Instruction Cache Energy Saving Through Compiler Way-Placement

Timothy M. Jones†, Sandro Bartolini‡, Bruno De Bus§, John Cavazosζ and Michael F.P. O'Boyle†

†Member of HiPEAC, School of Informatics, University of Edinburgh, UK, {tjones1,mob}@inf.ed.ac.uk

‡Member of HiPEAC, Dipartimento di Ingegneria dell'Informazione, Università di Siena, Italy, bartolini@dii.unisi.it

§Member of HiPEAC, Department of Electronics and Information Systems (ELIS), University of Ghent, Belgium, bruno.debus@elis.ugent.be

ζComputer and Information Sciences, University of Delaware, USA, cavazos@cis.udel.edu

Abstract

Fetching instructions from a set-associative cache in an embedded processor can consume a large amount of energy due to the tag checks performed. Recent proposals to address this issue involve predicting or memoizing the correct way to access. However, they also require significant hardware storage which negates much of the energy saving.

This paper proposes way-placement to save instruction cache energy. The compiler places the most frequently executed instructions at the start of the binary and at runtime these are mapped to explicit ways within the cache. We compare with a state-of-the-art hardware technique and show that our scheme saves almost 50% of the instruction cache energy compared to 32% for the hardware approach. We report results on a variety of cache sizes and associativities, achieving 59% instruction cache energy savings and an ED product of 0.80 in the best configuration with negligible hardware overhead and no ISA changes.

1 Introduction

Embedded processors often dissipate a large fraction of their total power budget in their on-chip memories. The StrongARM SA-100, for example, consumes 27% of its total energy in the instruction cache [13]. However, designers must often use higher-energy set-associative instruction caches, rather than direct-mapped, to maintain high performance and reduce cache misses.

There have been many recent proposals to reduce the energy of a set-associative instruction cache access [12, 6]. For example, Ma *et al.* [12] proposed a way-memoization scheme that avoids tag checks completely in the vast majority of accesses. Cache lines are augmented with link information that points to the next cache way to be accessed. In this approach the next instruction can be fetched from the linked cache line without any tag comparison, as long as the link is valid. However, the downside to this and similar pure-hardware schemes is that they require extra storage to be added to the processor to indicate the next way to be accessed, introducing a power overhead and negating some of the energy savings achieved.

This paper proposes a novel instruction cache energy saving method that is compiler-controlled, rather than requiring substantial hardware support. Our scheme is flexible, working across different cache configurations without the need for any recompilation, and requires no ISA changes. It works by placing frequently executed code at the start of the program binary. Whenever these instructions are brought into the cache they are explicitly placed in a set and way determined by bits from their addresses. The portion of the binary mapped in this manner is termed the *way-placement area*. We choose the frequently executed instructions to place in this manner since they cause the majority of instruction cache accesses. Whenever a way-placed instruction is fetched from the cache, only the specific set and way that it resides in needs to be accessed. The tag checks for the other ways can be removed, saving energy. By explicitly placing frequently executed instructions in particular ways, we remove the uncertainty surrounding prediction schemes [6], avoid the need for extra storage [12], and eliminate the tag check in all other ways of the cache.

Furthermore, depending on the specific configuration of the instruction cache, *e.g.* its size or the amount of associativity it uses, different sizes of the way-placement area can be selected. The size of the way-placement area can be dynamically decided since our optimised program layout is compatible with different sizes of way-placement area. Hence, there is no need to recompile programs using our way-placement scheme.

The rest of this paper is structured as follows. Section 2 gives a small example of how our way-placement scheme works and section 3 then provides a detailed overview of our scheme. Section 4 discusses the hardware modifications we make and section 5 describes our experimental setup. We present our results in section 6 and related work in section 7. Finally, we conclude this paper in section 8.

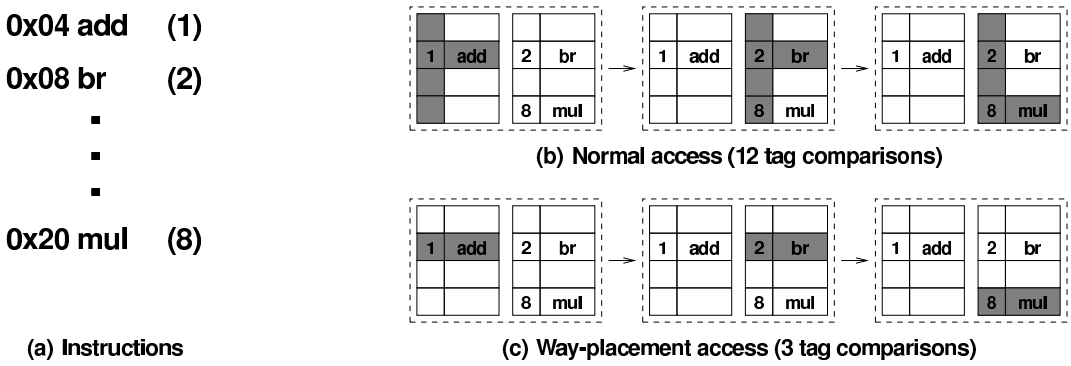

Figure 1. Access to three instructions from the instruction cache. The instructions are shown with their tags in brackets. We show a small cache with two sets and four ways as an example. During normal access all the tags in one set are checked, resulting in 12 comparisons. Using our scheme means that only one tag in one set needs to be compared each time, meaning just 3 comparisons.

2 Way-placement example

In embedded processors, associative caches are typically employed and their usage can contribute to a significant fraction of total processor energy. In figure 1 we show an example of accessing ways in the instruction cache in both the baseline and using our way-placement approach.

Figure 1(a) shows three instructions, their addresses, and their tags. The accessing of these three instructions from the instruction cache is shown in figure 1(b). Each diagram shows a two-set, four-way cache laid out in the same manner as in the XScale[1] [7]. Each set in the cache is a fully-associative sub-bank with all the ways grouped together. So each vertical block is a set containing all four ways and each horizontal line is a way within the set. As in the XScale, cache tags on the left of each block are stored in a CAM cache [16] and are searched first with instructions on the right accessed on a tag match [7]. In this baseline case (figure 1(b)), first the *add* is fetched, so the left-hand set is queried by performing a fully-associative search. Next is *br* so the right-hand set is searched. Finally the *mul* which requires the right-hand set to incur another fully-associative tag lookup. Thus, a standard instruction cache without way-placement requires 12 tag comparisons.

The method for querying the instruction cache in our approach is shown in figure 1(c). Here we know exactly which set and way the required instruction will be in, if it is in the cache, as described in section 4. Therefore we just need one tag comparison each time. As before, the *add* is fetched first and we only require tag check in the left-hand set. When the *br* and *mul* instructions are accessed we perform one tag lookup for each in the right-hand set. In total, our approach requires only 3 tag comparisons, a saving of 75%.

[1]We implement our scheme in an XScale simulator

3 Compiler analysis for way-placement

Our compiler-controlled technique uses profile information to place frequently-executed code in the way-placement area. First we read in the object files and libraries that are to be linked together, constructing an interprocedural control-flow graph (ICFG) where each node is a basic block. These blocks are annotated with execution counts obtained through profiling. We then construct chains of basic blocks, linking blocks when they have a predefined ordering that we must respect (*i.e.* call/return site pairs or blocks that have a fall-through edge between them). Once this is complete, all remaining basic blocks are considered as chains by themselves.

The next stage of our algorithm uses the execution counts, with which we have annotated the blocks, to guide code placement. We assign a weight to each chain that is equal to the sum of the instruction counts in that chain. Then we order the chains by weight so that the heaviest chain is first, and link them together to form one large chain for the program, which is written out as the final binary.

4 Microarchitectural modifications

The modifications to the microarchitecture to support our way-placement scheme are minor and are made to the instruction translation lookaside buffer (I-TLB), described in section 4.1, and the instruction cache, described in section 4.2. The overheads of our scheme are simply one bit per entry in the I-TLB and a single extra bit for the whole instruction cache.

4.1 I-TLB modifications

We make the way-placement area a multiple of the memory page size, meaning we can associate a single bit with

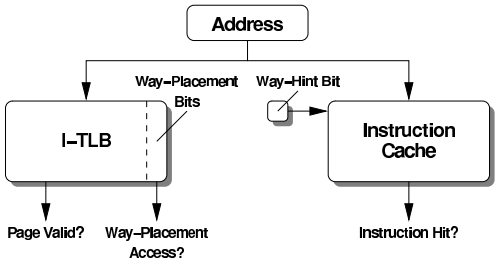

Figure 2. Overview of our microarchitectural changes. Way-placement bits are stored in the I-TLB. Since the cache and I-TLB are accessed in parallel the way-hint bit indicates whether to perform a way-placement access.

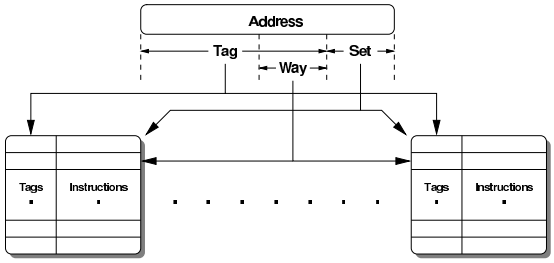

Figure 3. Accessing the instruction cache in our scheme. As normal, the address is broken up into the set and the tag. In addition, the least significant bits of the tag are also used to select the correct way within the set.

each page in the I-TLB that indicates an access to the way-placement area. We call this the way-placement bit. It is stored with existing page permission bits and set by the operating system when it writes into the I-TLB. Our compiler pass (described in section 3) always puts the best candidate instructions for way-placement at the start of the binary and those that are less suitable towards the end. Knowing that this will happen enables the operating system to choose the best sized way-placement area either on a static or per-program basis, even adjusting it during program execution. This coarse-grained approach to way-placement keeps the hardware overheads extremely low.

Accesses to the I-TLB must be performed in parallel with the instruction cache in order to keep the access latency low [7]. This means that we do not know whether we are reading from the way-placement area until after the access has occurred (and we have read the way-placement bit). To overcome this problem we provide a single extra bit which is accessed before the instruction cache. It is called the way-hint bit and it records whether the previous access was to the way-placement area or not. Figure 2 shows how the I-TLB, instruction cache and way-hint bit are accessed.

There are two scenarios where the way-hint bit could be wrong. In the first, the way-hint bit falsely indicates a non-way-placement access. Here we simply miss an opportunity to save energy. In the second scenario the way-hint bit informs us that we are accessing an instruction in the way-placement area, but after reading the I-TLB we find out that we were not. In this case we must perform a second access to the instruction cache, reading all ways. Here we incur a cycle penalty and energy overhead for the additional instruction cache access.

Using the way-hint bit to predict a way-placement access is very accurate since the processor will spend a long time reading instructions from the way-placement area. The instruction stream rarely switches between the way-placement area and outside because we have grouped the frequently executed basic blocks together (section 3). Hence in reality the performance and energy overheads of using this bit are negligible. However, they are still accounted for fully in all our experiments.

4.2 Instruction cache modifications

The modifications we make to the instruction cache are simple to achieve and allow us to perform a tag check in only one way of one set in our highly associative cache. We target the XScale processor [7] which uses content addressable memory (CAM) to implement the cache [13], although our scheme could also easily be applied to a standard RAM cache. Each CAM sub-bank within the cache contains all the ways from one cache set [16]. A fully-associative search is made for the required tag in one sub-bank on each access. There is one match line for each way within each sub-bank and these are precharged at the start of the access. If one of the CAM tags matches the broadcast tag then it discharges its match line and reads the corresponding data.

Our modifications are to simply disable the tag check and match line precharging to all but the required way on a way-placement access. The correct way is chosen by using the least significant bits from the address tag. A simple multiplexor can be used to select one of 2^N ways given N bits from the tag. For example, a 32-way cache uses the lower 5 bits from the tag to select the way when performing way-placement accesses. Note that the tag remains the same length as the way-placement bits are also used as part of it. A further modification, also used in [12], is to avoid tag checks completely when accessing an instruction from the same line as the previous access.

5 Experimental setup

We wrote our profile-guided code placement scheme in the Diablo [15] framework. We used the ARM backend for our experiments, run on the XTREM simulator [2] which has been validated against the Intel XScale processor [7].

Table 1. Baseline system configuration.

Parameter	Configuration
Pipeline	7/8 Stages
Functional Units	1 ALU, 1 MAC, 1 Load/Store
Issue	Single Issue, In-Order
Commit	Out-of-Order (Scoreboard)
Memory Bus Width	32 Bit
Memory Latency	50 Cycles
I-TLB, D-TLB	32-Entry Fully Associative
I-Cache, D-Cache	32KB, 32-Way, 32B Block
Data Buffers	32B Fill Buffer (Read) and 16B Write Buffer

An overview of the microarchitectural parameters are given in table 1. We keep all parameters constant and vary the size and associativity of the instruction cache in both the baseline and for our scheme (so we always compare equally configured machines). We describe all changes we make in the sections where we present the results.

We used the MiBench benchmark suite [4] and, for each benchmark, used two inputs: the *small* set for profiling and the *large* inputs for performance and power evaluation. We were unable to run some benchmarks within our environment (*lame*, *mad*, *typeset*, *ghostscript* and *gsm*) because they contain code which is rejected by the recent version of gcc. Other benchmarks we chose to leave out because small (train) and large (test) inputs require different programs to run (*basicmath*, *qsort*, *dijkstra* and *stringsearch*).

In section 6 we show the energy-delay (ED) product to determine the trade-off between performance and energy consumption. This is an important metric in microarchitecture design because it indicates how efficient the processor is at converting energy into speed of operation, the lower the value the better [5].

For comparison in section 6 we use a baseline with no instruction cache modification. We have also implemented the way-memoization scheme from [12]. This approach completely avoids tag lookups in the instruction cache by storing extra information (called links) in the data side of the cache that indicate which way the next access will occur in. The instruction cache we use has a 32B line size, meaning 8 instructions and 9 links in each line. In our initial 32KB, 32-way set-associative cache, each link is 6 bits, meaning a 21% overhead in the data side of the cache for this hardware scheme.

6 Results

We now present the results for our way-placement scheme and compare with a state-of-the-art approach to show that we can provide significant benefits over other techniques. Section 6.1 presents an initial evaluation of our way-placement approach. Section 6.2 considers altering the size of the way-placement area, then section 6.3 applies our scheme to a variety of differing cache sizes and associativities. Section 6.4 gives a summary of our results.

6.1 Initial evaluation

For our initial evaluation we used a 32KB, 32-way instruction cache as in the XScale processor [7]. Our initial way-placement area size is 16KB. There is no change in performance when using either way-placement or way-memoization, hence we show only instruction cache energy and ED product due to space limitations and because the ED product captures the trade-off between performance and energy, as described in section 5.

The instruction cache energy consumed by each benchmark and on average is shown in figure 4(a) where it is immediately clear that our approach saves more energy than way-memoization. In our technique, energy savings approach 50% compared with the way-memoization scheme where 32% of energy is saved. This is because of the overheads in the data side of the instruction cache to store link information that are inherent in this technique.

Figure 4(b) shows the ED product for each benchmark and on average. We achieve an ED product of 0.93 on average with two benchmarks below 0.9. This shows that our scheme saves considerable energy in the instruction cache and overall in the processor, out-performing the state-of-the-art way-memoization approach and producing an impressive ED product value.

6.2 Evaluating way-placement area size

Having shown that our way-placement scheme is beneficial this section explores the effects of altering the size of the way-placement area. As described in section 4.1, we do not have to recompile to perform these alterations. Again we use a 32KB, 32-way instruction cache. Figure 5 shows the results where we vary the way-placement area from 16KB (as in section 6.1) down to just 1KB. Our results are averaged across all benchmarks.

As figure 5(a) shows, significant energy savings occur in the instruction cache with all sizes of way-placement area. Even with just a 1KB way-placement area, energy is reduced to 56% of the baseline. This is still significantly more than the way-memoization scheme which can only reduce it to 68%. The ED product for each way-placement area size is shown in figure 5(b). The ED product for all way-placement sizes is always better than for the way-memoization scheme, achieving 0.94 in the 1KB case.

6.3 Varying cache parameters

We now consider the effects of varying the size and associativity of the instruction cache on our scheme. Figure

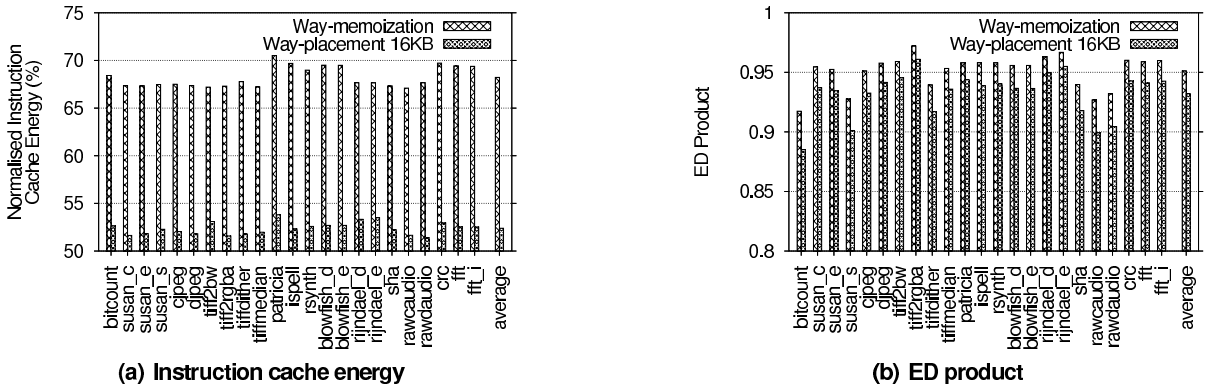

Figure 4. Normalised instruction cache energy and ED product for the way-memoization scheme and our way-placement approach compared to the baseline with no instruction cache optimisations.

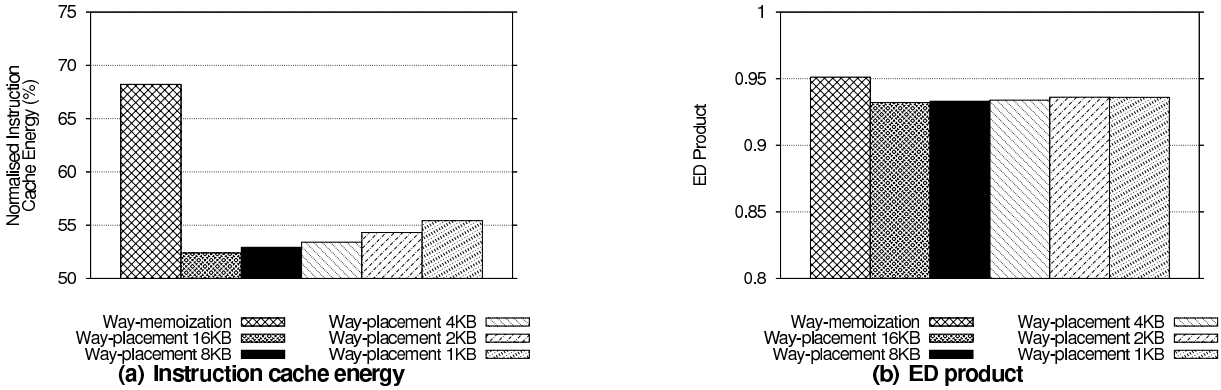

Figure 5. Normalised instruction cache energy and ED product for the way-memoization scheme and our approach with differing way-placement area sizes averaged across all benchmarks.

6(a) shows that our scheme saves considerable instruction cache energy no matter the size or associativity of the cache used. We achieve at least 59% energy savings for all way-placement area sizes for the 64KB, 16-way configuration. In the 16KB, 64-way cache, where the way-memoization performs poorly by increasing cache energy, our schemes all reduce instruction cache energy consumption to 82%.

Figure 6(b) shows the ED product when using these cache configurations. We achieve the best ED product in the 64KB, 16-way instruction cache. This is 0.80 for the 16KB and 8KB way-placement area sizes and 0.81 for the 4KB, 2KB and 1KB sizes. The highest ED product for any of our schemes, which is still better than the baseline and the way-memoization technique, is 0.98 for the 2KB way-placement area size in the 16KB, 64-way instruction cache.

6.4 Summary

We have presented results for our way-placement scheme and shown that it can reduce instruction cache energy and ED product on all cache configurations, outperforming a state-of-the-art hardware approach. We saved almost 50% cache energy in our initial evaluation and in the best case, with a 64KB, 16-way cache we achieved an ED product of 0.80 using a 16KB or 8KB way-placement area.

7 Related work

Several papers have addressed the problem of high instruction cache energy consumption by using an additional buffer or cache between the CPU and instruction cache. [1, 11, 9]. However, these schemes introduce a non-negligible amount of extra hardware in the form of another cache between processor and main instruction cache which can introduce extra fetch latency when a miss occurs.

Ravindran *et al.* [14] proposed a compiler assisted method for dynamically reconfiguring the scratchpad memory content to reduce cache accesses. However, this scheme requires a scratchpad memory to be provided in the proces-

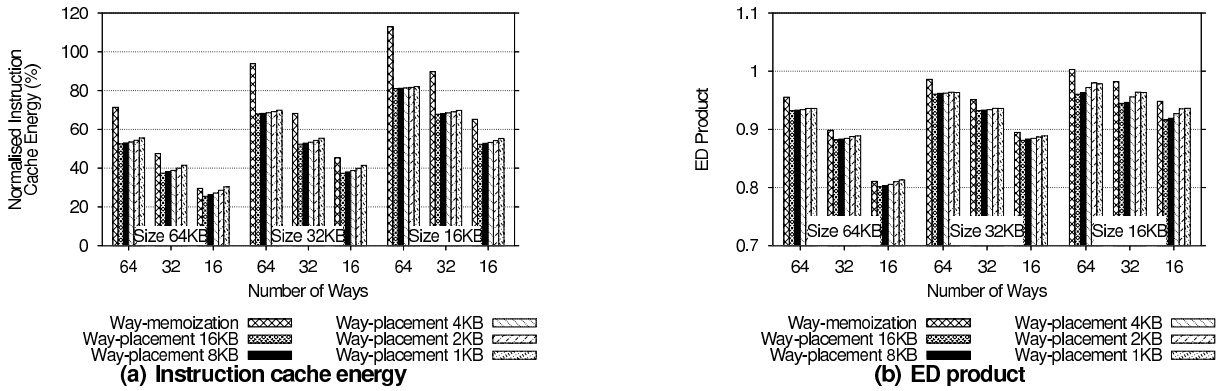

Figure 6. Normalised instruction cache energy and ED product for the way-memoization scheme and our approach with two different way-placement area sizes. We show results for 64KB, 32KB and 16KB caches with 64, 32 and 16-way associativity, averaged across all benchmarks.

sor and would generally only apply to loops. Ishihara *et al.* [8] highlight the benefits of using a non-uniform way-reconfigurable cache in embedded systems. Other proposals aimed to reduce the number of tag checks in an associative cache by predicting the cache way to be accessed [6, 12]. However, for these schemes, incorrect predictions require extra logic for recovery and a performance penalty is incurred.

Further techniques specifically address the problem of leakage power in caches. Flautner *et al.* [3] and Kaxiras *et al.* [10] proposed schemes to selectively place some of the unused cache lines into a low-leakage, state-preserving or switched-off state. They relied on hardware runtime mechanisms to estimate cache line usage. These approaches are orthogonal to our scheme and can therefore be used together for additional energy savings.

8 Conclusions

This paper has presented a novel approach to energy saving in the instruction cache. We place frequently executed portions of the code together in the program binary in a way-placement area which can be set to any size required, even at runtime. Accesses to instructions in this area only need to perform one tag check, saving energy.

We compare our scheme to a state-of-the-art hardware approach and find that we save significantly more energy in the instruction cache. For our initial evaluation we saved almost 50% of the energy compared with 32% for the hardware scheme. We evaluated our approach across a variety of cache sizes and associativities and found that we can always pick a way-placement area size that is beneficial in terms of energy. In the best case we reduce instruction cache energy by 59% and achieve an ED product of 0.80, out-performing the best hardware approach.

Acknowledgements The authors would like to thank Nigel Topham for initial discussions and helpful comments.

References

[1] N. Bellas et al. Architectural and compiler techniques for energy reduction in high-performance microprocessors. *IEEE Transactions on VLSI*, 8(3), 2000.

[2] G. Contreras et al. XTREM: a power simulator for the Intel XScale core. In *LCTES*, 2004.

[3] K. Flautner et al. Drowsy caches: Simple techniques for reducing leakage power. In *ISCA-29*, 2002.

[4] M. R. Guthaus et al. MiBench: A free, commercially representative embedded benchmark suite. In *WWC-4 (MICRO-34)*, 2001.

[5] C.-H. Hsu et al. Towards efficient supercomputing: A quest for the right metric. In *HP-PAC*, 2005.

[6] K. Inoue et al. Way-predicting set-associative cache for high performance and low energy consumption. In *ISLPED*, 1999.

[7] Intel Corporation. Intel XScale microarchitecture. http://www.intel.com/design/intelxscale/.

[8] T. Ishihara et al. A non-uniform cache architecture for low power system design. In *ISLPED*, 2005.

[9] C. Jung et al. Instruction cache organisation for embedded low-power processors. *IEE Electronics Letters*, 37(9), 2001.

[10] S. Kaxiras et al. Cache decay: Exploiting generational behavior to reduce cache leakage power. In *ISCA-28*, 2001.

[11] J. Kin et al. The filter cache: An energy efficient memory structure. In *MICRO-30*, 1997.

[12] A. Ma et al. Way memoization to reduce fetch energy in instruction caches. In *WCED (ISCA-28)*, 2001.

[13] J. Montanaro et al. A 160-MHz, 32-b, 0.5-W CMOS RISC microprocessor. *IEEE JSSC*, 31(1), 1996.

[14] R. A. Ravindran et al. Compiler managed dynamic instruction placement in a low-power code cache. In *CGO*, 2005.

[15] L. Van Put et al. DIABLO: a reliable, retargetable and extensible link-time rewriting framework. In *ISSPIT*, 2005.

[16] M. Zhang et al. Highly-associative caches for low-power processors. In *Koolchips Workshop (MICRO-33)*, 2000.