Instruction buffering exploration for low energy embedded processors*

Tom Vander Aa[1], Murali Jayapala[1], Francisco Barat[1], Geert Deconinck[1], Rudy Lauwereins[2], Henk Corporaal[3], and Francky Catthoor[2]

[1] ESAT/ELECTA, Kasteelpark Arenberg 10, K.U.Leuven, Heverlee, Belgium-3001
{first_name.last_name}@esat.kuleuven.ac.be
[2] IMEC vzw, Kapeldreef 75, Heverlee, Belgium-3001
[3] TU Eindhoven, Electrical Engineering, Den Dolech 2, 5612 AZ Eindhoven, Netherlands

Abstract. For multimedia applications, loop buffering is an efficient mechanism to reduce the power in the instruction memory of embedded processors. Especially software controlled loop buffers are energy efficient. However current compilers do not fully take advantage of the possibilities of such loop buffers. This paper presents an algorithm the explore for an application or a set of applications what is the optimal loop buffer configuration and the optimal way to use this configuration. Results for the MediaBench application suite show an additional 35% reduction (on average) in energy in the instruction memory hierarchy as compared to traditional approaches to the loop buffer without any performance implications.

1 Introduction and motivation

Low energy is one of the key design goals of the current embedded systems for multimedia applications. Typically the core of such systems are programmable processors. VLIW DSP's in particular are known to be very effective in achieving high performance and sometimes low power for our domain of interest [10]. Examples of such processors are the Trimedia [17] processor from Philips or the 'C6x processors from Texas Instruments [18]. However, power analysis of such processors indicates that a significant amount of power is consumed in the on-chip (instruction) memory hierarchy: 30% of the total power according to [4]. If the appropriate data memory hierarchy mapping techniques are applied first [7, 16], and if all methods to reduce power in the data path are applied [5], we have performed experiments that show this number goes up to 50% if nothing is done here. Hence, reducing this part of the budget is crucial in reducing the overall power consumption of the system.

Loop buffering is an effective scheme to reduce energy consumption in the instruction memory hierarchy [15]. In any typical multimedia application, significant amount of execution time is spent in small program segments. By storing

* This project is partially supported by the Fund for Scientific Research - Flanders (FWO) through projects G.0036.99 and G.0160.02 and the postdoctoral fellowship of G.Deconinck, and by the IWT through MEDEA+ project A502 MESA.

them in a small loop buffer (also called L0 buffer) instead of the big instruction cache (IL1), energy can be reduced. However, even the current way of applying this with a single central loop buffer and a standard compiler still leads to a total power contribution of more than 20%.

An important way to reduce this further is by more effectively managing these loop buffers through a dedicated pre-compiler. The pre-compiler should be responsible for mapping the appropriate parts of the application onto these L0 buffers. However, to the best of our knowledge there has been little work on algorithms and tools that use the information available in the application to effectively profit from a software controlled loop buffer. As will be shown, the instruction memory energy consumption can be significantly reduced when the appropriate loop buffer configuration is used in the appropriate way.

An algorithm to explore the loop buffer design space is presented. In that, given a program (e.g. in C), the algorithm finds the most energy efficient loop buffer configuration and also decides what loops should be mapped to the loop buffer. Results show an average reduction in instruction memory energy consumption for typical multimedia applications of 35% as compared to a simple loop buffer mapping method currently applied in existing work.

The rest of this paper is organized as follows: A brief account of the related work is presented in Section 2. In Section 3 the software controlled loop buffer organization is described. In Section 4 the energy model under consideration for our exploration tool is outlined. Section 5 describes the loop buffer exploration algorithm, which is our main contribution. Section 6 presents the simulation results and Section 7 is the conclusion.

2 Related Work

Several loop buffering schemes have been proposed in the past [12, 1, 2] An overview of the options can be found in [14]. Initially only inner loops without any control constructs could be mapped to the loop buffer. In [9] support is added for control constructs such as if-then-else, subroutine calls and nested loops. Our loop buffer architecture also supports conditional constructs, as well as mapping a set of nested loops. What is not discussed in [9] are the methodologies on how to efficiently use these new features.

Our main reason to use a software controlled loop buffer is to exploit knowledge about the program in the compiler to reduce power. But, as is discussed in [15], software controlled loop buffers also do not need the energy consuming tag memories. Furthermore, they do not suffer any cycle penalty.

The idea to add compiler support has already been proposed in [3]. In that paper, however, a regular cache is used (with tags) and no energy model is used directly in the exploration framework.

The creation of a custom memory hierarchy has already been explored extensively in a data memory context [7, 16]. For instruction memory, [8] presents tuning of the loop buffer size for simple loop buffer architectures, only supporting inner loops.

3 Operation of the low loop buffer organization

Figure 1 illustrates the essentials of the low power loop buffer under consideration. Instructions are fed to the processing unit either from the level 1 instruction cache or through the loop buffer.

Initially the loop buffer is turned off and the program executes via IL1 (Figure 1, left). When a special instruction is encountered marking the beginning of the loop that has to be mapped to the loop buffer, the loop buffer will be turned on. The form of this special instruction is `lbon <startaddress>, <endaddress>`, where *startaddress* is the address of the first instruction of the loop and *endaddress* that of the last one. These values are stored in the local controller (LC) and will be used during the execution of the loop.

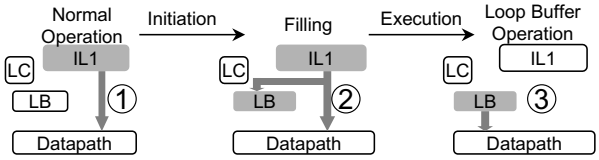

Fig. 1. The processor operates in three phases controlled by the local controller (LC): normal operation, filling of the Loop Buffer and Loop Buffer operation

If a `lbon` is encountered and no *startaddress* is stored in the local controller, or the *startaddress* in the LC is different from the one in the `lbon` instruction, the first iteration will be used to fill the loop buffer. The instruction stream will go from IL1 both to the loop buffer and to the processing unit (Figure 1, middle). After the first iteration the IL1 can be put into low power mode and only the loop buffer will be used (Figure 1, right).

When the loop buffer is used, a local controller will translate the program counter to an index in the loop buffer by using the stored *startaddress*. This mechanism reduces the power by avoiding the expensive tag memory found in normal caches.

When the LC detects the program counter is bigger than *endaddress*, the loop buffer will be turned off.

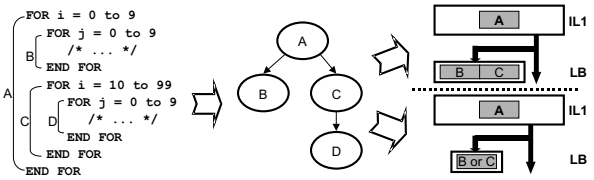

Fig. 2. Loop nest, corresponding tree representation and two possible mappings. A is the parent of B. B is the child of A. B and C are siblings.

4 Energy dissipation model

A tree like representation for the loops, as shown in Figure 2, is extracted from the source code of the application. From this representation we can identify different mappings of loops on the loop buffer. The figure shows two possible configurations for a given program, both leading to different loop buffer sizes and different energy consumptions. In the first configuration B and C are mapped to different locations in the loop buffer, leading to a bigger size. In the second, loops B and C are loaded each time the loop starts, leading to more loads from IL1. For a given mapping of loops, the energy E is:

$$\begin{aligned} E = \quad & \sum_{l \in unmapped}^{l} N_{exec}(l) \times E_{access}(IL1) \quad (1) \\ & + \sum_{l \in mapped}^{l} N_{load}(l) \times E_{access}(IL1) \quad (2) \\ & + \sum_{l \in mapped}^{l} N_{exec}(l) \times E_{access}(lb) \quad (3) \end{aligned} \tag{1}$$

The three sums correspond to the three places in the instruction memory where energy is consumed (see also Figure 1):

1. Executing instructions from IL1.
2. Loading the instructions into the loop buffer from IL1.
3. Executing instruction from the loop buffer on the processor core.

Loops that are *mapped* contribute to the last two terms, loop that are not mapped (*unmapped*) to the first term. N is the number of accesses to memory (loop buffer or IL1) due to the loading or execution of the loop. E_{access} is the energy per access of the loop buffer or the instruction level 1 cache. The value of $E_{access}(lb)$ depends on the size of the loop buffer, which on itself can be calculated when you know what loops are currently mapped and if loops are mapped together or reloaded each time the loops is invoked.

The energy values we have used for the memories (E_{access}) are calculated using the Wattch [6] power models. For the IL1 we used a regular cache, for the loop buffer we used a cache without tags. Although the Wattch model has some known limitations, it is still suited for our purpose since we only need good relative energy values.

5 Design Space Exploration

Since the compiler is responsible for inserting the `lbon` instructions, it should decide what is the most energy efficient way to do so. This leads to a design space exploration problem with the following goals:

1. What is the optimal loop buffer size? If the loop buffer is too small, not enough loops fit and there will be too many access to the IL1 cache. If the loop buffer is too big the energy per access of the loop buffer will be too big.
2. What is the best way to map the loops that fit? If we decide to use the loop buffer for a certain loop, we still have several options: Do we map the loop entirely or only parts of the loop? If we load two loops like B and C in Figure 2, do we put them in the same address space or in different address

spaces? The former will save us space in the loop buffer, we will have to load the loop each time it starts. The latter case needs a bigger loop buffer, but will save us accesses to IL1 to load the loops (as can be seen from the energy model from the previous section).

5.1 Size of the design space

Using a brute force approach to find the optimal mapping of loops to the loop buffer by trying all the configurations is not feasible. The number of combinations you would have to try for the outer-most loops of the program is:

$$\#Sol(OuterLoop) = 2 + \prod_{C \in Child\ Loops} \#Sol(C) \tag{2}$$

We count the two basic solutions (mapping the loop completely or not mapping the loop at all), plus all the combinations of possible solutions of the immediately nested loops (children). For MPEG2 encoding, for example, this would lead to more than 10^{48} combinations. It's clear that we need an effective heuristic to find an optimal mapping.

5.2 Exploration algorithm

Instead of the brute force approach, we use a recursive algorithm to explore our design space in a more intelligent way (see Algorithm 1). The loops of a program are represented as a tree as already presented in Figure 2. The Algorithm works like this: the procedure `Find_Mappings` will be called with the top loop of the tree as an argument. If the program has multiple top loops, a dummy loop with iteration count of one, can be assumed around the whole program.

Two basic solutions exist for a loop passed to `Find_Mappings`: mapping the loop completely or not mapping the loop at all. Both solutions are kept since they will have a different energy consumption and required loop buffer size. If the loop has children, `Find_Mappings` is called recursively to find the solutions of the child loops. These solutions are then combined in the procedure `Filter_Solution`. Here a heuristic is used to prune the design space: if for a certain loop and loop buffer size several solutions exist, we only keep one solution for each possible size, namely, the most energy efficient solution. Although we do not have a proof, we believe that only this solution might lead to the optimum for the top loop. For several small examples we did an exhaustive search, and for these our assumption was indeed true. Using this heuristic, the number of solutions you have to keep is limited to the number of different loop buffer sizes you will encounter. For a typical application this will be less than 1000 combinations, which is much less than the total number of combinations of Equation 2.

5.3 Extension: partial mapping of loops

We have extended the algorithm to support partial mapping of loops. If a certain loop body contains basic blocks that are almost never executed (they contain for example exception handling code), it does not make sense to map those to the loop buffer. To add support for this we had to make a minor modification to the

Algorithm 1 Design space exploration to map a given set of loops to a loop buffer. Energy and needed loop buffer size are calculated using Equation 1

Procedure: Find_Mappings
Input: Loop l
Output: Set of solutions $S = \{(z_1, E_1), (z_2, E_2), ...(z_n, E_n)\}$ such that for each possible loop buffer size z_i, E_i is the minimal energy for that size}
Begin
/* 2 base solutions */
$S \Leftarrow \{(\ 0,$ Compute_Energy(l, unmapped) $)\}$
$S \Leftarrow S \cup \{(\ $ size(l), Compute_Energy(L, mapped) $)\}$

/* Find the solutions of the children */
let $\{c_1, c_2, ..., c_n\} \leftarrow$ children of l
for $i = 0$ to n **do**
$Sol_i \leftarrow$ Find_Mappings(c_i)
end for
Filter_Solutions($S, Sol_1 \times Sol_2 \times ... \times Sol_n$)
End

Procedure: Filter_Solutions
Input: $Sol_1 \times Sol_2 \times ... \times Sol_n$, Set of Solutions S
Output: Updated set of Solutions S
Begin
/* Combine the solutions of the children */
/* filtering out the not needed solutions */
for all $(sol_1, sol_2, ..., sol_n) \in Sol_1 \times Sol_2 \times ... \times Sol_n$ **do**
let $(z, energy\_current) \Leftarrow$ New solution by combining $s_1, s_2, ..., s_n$
/* We only keep one (the best) solution per size */
let $(z, energy\_optimal) \Leftarrow$ Current solution with size z
if $energy\_optimal$ does not exist yet **then**
$S \Leftarrow S \cup \{(z, energy\_current)\}$
else if $energy\_current < energy\_optimal$ **then**
$S \Leftarrow S \setminus \{(z, energy\_optimal)\} \cup \{(z, energy\_current)\}$
end if
end for
End

algorithm. Instead of only two basic solutions – mapping the loop completely or not mapping the loop – we now also consider all possibilities in between. We start with no basic blocks mapped, and for each new possibility we add the block that is executed the most amongst all of them that are not mapped yet. The solution generated last is precisely the one where the whole loop is mapped.

This scheme can be handled with our loop buffer hardware since the compiler we use applies trace-based code layout [19]. The blocks of the most frequently executed trace are grouped together at the beginning of the procedure. The less executed code is put at the end of the instruction layout. By changing the value of *endaddress* of the `lbon` instruction, you decide what basic blocks are mapped.

6 Results and discussion

We have used the MediaBench [13] application suite to compare our work, both mapping of only complete and partial loops, to existing implementations of the loop buffer. Table 1 shows the benchmarks we used, and the optimal loop buffer size (in number of instructions) for each benchmark in terms of energy.

Table 1. List of benchmarks used, and optimal loop buffer size

Benchmark	Description	Opt. Size	Benchmark	Description	Opt. Size
ADPCM decode	Audio	45	ADPCM encode	Audio	53
AES	Encryption	88	Blowfish encode	Encryption	51
JPEG decode	Image	128	JPEG encode	Image	59
EPIC	Image	55	g721 decode	Audio	144
g721 encode	Audio	155	ghostscript	Image	85
gsm decode	Audio	81	gsm encode	Audio	81
H.263	Video	182	mesa osdemo	3D graphics	64
MPEG2 decode	Video	57	MPEG2 encode	Video	76
Rasta	Speech Recogn.	76	SHA	Encryption	25
Snake	3D graphics	179	**Overall Optimum**		**99**

6.1 Energy versus loop buffer size

Figure 3 (left) shows for an example application the loop buffer size versus the energy of the optimal mapping on a loop buffer of that size. Since the algorithm we use (the extended one of Section 5.3) only generates solutions for certain loop buffer sizes, the graph is discrete. Figure 3 (right) shows the same information but for all applications tested together. For all sizes between 0 and 255 we took the energy of optimal solutions for that size of all applications. The sum of those values leads for the optimal solution for each size.

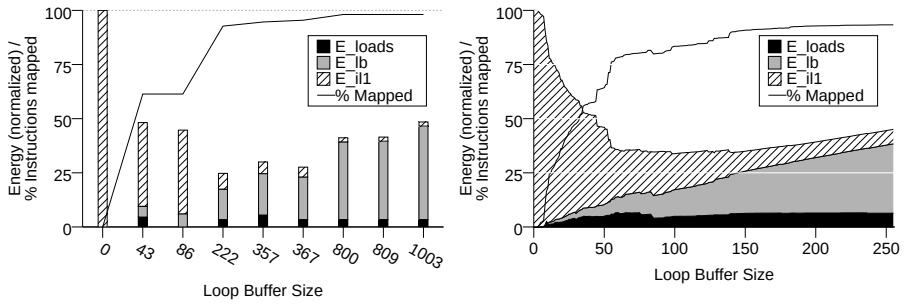

Fig. 3. Instruction memory energy consumption versus the loop buffer size, for one specific benchmark (left) and for all benchmarks together (right)

The graph also shows you can indeed efficiently explore the energy-size trade-off with the algorithm. For small sizes the total energy is dominated by the IL1

energy. As the loop buffer size increases more loops will be mapped to the small loop buffer and the IL1 energy will go down. With the size, also the energy per access of the loop buffer increases, until at a certain point the most energy will be consumed in the loop buffer itself. Generally, this will happen in the *knee* of the *"% Mapped"*-curve, where it does not pay off anymore to map more loops to the loop buffer.

An interesting feature to notice in the left graph, is that the algorithm produces two solutions with the same loops mapped. One solution needs a size of 43, the other a size of 86. This is because the program has two loops of the same size (43), and they can be mapped to different locations in the loop buffer, needing a loop buffer of size 86, or they can be mapped on the same location. For this last solution you are required to load the loop each time you enter it, resulting in more loads from IL1 to the loop buffer. You cannot say beforehand what will be the most energy efficient solution in such case, since this depends on the behavior of the program.

The other graph shows the global energy behavior of all applications together. The minimal energy is consumed for a loop buffer of size 99. So if you would like to build *one* loop buffer for the whole application domain, 99 would be the best choice.

6.2 Comparison of different mapping strategies

Figure 4 shows the normalized energy consumption in the instruction memory of the optimal mapping according to five different mapping strategies:

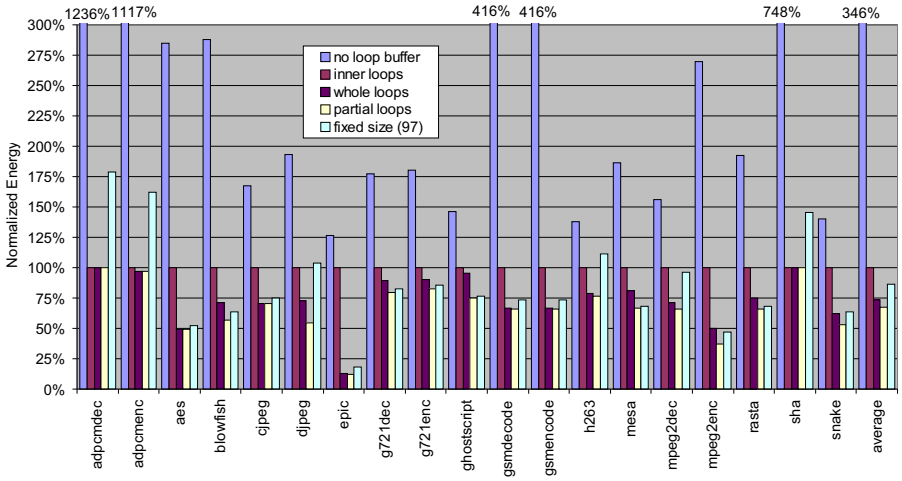

Fig. 4. Energy comparison of different loop buffer mapping strategies applied on the MediaBench suite

- **no loop buffer**: All loops are executed from the IL1 memory. Since much previous work has been done on loop buffers and it is well known that multimedia applications contain a lot of loops, making them very suitable for use with a loop buffer, has been done it would not be fair to use this case (no loop buffer) as our reference case. What can be seen, however, is that by simply using a loop buffer, although maybe in a not so clever way, you can already reduce the energy in instruction memory significantly (72% on average).
- **inner loops**: This strategy only maps the most inner loops to the loop buffer. Since this is what is implemented in existing loop buffer architectures, the other energy values are normalized against this strategy.
- **whole loops**: This is the basic version of the algorithm, as described in Section 5.2. The average gain amongst all benchmarks as compared to the previous strategy is 35%, with maxima for some benchmarks of up to 88%. This gain has two main reasons: the first reason for energy reduction is the fact that not only inner loops but also the outer loops can be mapped. If the inner loops have a too low iteration count, we can also map the surrounding loop(s), because we have support in our loop buffer for nested loops. For this reason simple applications with mostly inner loops, such as ADPCM and SHA, perform already very well with the strategy that only maps inner loops. For complex applications, such as MPEG2 and JPEG, taking into account all the loops and not just the inner one, has indeed a big impact (a gain of a factor 2 or more). The other main reason is the fact that we explore the trade off between mapping two sibling loops separately or together on the loop buffer.
- **partial loops**: This is the extended version of our algorithm, that is able to map loops only partially. For certain benchmarks, such as ghostscript or MPEG2 Decoding, there are loops that contain less frequently executed basic blocks. For these benchmarks not mapping the whole loop gives a significant gain. Averaged over all applications the gain compared to the previous strategy is limited, only 7%.
- **fixed size**: As can been seen in Figure 3, the overall optimal size over all benchmarks is 99 instructions. We have fixed this size and calculated the optimal mapping for all benchmarks. For some benchmarks the optimal mapping (with a free-to-choose loop buffer size, cf. Table 1) is close to 99. For these benchmarks the penalty for not being able to choose the size of the loop buffer is small. For others, however there is a penalty because 99 instructions is not enough or too much for these benchmarks. The average penalty as compared to *partial loops* is 20%. A solution here is to perform additional code transformations to make your application more suitable for a specific loop buffer configuration.

7 Conclusion and Future Work

This paper presented loop buffer exploration tool based on detailed analytical energy models for software controlled loop buffers. An algorithm was presented to

find a good loop buffer configuration and an optimal mapping for an application on the loop buffer. The knowledge the compiler has about the application is exploited to make the right decisions. Different loop buffers can be evaluated allowing the user to make the trade-off between the size of the loop buffer and the consumed energy.

The algorithm was demonstrated using MediaBench, giving an average of 65% reduction in energy consumption, with peaks of 88% for some applications. The algorithm also allows designers to find the best loop buffer configuration for an application domain instead of one single application.

After these optimizations the original contribution of the instruction memory to the total processor power has gone down from 53% to 20%. This means that there is still room for improvement. Since the loop buffer is now the *hot spot* of the instruction memory, clustering [11] can be applied to the loop buffer to further reduce the energy. How to optimally use a software controlled, clustered loop buffer will be the subject of future work.

References

1. Tim Anderson and Sanjive Agarwala. Effective hardware-based two-way loop cache for high performance low power processors. In *Proc of ICCD*, September 2000.
2. R. S. Bajwa and et al. Instruction buffering to reduce power in processors for signal processing. *IEEE Transactions on VLSI*, 5(4):417–424, December 1997.
3. Nikolaos Bellas, Ibrahim Hajj, Constantine Polychronopoulos, and George Stamoulis. Architectural and compiler support for energy reduction in the memory hierarchy of high performance microprocessors. In *Proc of ISLPED*, August 1998.
4. L. Benini, D. Bruni, M. Chinosi, C. Silvano, V. Zaccaria, and R. Zafalon. A power modeling and estimation framework for vliw-based embedded systems. In *in Proc. Int. Workshop on Power And Timing Modeling, Optimization and Simulation PATMOS*, September 2001.
5. Luca Benini and Giovanni de Micheli. Sysmtem-level power optimization: Techniques and tools. *ACM TODAES*, 5(2):115–192, April 2000.
6. David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *Proc of ISCA*, pages 83–94, June 2000.
7. Francky Catthoor, Koen Danckaert, Chidamber Kulkarni, Erik Brockmeyer, Per Gunnar Kjeldsberg, Tanja Van Achteren, and Thierry Omnes. *Data access and storage management for embedded programmable processors*. Kluwer Academic Publishers, March 2002.
8. S. Cotterell and F. Vahid. Tuning of loop cache architectures to programs in embedded system design. In *Proc of International Symposium on System Synthesis (ISSS)*, October 2002.
9. A. Gordon-Ross, S. Cotterell, and F. Vahid. Exploiting fixed programs in embedded systems: A loop cache example. In *Proc of IEEE Computer Architecture Letters*, Jan 2002.
10. Margarida F. Jacome and Gustavo de Veciana. Design challenges for new application-specific processors. *Special issue on Design of Embedded Systems in IEEE Design & Test of Computers*, April-June 2000.
11. Murali Jayapala, Francisco Barat, Pieter OpDeBeeck, Francky Catthoor, Geert Deconinck, and Henk Corporaal. A low energy clustered instruction memory hierarchy for long instruction word processors. In *Proc of PATMOS*, September 2002.
12. Johnson Kin, Munish Gupta, and William H. Mangione-Smith. Filtering memory references to increase energy efficiency. *IEEE Transactions on Computers*, 49(1):1–15, January 2000.
13. Chunho Lee and et al. Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *International Symposium on Microarchitecture*, pages 330–335, 1997.
14. Lea Hwang Lee, Bill Moyer, John Arends, and Ann Arbor. Low-cost embedded program loop caching - revisited. Technical report, EECS, University of Michigan, December 1999.
15. Lea Hwang Lee, William Moyer, and John Arends. Instruction fetch energy reduction using loop caches for embedded applications with small tight loops. In *Proc of ISLPED*, August 1999.
16. Preeti Ranjan Panda, Nikil D. Dutt, and Alexandru Nicolau. Memory data organization for improved cache performance in embedded processor applications. *ACM TODAES*, 2(4):384–409, 1997.
17. G.A. Slavenburg, S. Rathnam, and H. Dijkstra. The Trimedia TM-1 PCI VLIW media processor. In *Proceedings Hot Chips VIII Conference*, 1996.
18. Texas Instruments Inc., http://www.ti.com. *TMS320 DSP Family Overview*.
19. Trimaran group, http://www.trimaran.org. *Trimaran: An Infrastructure for Research in Instruction-Level Parallelism*, 1999.