uses dual-port memories to reduce the number of RAMs to three: two for input data and one for α metrics. This is the same as our dual port configuration for $\eta = 1$. The work presented in [3] describes the variations in area, throughput and memory energy for monolithic memories for different values of η. While the throughput variations are very similar to those presented here, the memory energy is significantly different – sharp increase in [3] vs. almost constant energy with increase in η in our architecture. This difference could be due to differences in data access pattern, use of monolithic memory, etc.

IV. Conclusion

In this paper, the optimal memory sub-bank structure for a MAP-based SISO decoder that achieves a very high throughput for different SW configuration was derived. A tradeoff between performance parameters such as throughput/decoding latency, and architectural parameters such as number of sub-banks, memory size, and memory energy was established.

The SW approach reduced the latency of the SISO decoder and thereby reduced the overall decoding latency of the iterative Turbo decoder. For very high throughput Turbo decoders, the interleaver/de-interleaver memory access is clearly a bottleneck. Interleaving techniques such as those proposed in [10] in conjunction with the proposed SISO decoder architecture can help realize ultra high throughput Turbo decoders.

References

[1] C. Berrou, A. Glavieux, and P. Thitimajshima, "Near Shannon limit error-correcting coding and decoding," in *Proc. Int. Conf. Commun.*, 1993, pp. 1064–1070.

[2] Z. Wang, Z. Chi, and K. K. Parhi, "Area-efficient high-speed decoding schemes for Turbo decoders," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 10, no. 6, pp. 902–912, Dec. 2002.

[3] C. Schurgers, F. Catthoor, and M. Engels, "Memory optimization of MAP Turbo decoder algorithms," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 9, no. 2, pp. 305–312, Apr. 2001.

[4] G. Masera, G. Piccinini, M. R. Roch, and M. Zamboni, "VLSI architecture for Turbo codes," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 7, no. 3, pp. 369–379, Sep. 1999.

[5] G. Masera, M. Mazza, G. Piccinini, F. Viglione, and M. Zamboni, "Architectural strategies for low-power VLSI turbo decoders," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 10, no. 3, pp. 279–285, Jun. 2002.

[6] M. M. Mansour and N. R. Shanbhag, "VLSI architecture for SISO-APP decoders," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 11, no. 4, pp. 627–650, Aug. 2003.

[7] A. J. Viterbi, "An intuitive justification and a simplified implementation of the MAP decoder for convolutional codes," *IEEE J. Select. Areas Commun.*, vol. 16, no. 2, pp. 260–264, Feb. 1998.

[8] J. Erfanian, S. Pasupathy, and G. Gulak, "Reduced complexity symbol detectors with parallel structure for ISI channels," *IEEE Trans. Commun.*, vol. 42, no. 2/3/4, pp. 1661–1671, Feb./Mar./Apr. 1994.

[9] Z. Wang, H. Suzuki, and K. Parhi, "Finite wordlength analysis and adaptivdecoding for Turbo/MAP decoders," *J. Very Large Scale Integr. (VLSI) Syst. Signal Process.*, vol. 29, pp. 209–221, 2001.

[10] Z. Wang and K. K. Parhi, "High performance, high throughput Turbo/SOVA decoder design," *IEEE Trans. Commun.*, vol. 51, no. 4, pp. 570–579, Apr. 2003.

Instruction Code Mapping for Performance Increase and Energy Reduction in Embedded Computer Systems

Sri Parameswaran and Jörg Henkel

***Abstract*—In this paper, we present a novel and fast constructive technique that relocates the instruction code in such a manner into the main memory that the cache is utilized more efficiently. The technique is applied as a preprocessing step, i.e., before the code is executed. Our technique is applicable in embedded systems where the number and characteristics of tasks running on the system is known *a priori*. The technique does not impose any computational overhead to the system. As a result of applying our technique to a variety of real-world applications we observed through simulation a significant drop of cache misses. Furthermore, the energy consumption of the whole system (CPU, caches, buses, main memory) is reduced by up to 65%. These benefits could be achieved by a slightly increased main memory size of about 13% on average.**

***Index Terms*—Cache memories, energy conservation, low power.**

I. Introduction

In embedded systems containing a memory hierarchy, the performance can often be improved by reducing the overhead of instruction cache misses. Along with that often comes also a reduction in energy consumption of the whole system: assumed, a basic system consists at least of a CPU, an instruction cache, a main memory and a bus system, then the following mechanisms can reduce the power/energy consumption of the whole system:

- lesser cache misses lead to lesser main memory accesses and thus to a reduced energy consumption of the main memory;
- reduced cache misses decreases the traffic on the bus and thus the energy/power consumption on the buses;
- lesser waiting cycles will be imposed to the CPU and such reducing the CPU energy/power consumption.

The potential savings in energy vary by system. As an example, Liu *et al.* have estimated for memories that the total power savings can be up to 70% of the total power [4].

Cache misses can be classified as follows: *compulsory misses, conflict misses*, and *capacity misses*. A compulsory miss occurs when an instruction or data is referenced for the first time, a conflict miss occurs when data or instruction compete for the same cache location, and a capacity miss occurs when the size of the cache is too small for the amount of memory needed to execute the program under consideration. New processor architectures along with their application in mobile computing/communication/internet devices demand efficient cache architectures. To satisfy the fast access demands of the system, direct mapped caches are commonly used. While these satisfy the speed constraints, they increase the amount of conflict misses, since there is no alternate position for the requested data or instruction to occupy the cache (as is the case in set associative cache design). Modern processors contain both an instruction cache and a data cache. There have been several differing methods to reduce cache misses in both.

In our I-CoPES methodology, which is described in this paper, a novel scheme is introduced to reduce the before-mentioned instruction miss conflict. Our methodology uses a constructive algorithm to

Manuscript received February 18, 2002; revised August 31, 2003.

S. Parameswaran is with the School of Computer Science and Engineering, University of New South Wales, Sydney 2052, Australia (e-mail: sridevan@cse.unsw.edu.au).

J. Henkel is with NEC Laboratories America, Princeton, NJ 08540 USA (e-mail: henkel@nec-lab.com).

Digital Object Identifier 10.1109/TVLSI.2004.842936

re-order the mapping of basic blocks [2] in the cache, so that the total number of misses are reduced. Another heuristic algorithm is used to map those basic blocks to the main memory from the allocated cache locations reducing the number of conflict misses. This paper further presents the energy savings, which can be obtained by reducing conflict misses. In our system model all of the components of the system are on-chip, and therefore, for the first time we will be showing the total energy savings that can be achieved by reducing cache misses through instruction code placement for a complete system-on-chip (SOC).

II. Related Work

Diverse work has been published on the topic of reducing the cache misses by reorganizing data or instructions in the cache. The work on cache misses has predominantly concentrated on the data cache optimizations [7]–[13], [29], [28]. As for instruction cache optimization methods, many basic approaches have been proposed in the early 1990s with improvements and advances in recent years. We concentrate our discussion of related work to instruction caches.

Hwu *et al.* [3] effectively reduced the cache miss rates in a compiler called IMPACT-1 by function in-line expansion, trace selection, function layout and global layout. McFarling in [5] and [6] analyzed functions such that the dependencies among functions are exposed and exploited in order to reduce cache misses. Chow [22] reduced cache conflicts by sorting functions by their execution frequencies, and then grouping functions together to reduce conflict misses. All of the above methods are applied at the function level.

The first of the global methods was proposed in [27] by Tomiyama and Yasuura, where an ILP formulation is applied in order to reduce the cache misses. Their ILP formulation reduces the execution speed of the application, though not a suitable method for optimizing large programs. Performance estimation of such caches has been reported in the literature [26]. Kirovski *et al.* [25] use the operating frequency of the system and the clock frequency to estimate the miss penalty and with that information reduce cache misses by reorganizing basic blocks.

In [23] and [24] the authors use a scheme where half the cache is assigned for high priority tasks and the other half is allocated for nonhigh priority tasks. This method is applied to instruction cache optimization in multiprocessor systems by Li and Wolf in [14], though they later abandoned it in [15] for random placement of instructions in memory and doubling of cache sizes until deadlines were met. In [31], a constructive method is given to place tasks in memory for multi–processor systems.

Power/energy optimizations [16]–[20] are relevant to our approach. However, none of these have optimized for energy reduction by code placement.

III. Relevance, Assumptions, and Limitations of Our Work

In the work described in this paper, we give a constructive algorithm, which works effectively with the goal to reduce conflict misses. For the first time in the instruction cache optimization literature, we give a methodology to show the effect of code placement in both energy consumption and performance for a complete systems.

The following assumption apply in order to make use of our methodology.

1) The size of the cache is known *a priori*. This assumption is made to allow for greater optimization.
2) The size of a basic block is not bigger than the size of the cache. This assumption is quite valid in embedded systems where the basic blocks are usually small enough to fit into small caches. However, if the basic block is too large for the cache it can be broken up into smaller granules such that each granule will fit into the cache.
3) Only Level-1 caches are available for use. Having higher level caches does not reduce the effectiveness of the approach.
4) The caches are direct mapped.
5) The instructions of a task are allocated to a contiguous region of memory. This states that a task mapped to the cache will be in a contiguous region in that cache.
6) The problem is sufficiently large so that the total size of the instructions are several times larger than the size of cache.
7) The methodology targets the low power Application Specific Instruction Set Processors (ASIPs), where the cache size is tailored to suit the application, and silicon area restricts the use of large cache sizes.

IV. Problem Statement

Let the size of the cache associated with processor P be equal to S. The basic blocks to be placed in the cache be $b_1, b_2, \ldots, b_n$ all of which belong to the set B. Each block b_i is associated with size s_i. Let a loop l_r be defined as being a subset of B and the loops be numbered from $l_1, l_2, \ldots, l_r, \ldots, l_m$. Note that while multiple loops can share a basic block, the set of basic blocks within a loop will be unique to that loop. Some basic blocks will not belong to any of the loops. The problem is to arrange the basic blocks of each of the loops (and other basic blocks which do not belong to any of the loops) in main memory so that it can be brought into the cache such that the conflict misses are reduced; and the total main memory used is minimized.

It can be shown that in a special case of the problem, where a basic block b_i can only belong to a single loop l_k, is a bin-packing problem. Therefore, this problem is at least as hard as the bin-packing problem which is NP-Hard. To solve this problem in reasonable run-time it is necessary to use heuristics.

V. Allocation of Basic Blocks in Cache and Memory

The overall methodology contains two algorithms. The first algorithm places basic blocks in the cache so that basic blocks with high frequency are swapped out as little as possible. The second algorithm takes the placed basic blocks and maps them into main memory. Both of these algorithms are performed as a preprocessing step, taking the application's original instructions in memory and re-mapping them to different locations.

In order to re-map instructions, it is necessary to identify basic blocks which are executed together and the number of times they are executed. We identified these parameters by running the application through an instruction set simulator (ISS). Without the ISS it is impossible to determine critical sequences of blocks which execute together. The number of basic blocks in the applications under consideration varied from 100 to 900.

A. Cache Allocation

The cache allocation algorithms is given in Fig. 1. The complexity of this algorithm is $O(B^2)$, where B is the number of basic blocks.

B. Memory Allocation

The memory allocation algorithm takes the already placed basic blocks in the cache and directly maps them to the memory. Fig. 2 shows an example of how the basic blocks are mapped to the memory from the cache. In this figure blocks 1 to 5 are mapped directly on to the memory, but the block 6 is mapped to some memory locations further away, such that the mapped block will end up in the desired position in the cache. Thus, if a basic block is mapped to the location from t_x to e_x in cache of the processor, the basic block can be placed in memory at any one of the address ranges from addresses $t_x + i * S$

For Each *super loop in ordered list* {
Until *Cache is filled* {
Allocate basic blocks to the cache in descending order of frequency f_b until no more blocks can be allocated
}
Reorder *Unallocated basic blocks in order of size and place in list* BB_u ($BB_u = bb_{u1}, bb_{u2}, bb_{u3}...bb_{uy}$, *where y is the number of unallocated tasks for that super loop)*
Allocate bb_{u1} *from address* A_{ls} *to end of cache (where* A_{ls} *= S - sizeof (*bb_{u1}*))*
Remove bb_{u1} *from* BB_u
Repeat *until all tasks are allocated* {
Find *the next largest unallocated Task* bb_{up} *from the list* BB_u
Allocate bb_{up} *from address* A_{ls} *to* A_{le} *where* $A_{le} = A_{ls} + sizeof(bb_{up})$
Mark bb_{up} *from list* BB_u *as allocated*
Move along the list BB_u *and place as many tasks as possible between* A_{le} *and* S
Mark placed tasks as allocated }}

Fig. 1. Cache algorithm.

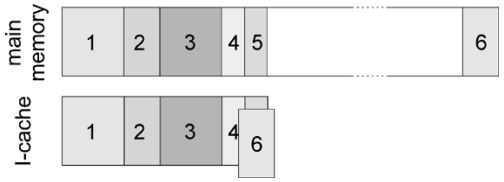

Fig. 2. Memory allocation example.

to $e_x + i * S$, where i is a positive integer. However, since tasks in the cache will wrap around the cache, an offset Z_r, can be added to each basic block allocated from super loop sl_r, and the basic block can be placed from memory location $t_x + Z_r + i * S$ to memory location $e_x + Z_r + i * S$. This introduction of the offset allows the reduction in size of the total memory needed for the system.

Fig. 3 gives the final memory allocation algorithm. The complexity of this algorithm is $O(SL*M)$, where SL is the number of superloops and M is the size of the code.

VI. Validation

In order to verify the usefulness of the I-CoPES approach we had to provide a simulation environment that captures the behavior (timing true and data true) of a whole sub-system comprising a SPARC CPU [32] CPU,[1] the caches (instructions and data), the main memory and the buses in between. Data cache is set to 1024 bytes and never varied. The instructions are single instructions issued in order. The simulation environment, shown in Fig. 4, takes as an input the application program plus typical input stimuli data (as far as the application characteristics, i.e., run-time, instruction cache access, depends on the input stimuli). A trace simulator (QPT, see [21]) generates the instruction traces and feeds them into a cache simulator (Dinero III, see [21]). The output of the cache simulator are cache miss/hit ratios that are fed into analytical power models for estimating the respective energy data for the I-cache, D-cache, buses and main memory whereas the power estimation of the CPU is accomplished by an ISS that is coupled to a table containing energy data for each instruction and addressing mode. Energy consumption and performance estimation models are obtained from [30].

[1]The CPU is an embedded version of the SPARC. The features in a summary are four-stage pipeline, delayed branch, register windowing, in-order execution, etc. The core is "lightweight" (around 120 kgates).

Reorder *super loops from largest sum to smallest sum of total basic block size and name them* $sl_a, sl_b, sl_c...$
For *all basic blocks ordered in the cache allocation order in* sl_a *do* {
$i = 0$
While *basic block is not allocated* **do** {
If *memory locations* $t_x + i*sizeof(cache)$ *to* $e_x + i*S$ *is free* **then**
Map *basic block to address* $t_x + i * S$ *to* $e_x + i * S$
Else
$i++$ } }
For *all basic blocks ordered in descending order of size in the next super loop until the end of the super loop list* **do** {
Allocate *largest basic block in the r st available contiguous memory block (*M_x *to* M_y*), which will hold the basic block*
Calculate $Z_r = (M_x mod S) - t_x$, *where* t_x *is the address in which the basic block being allocated starts in the cache at address* 0 *and* sl_r *is the present super loop under consideration*
While *basic block not allocated* **do** {
If *memory locations* $t_x + Z_r + i * S$ *to* $e_x + Z_r + i * S$ *is free* **then**
Map *basic block to address* $t_x + Z_r + i * S$ *to* $e_x + Z_r + i * S$
Else
i++ } }

Fig. 3. Main memory algorithm.

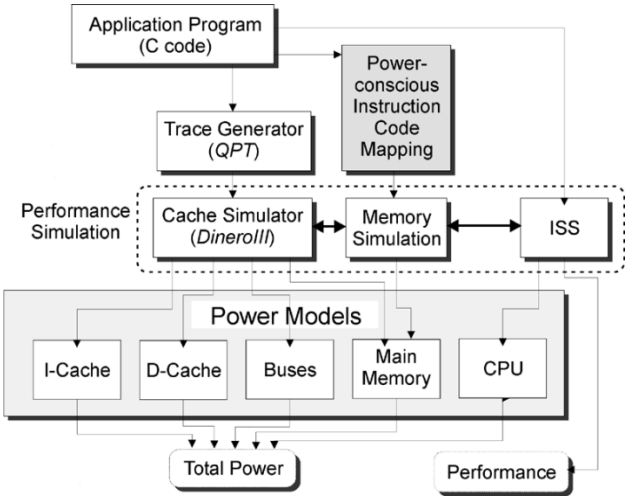

Fig. 4. Experimental setup that allows to simulate the power/energy consumption and performance of the whole system.

VII. Experiments and Results

We have validated the I-CoPES methodology by means of a set of seven applications. The applications have been chosen with as much variety in characteristics as possible in order to show the wide application area of the methodology. Thus, the applications varied in size (8 k to 200 k), application area (video, animation, algorithmic, etc.) and application domain (data dominated and/or control dominated). The applications used were a complete MPEGII video encoder *MPEG*, the problem of locating eight queens on a chess board without interfering *q8* (we used 11 queens to make the problem reasonably large), a video trick animation algorithm *trick1*, the Whetston benchmark sequences *whetston*, the UNIX command compress *compress*, a chromakey video mixer as part of a digital video studio equipment, and the travelling salesman problem *tsp*.

TABLE I
RESULTS

App.	I-Cache Size	# of Inst.	FIFO miss	Miss Ratio	I-CoPES Miss	I-CoPES Miss Ratio	FIFO (J)	I-CoPES Energy (J)	Energy Sav %	Mem. Exp.
	128	22406459	16343210	72.94%	15985131	71.34%	23.966566	23.459693	2.11%	
	256	22406459	12592796	56.20%	11486547	51.26%	18.91491	17.35585	8.24%	see
mpeg	512	22406459	10721491	47.85%	8143642	36.35%	16.424782	12.887134	21.54%	disc-
	1024	22406459	7378424	32.93%	4654360	20.77%	11.969864	8.1840637	31.63%	ussion
	2048	22406459	582241	2.60%	310957	1.39%	2.8494426	2.4016655	15.71%	
	64	44814000	30307447	67.63%	23340791	52.08%	4.7998051	3.8059608	20.71%	0.5%
	128	44814000	27500971	61.37%	11381784	25.40%	4.4282823	2.1563028	51.31%	0.04%
Q8	256	44814000	4538800	10.13%	3772601	8.42%	1.21218817	1.11561781	7.97%	0.04%
	512	44814000	241	0.00%	241	0.00%	0.62180677	0.63018477	-1.35%	0%
	128	103104786	103104786	100.00%	103104786	100.00%	19.514151	19.602431	-0.45%	11%
	256	103104786	102669907	99.58%	100175851	97.16%	19.582832	19.368942	1.09%	23%
trick1	512	103104786	90421317	87.70%	64081456	62.15%	18.081579	14.2167598	21.37%	23%
	1024	103104786	73378138	71.17%	39026	0.04%	16.152711	5.53378491	65.74%	12.5%
	2048	103104786	18248122	17.70%	486	0.00%	8.673339	5.7163015	34.09%	0.1%
	128	1749402	1701363	97.25%	1635829	93.51%	0.29518329	0.28625952	3.02%	4.0%
	256	1749402	1215194	69.46%	836405	47.81%	0.22842737	0.1776056	22.25%	4.1%
whetston	512	1749402	108059	6.18%	106669	6.10%	0.073333428	0.076303336	-4.05%	1.0%
	1024	1749402	993	0.06%	993	0.06%	0.062205727	0.064868727	-4.28%	1.1%
	128	53280973	39686614	74.49%	28533943	53.55%	7.494007	5.9797702	20.21%	1.0%
	256	53280973	29161991	54.73%	10489863	19.69%	6.0605139	3.4465544	43.13%	2.1%
compress	512	53280973	7102452	13.33%	1555604	2.92%	3.0002462	2.23145635	25.62%	3.4%
	1024	53280973	1272890	2.39%	256458	0.48%	2.2959965	2.16208414	5.83%	1.7%
	2048	53280973	443168	0.83%	53789	0.10%	2.4338514	2.391145	1.75%	0.5%
	64	9849864	8123533	82.47%	7672106	77.89%	1.9128616	1.8714207	2.17%	12.6 %
	128	9849864	4299216	43.65%	3221671	32.71%	1.3871321	1.26189137	9.03%	17.2%
	256	9849864	1744685	17.71%	1407958	14.29%	1.03327393	1.01142947	2.11%	20.6%
key	512	9849864	779934	7.92%	416077	4.22%	0.90934229	0.88227793	2.98%	24.7%
	1024	9849864	95099	0.97%	13254	0.13%	0.8351728	0.8299966	0.62%	31.0%
	2048	9849864	52786	0.54%	3595	0.04%	0.87677332	0.87650856	0.03%	12.4%
	64	3403735	2657486	78.08%	2622434	77.05%	0.51087595	0.50016747	2.10%	7.1%
	128	3403735	1931083	56.73%	1902922	55.91%	0.41021391	0.40003662	2.48%	12.2%
	256	3403735	1240095	36.43%	1180148	34.67%	0.31553183	0.30108842	4.58%	28.3%
tsp	512	3403735	856395	25.16%	778015	22.86%	0.26643111	0.25965276	2.54%	32.8%
	1024	3403735	488049	14.34%	495250	14.55%	0.22323306	0.22760796	-1.96%	69.6%
	2048	3403735	288144	8.47%	362509	10.65%	0.212027719	0.22134353	-4.39%	42.1%

The final results are given in Table I.

As can be seen from the table, for most of the applications, there is a substantial reduction in miss ratios. For example, the miss ratio is reduced from 47.85% to 36.35% for the *mpeg* application with 512 bytes of cache. For the smaller cache sizes, the miss ratios are not reduced substantially since the application has several basic blocks which are comparable in size to the size of the cache. A more significant reduction in miss rates can be seen in the application called *trick1*, where for the case with cache size of 1024, the miss rates plunge from 71% to 0.04%. The miss rates for the *tsp* problem were reasonably high, and this was because a few long loops were executed repeatedly. Such a case requires a lot of swapping of the cache, and therefore the miss rates did not substantially change.

The common characteristic of programs which benefit most are the ones which have a sequence of basic blocks which run several times and that these basic blocks are less than the cache size. In the original code these basic blocks would have been scattered. The ones that least benefit are those programs whose main loop is well contained within the cache size and is not scattered in the original program. This is consistent with the results which we get, where large cache sizes can contain most of the program and will perform poorly compared to ones where the cache is small and a large amount of swapping has to take place.

The total memory sizes increased by between 0% and 70% . The average increase was 13%. Note that the *MPEG* application has a number of system calls which are not reflected by the generated trace. Thus there are a number of blocks with just two instructions (which would not be the case in a self contained embedded application). These two instruction blocks cause the memory size increase to not reflect reality (since a number of them have to be placed a cache size apart).

However, in an embedded system which is on a single chip, the unused memory locations can be removed under certain circumstances, and then there will be no noticeable increase in memory size and consequently the chip size will remain almost the same. There is a possibility, that an additional line might be needed for the address bus, and therefore in the extreme case, we could expect a small increase in chip size.

The energy consumption reduction is also substantial, as can be seen from the last column. There are a few cases where the energy consumed has increased, but these are for cases where the miss rates are relatively small. The increase is due to the increased switching which is the result of the reordering of blocks in memory. The energy consumption in the cases where it has increased has never been more than 4.5%, and where it has decreased, it has decreased by up to 65%.

Though we are focusing on the reduction of energy and obtain mostly significant reductions, this is *not* at the cost of performance. In fact, performance increases in all cases as the numbers of cache miss ratios show (in a comparison of columns Miss Ratio and I-CoPES Miss Ratio).[2]

The runtime for all of these applications varied from 1–8 min on a Sparc 10 machine. A much more substantial time (in the order of hours) was spent on extracting the instructions which formed the basic blocks and the loops which were made up of these basic blocks.

[2]There is only one exception i.e., application *tsp* with an I-cache size of 2048, but this would not be an appropriate cache size for this application anyway.

VIII. Conclusion

We presented in this paper a methodology which places instruction code in a constructive way into the memory such that the subsequent mapping to the instruction cache leads to a reduced cache miss ratio and thus to a power/energy minimization of the whole system (CPU, caches main memory, buses) combined with a performance increase. We utilize the fact that the application which is executed is well understood in an embedded system, and exploit the behavior of the program to optimize the system. We have validated our methodology using seven applications that range in size from 8 k to 200 k. The technique leads to a reduction of energy/power consumption of up to 65% for a whole system. This is at the cost of an average main memory increase of around 13%.

References

[1] *National Technology Roadmap for Semiconductors*, Semiconductor Industry Association (SIA), 1997.

[2] A. W. Aho, R. Sethi, and J. D. Ullmann, "COMPILERS Principles, Techniques and Tools," AT&T Bell Labs., 1987.

[3] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu, "IMPACT: An architectural framework for multiple-instruction-issue processors," *Computer Architecture News*, vol. 19, no. 3, May 1991.

[4] D. Liu and C. Svensson, "Impact of supply voltage on power consumption, speed, and reliability of CMOS circuits," in *Proc. 4th Int. Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS'94)*, Barcelona, Spain, 1994, pp. 116–123.

[5] S. McFarling, "Program optimization for instruction caches," in *Proc. 3rd Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, New York, NY, 1989, pp. 183–91.

[6] S. McFarling, "Procedure merging with instruction caches," *SIGPLAN Notices*, vol. 26, no. 6, p. 71, Jun. 1991.

[7] P. Panda, N. Dutt, and A. Nicolau, "Memory organization for improved data cache performance in embedded processors," in *Proc. 9th Int. Symp. System Synthesis (ISSS'96)*, San Diego, CA, Nov. 1996, pp. 90–95.

[8] P. R. Panda, N. D. Dutt, and A. Nicolau, "Efficient utilization of scratch-pad memory in embedded processor applications," in *Proc. Eur. Design and Test Conf.*, Paris, France, Mar. 1997.

[9] P. R. Panda and N. D. Dutt, "Behavioral array mapping into multiport memories targeting low power," in *Proc. 10th Int. Conf. VLSI Design*, Hyderabad, India, Jan. 1997.

[10] P. R. Panda, H. Nakamura, N. D. Dutt, and A. Nicolau, "Improving cache performance through tiling and data alignment," *Lecture Notes in Computer Science (LNCS)*, vol. 1253, pp. 167–185, 1997.

[11] P. R. Panda, N. D. Dutt, and A. Nicolau, "Memory data organization for improved cache performance in embedded processor applications," *ACM Trans. Design Automation Electron. Syst.*, vol. 2, pp. 384–409, 1997.

[12] P. R. Panda, H. Nakamura, N. D. Dutt, and A. Nicolau, "Augmenting loop tiling with data alignment for improved cache performance," *IEEE Trans. Comput.*, vol. 48, no. 2, pp. 142–149, Feb. 1999.

[13] P.-P. Ranjan, H. Nakamura, N. D. Dutt, and A. Nicolau, "A Data alignment technique for improving cache performance," in *Proc. Int. Conf. Computer Design (ICCD'97)*, Austin, TX, Oct. 1997, pp. 587–592.

[14] L. Yanbing and W. Wolf, "Hardware/software co-synthesis with memory hierarchies," in *Proc. Design Automation Conf. (DAC'98)*, San Francisco, CA, Jun. 1998, pp. 430–436.

[15] L. Yanbing and W. H. Wolf, "Hardware/software co-synthesis with memory hierarchies," *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.*, vol. 18, no. 10, pp. 1405–1417, Oct. 1999.

[16] B. P. Dave, G. Lakshminarayana, and N. K. Jha, "Cosyn: Hardware-software co-synthesis of embedded systems," in *Proc. 34th Design Automation Conf. (DAC'97)*, 1997, pp. 703–708.

[17] I. Hong *et al.*, "Power optimization of variable voltage core-based systems," in *IEEE Proc. 35th. Design Automation Conf. (DAC'98)*, 1998, pp. 176–181.

[18] T. Givargis, F. Vahid, and J. Henkel, "A hybrid approach for core-based system-level power modeling," in *Proc. ASP-DAC'99*, 1999, pp. 141–145.

[19] T. Simunic, L. Benini, and G. De Micheli, "Cycle–accurate simulation of energy consumption in embedded systems," in *Proc. Design Automation Conf. (DAC'99)*, 1999, pp. 867–872.

[20] M. Lajolo, A. Raghunathan, S. Dey, and L. Lavagno, "Efficient power co-estimation techniques for system-on-chip design," in *Proc. DATE'00*, 2000, pp. 27–34.

[21] M. D. Hill *et al.*, "WARTS: Wisconsin Architectural Research Tool Set," Comput. Sci. Dept, Univ. Wisconsin, Madison, 2004.

[22] F. Chow, "A Portable Machine-Independent Global Optimizer—Design and Measurements," Comput. Syst. Lab., Stanford Univ., Stanford, CA, 1983.

[23] D. B. Kirk, "SMART (Strategic Memory Allocation for Real-Time) cache design," in *Proc. Real Time Systems Symp.*, Los Alamitos, CA, 1989, pp. 229–237.

[24] D. B. Kirk and J. K. Strosnider, "SMART (Strategic Memory Allocation for Real-Time) cache design using the MIPS R3000," in *Proc. 11th Real Time Systems Symp.*, Los Alamitos, CA, 1990, pp. 322–330.

[25] D. Kirovski, L. Chunho, M. Potkonjak, and S.-W. H. Mangione, "Application-driven synthesis of memory-intensive systems-on-chip," *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.*, vol. 18, no. 9, pp. 1316–1326, Sep. 1999.

[26] Y. T. Li, S. Malik, and A. Wolfe, "Performance estimation of embedded software with instruction cache modeling," *ACM Trans. Design Automation of Electronic Systems*, vol. 4, pp. 257–279, 1999.

[27] H. Tomiyama and H. Yasuura, "Code placement techniques for cache miss rate reduction," *ACM Trans. Design Automation Electronic Systems*, vol. 2, pp. 410–429, 1997.

[28] C. Kulkarni, F. Catthoor, and H. De Man, "Code transformations for low power caching in embedded multimedia processors," *Proc. 1st Merged Int. Parallel Processing Symp. and Symp. Parallel and Distributed Processing*, pp. 292–297, 1998.

[29] F. Catthoor, N. D. Dutt, and C. E. Kozyrakis, "How to solve the current memory access and data transfer bottlenecks: At the processor architecture or at the compiler level?," in *Proc. Design, Automation and Test in Europe Conf. and Exhibition*, Paris, France, 2000, pp. 426–435.

[30] Y. Li and J. Henkel, "A framework for estimating and minimizing energy dissipation of embedded HW/SW systems," in *IEEE/ACM 35th Design Automation Conf. (DAC'98)*, 1998, pp. 188–193.

[31] S. Parameswaran, "Code placement in hardware software co-synthesis to improve performance and reduce cost," *Proc. Design Automation and Test in Europe*, pp. 627–633, 2001.

[32] G. Blanck and S. Krueger, "The SuperSPARC Microprocessor," *37th IEEE Computer Society Int. Conf. Dig. Papers (CompCon)*, pp. 136–141, 1992.