which all the computations for an application rely on stack based memory organization[10]. More recently, the instruction set corresponding to stack architectures are used for compiling Java programs on a Java Virtual Machines (JVM) because of their compact encoding[12]. In the domain of high level synthesis, hardware stacks are proposed to exploit regularity of accesses[2, 7]. Some DSPs[1] and general purpose processors have a primitive hardware stack (8-32 word deep) used for storing return addresses. The stack was meant to eliminate cache misses during the retrieval of return address. Although we too propose a hardware stack, our focus is to reduce the power dissipation by exploiting the accesses that occur in programmable processors during the execution of function calls.

6. Conclusions and Future Work

In this paper, we showed that storing and retrieving the callee-saved registers and return address registers in a separate stack memory organization yields significant power/energy reduction compared to a traditional cache based implementation. Experiments on the SPECint 95 and MediaBench benchmarks show a reduction of up to 32.5% in energy dissipation. Also, using this technique, the performance improves marginally by 2-5%. In our experiments, we considered a SRAM based stack implementation. Future work will involve development of an efficient stack implementation that exploits the stack characteristics for low power. In addition to callee-saved registers, caller-saved registers also can be stored and retrieved from the stack for further reduction in power. However, not all caller-saved registers can be stored in the stack. This is because a smart compiler can store the caller-saved register once but might want to retrieve it multiple times across different function calls. Future work will involve modifying the compiler to use the stack for storing the applicable caller-saved registers in addition to callee-saved registers and return address register. Also we plan to quantify the overhead in the memory management unit because of the stack based memory organization.

7. Acknowledgements

The authors would like to thank Prof. Tony Givargis for his valuable comments and feedback on the work. The research was partially supported by grants from NSF (CCR-0203813).

References

[1] *Texas Instruments, TMS320C5X user's guide.* 1998, http://www-s.ti.com/sc/psheets/spru066d/spru056d.pdf.

[2] M. Aloqeely. et al. A new technique for exploiting regularity in data path synthesis. In *DAC*, pages 394–399, 1994.

[3] R. Banakar. et al. Scratchpad memory : A design alternative for cache on-chip memory in embedded systems. In *CODES*, 2002.

[4] D. Brooks. et al. Wattch: a framework for architectural-level power analysis and optimizations. In *ISCA*, pages 83–94, 2000.

[5] D. Burger and T. Austin. The SimpleScalar tool set, Version 2.0. In *Computer Architecture News*, pages 13–25, 1997.

[6] A. P. Chandrakasan. et al. Minimizing power consumption in digital CMOS circuits. *Proceedings of the IEEE*, 83:498–523, 1995.

[7] S. Gerez and E. Woutersen. Assignment of storage values to sequential read-write memories. In *EDAC*, pages 302–307, 1996.

[8] D. Hanson. *C Interfaces and Implementations-Techniques for Creating Reusable Software*. Addison Wesley, 1997.

[9] J. Kin. et al. The filter cache: An energy efficient memory structure. In *Intl. Symposium on Microarchitecture*, pages 184–193, 1997.

[10] P. Koopman. *Stack computers: The new wave*. Ellis Horwood, 1989.

[11] L. Lee. et al. Instruction fetch energy reduction using loop caches for embedded applications with small tight loops. In *ISLPED*, 1999.

[12] T. Lindholm and F. Yellin. *The Java Virtual Machine specification.* Addison Wesley, April 1999.

[13] Y.-H. Lu. et al. Comparing system level power management policies. In *IEEE Design and Test of Computers*, pages 10–19, 2001.

[14] P. Panda. et al. Efficient utilization of scratchpad memory in embedded processor applications. In *EDTC*, 1997.

[15] M. Pedram. Power minimization in IC design: principles and applications. *ACM TODAES*, 1:3–56, 1996.

[16] M. Pedram. et al. Power optimization in VLSI layout: a survey. *The Journal of VLSI Signal Processing Systems*, 15:221–232, 1997.

[17] G. Qu. What is the limit of energy saving by dynamic voltage scaling? In *ICCAD*, pages 560–563, 2001.

[18] M. Serrano. Inline expansion: when and how? In *Proc. of the conference on Prog. Languages, Impl. and Logic Programming.*, 1997.

[19] T. Simunic. et al. Dynamic voltage scaling and power management for portable systems. In *DAC*, pages 524–529, 2001.

[20] C. Su and A. Despain. Cache design tradeoffs for power and performance optimization: A case study. In *ISLPED*, 1995.