Power Efficient Instruction Caches for Embedded Systems

Dinesh C. Suresh, Walid A. Najjar, and Jun Yang

Department of Computer Science and Engineering, University of California,
Riverside, CA 92521, USA
{dinesh, najjar, junyang}@cs.ucr.edu

Abstract. Instruction caches typically consume 27% of the total power in modern high-end embedded systems. We propose a compiler-managed instruction store architecture (K-store) that places the computation intensive loops in a scratch-pad like SRAM memory and allocates the remaining instructions to a regular instruction cache. At runtime, execution is switched dynamically between the instructions in the traditional instruction cache and the ones in the K-store, by inserting jump instructions. The necessary jump instructions add 0.038% on an average to the total dynamic instruction count. We compare the performance and energy consumption of our K-store with that of a conventional instruction cache of equal size. When used in lieu of a 8KB, 4-way associative instruction cache, K-store provides 32% reduction in energy and 7% reduction in execution time. Unlike loop caches, K-store maps the frequent code in a reserved address space and hence, it can switch between the kernel memory and the instruction cache without any noticeable performance penalty.

1 Introduction

Low power is a very important design criterion in the design of a very large number of embedded computing systems. Caches consume over 50% of the total energy of an embedded system [11]. The energy consumed by the instruction cache is of particular importance since an instruction is fetched every cycle. While numerous low-power instruction cache designs have been proposed in the literature, recent trends in research [3][7][9] have been directed towards customizing caches for embedded system applications. Examples of such customized instruction cache architecture include loop-cache like architectures [3][4] that place frequently executed loops on a special, smaller sized instruction cache.

It is a well-known observation that a software program spends 90% of its execution time in executing 10% of the code: a feature known as the 90–10 rule. The 90/10 (or 80/20) rule is even more relevant in embedded applications than desktop ones. In one of our previous works [17], we identified and quantified the execution kernels in a large number of embedded programs. The execution kernel is defined as a set of *functions and/or loops* that together account for a substantial percentage of the overall execution time. We found that the execution kernels often possessed a high execution density (execution count per unit size). Table 1 summarizes the kernel sizes for applications

T.D. Hämäläinen et al. (Eds.): SAMOS 2005, LNCS 3553, pp. 182–191, 2005.

Table 1. Kernel and Program sizes, in bytes, and Static and Dynamic contributions of the Kernel for applications in the MediaBench and NetBench suites

Code	Kernel Size (B)	Program Size (B)	Kernel % static	Kernel % dynamic
DRR	740	22511	03.28	45.12
Jpegencode	1996	102975	01.93	54.91
url	1688	11271	14.90	58.29
Unepic	2440	29727	08.20	59.40
Dh	1001	54563	01.83	66.51
Md5	7124	12895	55.24	66.57
G721enc	2470	250439	00.99	67.81
G721dec	2296	250439	00.91	68.67
Mpegencode	1576	96263	01.63	68.75
Tl	2336	20223	11.55	70.70
Mpegdecode	704	68035	01.03	79.12
Crc	584	7483	07.80	87.22
Adpcmencode	916	8091	11.32	96.17
Adpcmencode	1216	8192	14.84	97.12

from the MediaBench [8] and NetBench [10] benchmark suites. From the data reported the vast majority of kernels were less than 4KB in size and most of these were less than 2KB which is well within the size of a scratchpad memory as is commonly available in embedded processors [5]. Table 1 also shows the percentage contribution of the kernel to the total program size (static) and to the execution time (dynamic).

Let us consider the frequently executed code for the Diffie-Hellman Key exchange (DH) application [10] shown in Table 1. The kernel for this application constitutes 66.51% of the total dynamic instruction count and it contains a most frequent function (NN_DigitMult) that takes up nearly 35% of the total execution time. Hence, any loop-cache like architecture executing such applications must cache both loops and function calls. Accommodating such functions in loop-cache like architecture often increases the size requirement of a loop cache. Scratchpad memories take up much lesser area and consume nearly 40% lesser power than instruction caches of equal size [2] and consequently, they are ideal alternatives to loop caches.

In this paper we propose an instruction store architecture that is designed to exploit specific features of execution kernels in embedded applications. We call it the Kernel Store or K-Store. The main idea in the K-store is that a scratchpad like memory is used to store the execution kernel (*both loops and functions*) of the application: This kernel memory is therefore a fast SRAM memory at the level of the cache that can be accessed in one cycle. Unlike the cache it does not have to support tag arrays. The remainder of the application is stored in main memory and is accessed in the traditional way via an instruction cache. The system software is modified to support the kernel memory by inserting jumps where appropriate in the code. The compiler maps the kernel instructions to a separate region in the address space and hence, facilitates easy detection of these instructions during run-time.

Traditional loop cache architectures [3][4][9] cannot store frequently executed function calls inside the loop cache. K-Store overcomes this limitation by caching the kernel code (both functions and loops) in a tagless scratchpad memory. A 8KB direct-mapped basic K-store provides 28% reduction in energy while a 8KB, direct mapped supplemental K-Store provides 32% reduction in energy when used in lieu of a conventional instruction cache of equal size.

The rest of this paper is organized as follows. Section 2 illustrates the design of our K-Store architecture. In section 3, we describe our experimental framework. In Section 4, we discuss the results and evaluate the energy and performance of K-Store architecture. We provide a list of related research work in section 5. In section 6, we present concluding remarks.

2 K-Store Design

The K-Store architecture consists of two components: the kernel memory and the instruction cache. Figure 1 shows the block diagram of our K-Store architecture: The K-Store consists of both the kernel memory and the instruction cache. During run-time, instruction block requests are intercepted by a logic circuit, which identifies whether the requested address belongs to the kernel space or the non-kernel space.

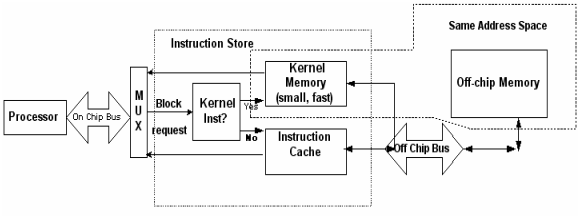

Fig. 1. K-store Architecture

By examining just a few bits of the instruction address during program execution, we can determine whether the instruction address lies in the kernel space or in the non-kernel space. For example, let us assume that for a given application, the compiler has mapped the kernel code in the address range of 4096 to 8192 bytes. We need a circuit to identify these kernel instructions at runtime. Figure 2 illustrates the operation of a logic circuit used to identify kernel instructions in this case. Here, the bit values in bit positions 1–12 are not useful in identifying the kernel instructions (don't care). When the 13^{th} LSB is high and all the remaining bits from the 14^{th} to the Most Significant Bit (MSB) are zeros, we can conclude that the instruction is a kernel instruction. A low value in all of the bit positions from 14^{th} to the MSB can be easily detected through the use of a five-levels of 2-input nor gate. While using 0.18-micron process technology, the access time of an 8k, 4-way set associative cache with block size of 32 bytes, as obtained using the CACTI tool [18], is 1.28 ns. A five level gate delay in 0.18-micron process technology typically amounts to 0.3 ns [14]. Hence, for a 500 MHz system (cycle time = 2 ns), the kernel detection logic can be easily accommodated within the same cycle. As shown earlier in Table 1, most of the kernel sizes are less than 4k and hence, checking for a high value on the 13^{th} or 14^{th} bit is sufficient to accommodate most of the application kernels. The K-store architecture is ideally suited for systems without a virtual memory.

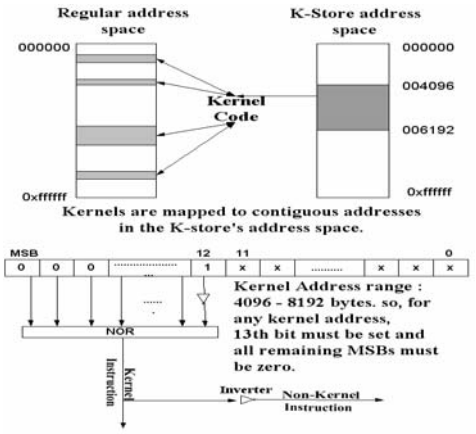

Fig. 2. Runtime identification of kernel instructions

As shown in Fig. 2, the compiler maps the kernel instructions to a reserved area in the off-chip address space. Occurrences of the kernel code in the original program can be replaced by jump instructions that transfer the control flow to the kernel address space. Upon completing the execution of the kernel code, we need a jump instruction to continue program execution in the non-kernel address space. Thus every call to the kernel code adds two additional control transfer instructions to the program. Using our simulator, we measured the total increase in the number of control transfer instructions in the program. As shown in Table 2, we found that the control transfer instructions increased the dynamic instruction count by 0.039% on an average.

In order to better understand the cache behavior of the non-kernel portions of a program, we investigated the use of kernel memory with varying cache configurations. To ensure a fair comparison, we restricted the size of the k-store's instruction cache so that the total size of the kernel memory and the k-store's I-cache was equal to that

Table 2. Additional jump instructions to map kernel instructions of different programs

Benchmark	Additional dynamic control Instructions	Dynamic Instruction Count	% of additional Instructions
Adpcmencode	148	31481991	0.00047
G721decode	1595527	1005741879	0.158642
G721encode	1907982	1068726694	0.1785
Jpegencode	20455	81017999	0.02547
Mpegdecode	578160	1020616339	0.056648
Mpegencode	2344064	7037415745	0.033309
Unepic	236	30588223	0.000772
CRC	1256	18524458	0.00678
Dh	452608	12450027332	0.0003635
Drr	65	16266546	0.0004
MD5	59844	371031482	0.01614
Tl	597	2054152	0.029063
url	60330	1426337205	0.00423
Average	540097	1889217696	0.0392

of the baseline cache. By doing so, the K-store's instruction has a smaller size than the baseline cache and consequently, all accesses to the K-store are serviced at lesser power when compared to that of a baseline cache. We call these k-store cache configurations as basic K-store.

We also investigate the use of the kernel memory as a supplement to a baseline cache. Hence, we picked a baseline cache and added a small kernel memory to it and observed the energy reduction in this case. In spite of the higher cost associated with this design, the number of off-chip accesses would be much lesser than that of the baseline cache and hence, this design should be highly energy-efficient. For the rest of this paper, we will refer to this K-store configuration as supplemental K-store. We will discuss our experimental setup in the following section.

3 Experimental Framework

We analyzed an extensive collection of embedded system benchmarks from the NetBench [10](CRC, MD5, DH, DRR, TL and URL) and the MediaBench [8] (ADPCM, JPEG, MPEG and G721) benchmark suites. Table 3 gives a brief description about the benchmarks used in our experiments. For each of these applications, we used our loop analysis software [17] to identify the time consuming loops and function calls. We then identified the kernel instructions in these benchmarks and we extended the Sim-cache simulator supplied with the Simplescalar tool set [15] in order to simulate our design

We calculate the energy savings using the following formula:

$$\mathbf{Energy}_{K-Store} = \mathbf{Energy}_{Kernel} + \mathbf{Energy}_{Cache} + \mathbf{Energy}_{Off-Chip}$$

The total number of accesses to each of these components are obtained from the *s*im-cache simulator [15] we obtained the energy per kernel memory access and energy per cache access from the CACTI tool [18]. We investigate the use of K-store in two different configurations – basic K-store and the supplemental K-store. For our instruction store design, we vary the kernel memory size (2K, 4K), instruction cache size (2K, 4K), associativity (Direct, 4-way), off-chip miss penalty (20, 40, 60, 100, 200 cycles) and we evaluate the impact of these parameters on the energy savings and memory access latency. For supplemental K-store, we use a kernel memory of size 1KB and compare our K-store design with a conventional instruction cache. For each of these configurations, we fixed the cache block size at 32 bytes. We used 0.18-micron process technology in our power model. In the following section, we present the energy and performance results for our K-Store architecture.

Table 3. Average normalized energy and memory cycle reduction for different cache configurations

Cache Configuration	Energy Reduction	Cycles Reduction
4K Direct mapped	21%	3.6%
4K-4way set assoc.	20%	3%
8K-Direct mapped	28%	0.5%
8K-4way set assoc.	25%	0.1%

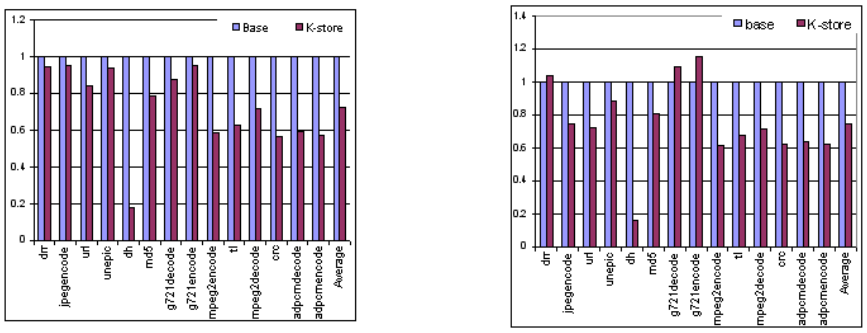

Fig. 3. Normalized energy consumption for (a) a direct mapped cache and (b) a 4-way set associative cache; Baseline= 8k, K-store: Kernel = 4k, Cache = 4k

4 Energy and Performance Evaluation

Basic K-Store. Figure 3a shows the normalized energy consumption for a 8Kb, direct mapped cache. The K-store uses kernel and cache memories of size 4Kb each. We find that K-Store provides an average energy reduction of 28% over a conventional instruction cache. Figure 3b shows the normalized energy consumption for a 8Kb, 4-way set associative cache. The K-store uses kernel memories and cache memories of size 4Kb each. In these graphs, we find that for the *Diffie-Hellman* key exchange application (*dh*), the energy savings is much higher than the rest. This is due to the fact that the *dh* kernel, has an extremely small static size and still contributes towards 66% of the total execution time. Applications like *adpcm* and *crc* are also characterized by very small static size and high execution count when compared to other applications under consideration. Hence, they yield significant energy savings.

Figure 4 shows the normalized cycle time for a 8Kb, direct mapped cache. The K-store uses kernel and cache memories of size 4Kb each. In spite of providing high-energy savings, the cycle time reduction for the *Adpcm* application is not so significant when compared to other applications. *Adpcm's* code size is comparable to the base cache size (8KB) and hence, the base cache provides a higher hit rate than the K-store's instruction cache (4KB). On an average, basic K-store yields a 0.5% reduction in the overall execution cycles.

We explored the design space to find out the optimal sizes of kernel memory and instruction caches for our instruction store architecture. In Table 3, we show the average normalized values of energy and memory cycle reduction for each of the cache configurations. For the results shown in Figures 3–6 and in Table 3, we assumed that an off-chip access was 60 times more expensive than an on-chip access. For each of the cache configurations, we also computed the values of energy savings and execution time reduction for varying values of off-chip penalties. Figure 5 provides the execution time reduction and the average energy savings for varying values of off-chip penalties.

The energy savings in the kernel memory are obtained due to two reasons: the smaller size of the kernel memory and the simple addressing scheme. Since the kernel memory is tagless, the tag comparison power, which contributes to 35% of the cache

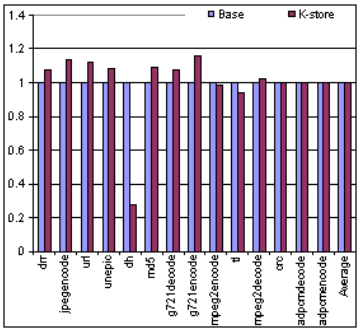

Fig. 4. Normalized execution time for a direct- mapped cache; Baseline = 8K, K-store: Kernel memory = 4K, cache = 4K

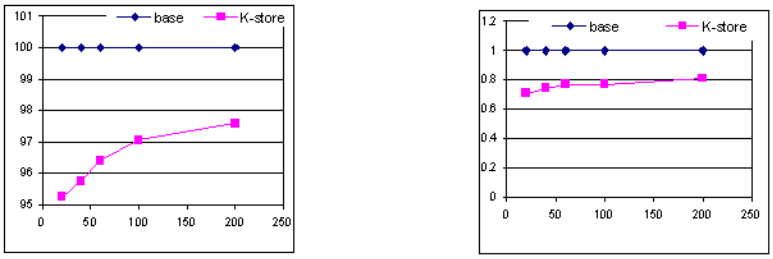

Fig. 5. Percentage reduction in (a) memory cycles and (b) energy for varying values of off-chip penalty

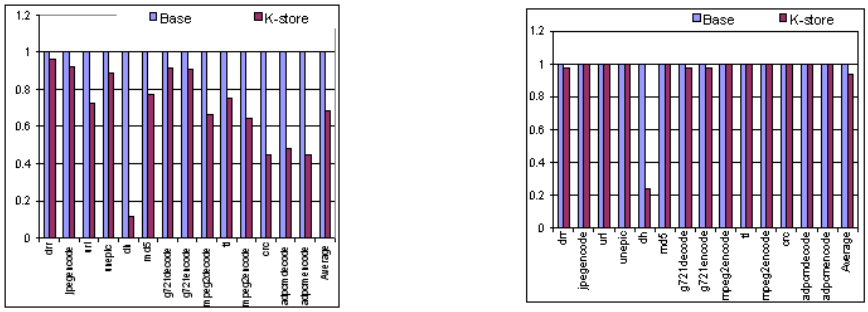

Fig. 6. Normalized (a) energy consumption and (b) execution cycles for a direct mapped cache; Baseline = 8K, K-store : Kernel memory = 1K, cache = 8K

power, is no longer necessary for a substantial fraction of the executed instructions. A smaller sized cache services the non-kernel instructions and hence, reduction in cache size is one of the main reasons for the energy savings reported in this paper. On an average, we found that reduction in cache size alone, accounted for 13% of the energy savings.

Supplemental K-Store. We also added the kernel memory to the best baseline cache configuration and evaluated the resultant energy and performance benefits. We found that when a 8KB, direct mapped instruction cache was augmented with a 1KB kernel memory, we achieved 32% reduction in energy and 7% reduction in execution time. The results are shown in Fig. 6. Since the instruction cache size is the same in the baseline cache and the K-store, K-store has fewer misses and hence, results in higher energy savings.

5 Related Work

Several techniques have been proposed to reduce the energy dissipation in the instruction caches of embedded processors. Many of these methods involve the usage of a tiny cache as a supplement to an existing instruction cache [9][2][7][13][19]. The tiny caches are designed in such a way that they exploit some feature of the application in order to capture most of the processor requests. By making sure that these smaller caches service bulk of the access request, significant amount of energy can be saved on each cache access.

Banakar et al. [2] have reported that a scratch-pad memory takes up 34% lesser area, consumes 40% less power and lowers cycle time by 18% when used in lieu of a cache of equal size. The power, cost and performance advantages provided by scratch-pad memories make them ideal candidates for replacing conventional caches. Researchers have rigorously investigated the use of a scratch pad memory to hold frequently used data items. Panda [12], Kandemir [6], and Avissar [1] have explored the use of scratch pad memory as a supplement for traditional cache architectures. They placed a small, fast, memory at the level of L1 cache and they use this memory to hold the most frequently used data items. Thus the off-chip traffic due to misses is reduced.

Avissar et al. [1] proposed an automatic compiler management strategy for data allocation amongst heterogeneous memory units. Panda [12] minimized the cross interference between different variables in the data cache by mapping them onto scratch pad memory and DRAMS. Sjodin [16] proposed a method wherein the critical variables with large number of accesses are stored on an on-chip SRAM while less critical variables allocated to a slower external RAM. Kandemir et al [6] proposed a compiler-directed on-chip software management strategy for data accesses. They store the reusable data values in nested loops onto an on-chip SRAM, thereby minimizing the data transfer between the off-chip memory and the on-chip scratch pad memory. K-Store uses the scratch pad memory to hold the frequently executed instructions and is hence, orthogonal to the works mentioned above, which focus on data storage.

Bellas et al. [4] have proposed the use of an L0 cache that resides between the CPU and the L1-cache. The compiler selects a few basic blocks to be placed in the L0-cache. Statically loaded loop caches (SLLC) [2] exploit compile-time information to preload the caches with the instructions of one frequently executed loop and hence, reduce the cold start misses.

K-Store is different from L0-cache [4] and SLLC [3] in the following aspects. Even though the SLLC and L0-cache exploit compile-time information, they can only extract simple tight loops that contain no function calls. This is due to the limitation of their

access mechanism – by testing if the current PC falls within the range of the beginning and the ending addresses of the loops, the control logic decides whether the current instruction is a loop instruction stored in the SLLC or the L-cache. Such a mechanism excludes those loops containing function calls. While in the K-Store architecture, there is no such restriction since the entire loop body is moved to a different memory address space, simplifying the detection of the kernel instructions and increasing the number of candidate kernels. Besides, preloaded loop caches are architecturally more complex than scratch-pad memories.

Cache Aware Scratchpad Allocation (CASA) [20] provides a sophisticated technique for analyzing conflicts within the instruction cache and reduces these conflicts by using the scratchpad. By using the scratchpad to store both kernel blocks and conflicting instruction blocks, CASA can be effectively combined with a K-store to achieve significant energy benefits.

In the wake of aforementioned discussion, our contributions in this paper can be summarized as follows: We propose that program segments with high execution density (*both loops and functions*) should be held in a scratch pad memory (Kernel memory) and the remaining instructions can be efficiently cached in a regular instruction cache. In order to facilitate easy and non-intrusive detection of kernel instructions, we map the kernel instructions to a separate region in the off-chip address space. We illustrate that our approach is highly energy efficient.

6 Conclusion

Most of the embedded system applications tend to have strong kernels, which are instruction blocks with high execution count and low static size. In this paper, we propose a compiler-managed instruction store architecture that exploits kernel features to provide energy and performance benefits. Our compiler-assisted instruction store places the computationally intensive kernel code (*functions and loops*) onto a small, fast *scratch-pad memory* (*kernel memory*) and allocates the remaining instruction blocks to a regular instruction cache. 8Kb direct mapped supplemental K-store provides 32% reduction in energy and 7% reduction in execution time when used in lieu of a direct mapped cache of equal size.

References

1. Avissar, O., Barua, R., Stewart, D.: An Optimal Allocation for Scratch-Pad Based Embedded Systems. ACM Trans. on Embedded Computing Systems **1** (2002) 6–26
2. Banakar, R., Steinke, S., Lee, B.-S., Balakrishnan, M., Marwedel, P.: Scratchpad Memory: A Design Alternative for Cache On-chip memory in Embedded Systems. In: Proceedings of the 10th Int. Workshop on Hardware/Software Codesign, Estes Park, CO (2002)
3. Cotterell, S., Vahid, F.: Tuning of Loop Cache Architectures to Programs in Embedded Systems Design. In: IEEE/ACM Int. Symp. on System Synthesis (2002) 8–13
4. Bellas, N., Hajj, I., Polychronopoulos, C., Stamoulis, G.: Energy and Performance Improvements in Microprocessor Design Using a Loop Cache. In: Int. Conf. on Computer Design (1999) 378–383

5. Intel Corp. Intel XScale (tm) Core Developer's Manual, 2002. http://developer.intel.com/design/intelxscale/.
6. Kandemir, M., Kadayif, I., Sezer, U.: Exploiting Scratch-Pad Memory Using Presburger Formulas. In: Int. Symp. on System Synthesis, Montreal, Canada (2001) 7-12
7. Kin, J., M. Gupta, M., Mangione-Smith, W.H.: The Filter Cache: An Energy Efficient Memory Architecture. In: the 30th Annual IEEE/ACM Symp. on Micro Architecture
8. Lee, C., Potkonjak, M., Smith, W.H.: MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In: Int. Symp. on Microarchitecture Research Triangle Park, NC (1997) 292–303
9. Lee, L., Moyer, B., Arends, J.: Instruction Fetch Energy Reduction Using Loop Caches for Embedded Applications with small tight loops. In: Int. Symp. on Low Power Design (1999)
10. Memik, G., Smith, W.H., Hu, W.: NetBench: A Benchmarking suite for Network processors. In: Proc. of Int. Conf. on Computer-Aided Design (ICCAD), San Jose, CA (2001) 39–42
11. Montanaro, J., et al.: A 160MHz, 32b, 0.5W CMOS RISC Microprocessor. IEEE Journal of Solid State Circuits (1996) 1703–1714
12. Panda, P.R., N.D. Dutt, N.D., Nicolau, A.: Efficient Utilization of Scratch-Pad Memory in Embedded Processor applications. In: Proc. of European Design and Test Conf., Paris (1997)
13. Ravindran, R., Nagarkar, P.D., Dashika, G.S., Marsman, E.D., Senger, R.M., Mahlke, S.A., Brown, R.: Compiler Managed Dynamic Instruction Placement in a Low-Power Code Cache. In: Proc. of the 3rd Intl. Symp. on Code Generation and Optimization (CGO) (2005)
14. http://www.semicon.toshiba.co.jp/eng/prd/asic/topix.html
15. Simplescalar Simulator. http://www.simplescalar.com
16. Sjodin, J., Von Platen, C.: Storage Allocation for Embedded Processors. In: International Conference on Compiler, Architecture and Synthesis for Embedded Systems (CASES 2001), Atlanta, GA (2001)
17. Suresh, D.C., Najjar, W.A., Vahid, F., Villarreal, J., Stitt, G.: Profiling Tools for Hardware/Software Partitioning of Embedded Systems. In: Proc. of ACM SIGPLAN conference of Language Compilers and Tools for Embedded Systems (LCTES), San Diego, CA (2003) 189–198
18. Steven, J. Wilton, E., Jouppi, N.P.: CACTI: An Enhanced Cache Access and Cycle Time Model. IEEE Journal of Solid State Circuits, **31** (1996) 677–688
19. Tang, W., Gupta, R., Nicolau, A.: Power Savings in Embedded Processors through Decode Filter Cache. In: Proceedings of the Design Automation and Test in Europe (2002)
20. Verma, M., Wehmeyer, L., Marwedel, P.: Cache-Aware Scratchpad Allocation Algorithm. In: Design Automation and Test in Europe (DATE), Paris, France (2004)