Block Cache for Embedded Systems

Dominic Hillenbrand Jörg Henkel

Chair for Embedded Systems (CES)
University of Karlsruhe (TH)
e-mail: {hillenbrand,henkel}@informatik.uni-karlsruhe.de

Abstract— On chip memories provide fast and energy efficient storage for code and data in comparison to caches or external memories. We present techniques and algorithms that allow for an automated use of on chip memory for code blocks of instructions which are dynamically scheduled at runtime to increase performance and reduce power consumption.

I INTRODUCTION AND RELATED WORK

In the light of increasing area, big on chip memories are becoming more and more affordable. In coming multi-core systems they are also a necessity to cope with the increasing memory bandwidth pressure towards external memories. We show that block caches alone can already outperform instruction caches of the same size and provide initial data and insights into the automated use of block caches and their respective on- and offline phases.

Energy and power dissipation constrain clock rates in embedded systems. However Moore's law still holds true and therefore available area will increase for the foreseeable future providing new opportunities to the designer. Internet enabled, wireless consumer gadgets e.g., deploy multi-core systems [1, 2] to securely deliver multimedia content in real time to its users. Typical workloads encompass video encoding/decoding, packet handling, wireless communication and cryptographic protocols simultaneously. SoCs such as the NEC mobile phone application processor MP211 statically assign specific tasks to five dedicated cores. A significant amount of power is consumed in the memory hierarchy. Especially off-chip DRAM accesses are costly in power. For performance reasons off-chip accesses are usually cached in instruction caches which consume more power and take more area than on chip memory [3, 4, 5] which we use for our block cache. On chip memories - also called scratchpad memory - are already a commodity in modern SoC designs. They are typically used to keep computational intensive code e.g. multimedia or cryptographic algorithms close to processing elements. On chip memories are not yet big enough to accommodate all application code. Therefore a designer must identify and prepare a set of performance critical system components for on chip usage. For todays multi-megabyte software systems this is no longer feasible. Previous work focused on static code placement [5] in on chip memories. To automatically identify worthy code portions (hot spots) profiling was used. However profiling may miss or underestimate the importance of code portions. An extension of [5], [6] places copying procedures in front of statically selected code to use on chip memory more efficiently. Additional care has to be taken that the copying costs amortize. This approach still lacks caching capability and adaptability. The copying code also increases the code size, for some benchmarks as high as four-fold [7]. The authors of [8] propose an integrated hardware / software approach for on chip data management (only). For efficient transfers from external to on chip memory DMA is used to reduce copying cost, subsequently lowering the amortization bound. In [9] a MMU is used to page code between off- and on chip memory. The pageable code resides in a special memory region. Additionally, there are two further regions which are either cached by a direct mapped instruction cache or not. The memory region partitioning is not runtime adaptable.

The rest of this paper is organized as follows. In Section II, we present our novel contributions. In Section III, we introduce our workflow, algorithm and design considerations. Experimental results are presented in Section IV-B, and we finally conclude in Section IX.

II OUR CONTRIBUTION

1) We have designed a new runtime adaptable instruction (block) cache which leverages on chip memory to increase performance, lower power consumption using less area in comparison to instruction caches.
2) The challenge of efficient offline code block composition is handled by our algorithm.
3) The effects of different block cache parameters, such as block size and online management strategies have been analyzed.

Hence, our block cache allows (increasing) on chip memories to be efficiently used, is scaleable in size and allows offline knowledge to be exploited in our algorithm. Besides these unique characteristics, our approach does not require source code access or compiler modifications. Re-linking is sufficient. Our tools can take gcc generated ELF-binaries and the resulting binaries are still executable on systems without block cache. Furthermore, the concept, the algorithms and our tools are instruction set independent.

III WORKFLOW

The designer must create function call traces of individual system components e.g. jpeg compression, so that a call graph can be constructed (Figure 1). The function call traces can be obtained by instrumenting or monitoring inside a simulator or on the target hardware. Our block composition algorithm takes

978-1-4244-1922-7/08/$25.00 ©2008 IEEE

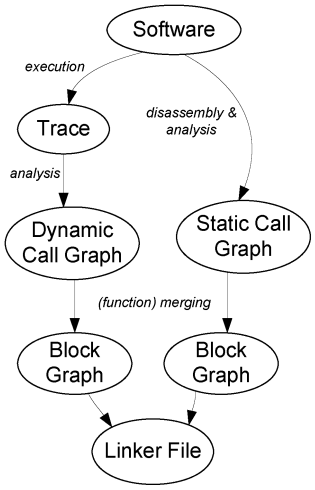

Fig. 1. Workflow, block building

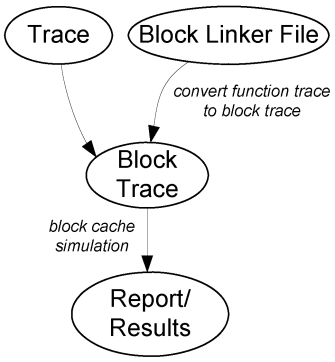

Fig. 2. Workflow, block cache simulation

the (dynamic) function call graph as input and merges functions into groups/ blocks of functions until the (block) size limit prevents further inclusions. The block size is chosen by the designer. Functions that are bigger than a single block need to be split or the block size needs to be increased. Function splitting can already be done by the compiler. After block composition a block linker file is created from the block graph. It states which functions go into a distinct block. The re-linker processes the input binary accordingly. The restructured binary is slightly bigger because most blocks will have some space left.

To cover functions not included in the execution trace we disassemble the binary and analyze it to derive a static function call graph which is merged to the dynamic block graph before the block linker file is generated.

In the next step (Figure 2) the designer can assess the performance of the block composition by feeding function call traces into our block cache simulator. Therefore he needs the previously determined block linker file. In the block cache simulator the number of simultaneously available code blocks (called block slots) and the online eviction strategy can be evaluated. For different block sizes, the composition process needs to be repeated. The overall cache size is the block size multiplied by the number of block slots.

Program III.1 Code merging algorithm

```
cfg_to_block_graph() {
  combine_neigbors() // first step
  merge_direct_children() // second step
  bubble_merge() // third & last step
}

combine_neigbors(graph) {
  while(success) {
    compute_centrality(graph);
    normalize_centrality(graph);
    Node n1 = select_node_with_centrality(graph,1.0);
    Node n2 = max(edge_weights(children(n1)),
                  edge_weights(parents(n1));
    if (n1 && n2) merge(n1,n2); else return;
  }
}

merge_direct_children(graph) {
  compute_centrality(graph);
  normalize_centrality(graph);

  foreach(Node n in centrality_sorted(graph):
          with_more_than_one_child) {
    c = sort_by_edge_weight(children(n));
    merge_until_block_size_reached(c);
  }
}

bubble_merge(Graph g,Node n,p
             parents of n) {
 walk_entire_graph(graph in post_order,
                   Node n)
  {
    foreach(p) { if merge(p,n) return; }
    recurse(parent p' of p,n) {
      if merge(p',n) exit;
    }
    recurse(children c' of p,n) {
     if merge(c',n) exit;
    }
    recurse(root r' of graph,n) {
      if merge(r',n) exit;
    }
  } // end walk_entire_graph
}
```

Benchmark	Code size of executed functions [Byte]	Average function size [Byte]	Number of functions
CJPEG	26935	216	125
MPEGDEC	27766	309	90
WGET	39185	180	219

TABLE I
BENCHMARKS: CODE AND FUNCTION SIZES

A MERGING ALGORITHM FOR DYNAMIC FUNCTION CALL GRAPHS

The merging algorithm uses the greedy strategy to reduce control flow transfer (function calls) between blocks to a minimum in order to avoid block cache misses. The frequency of control flow transfer is derived from the function call trace.

The pseudo code for the dynamic graph merging algorithm is shown in Figure III.1. **Step1**: For every node in the call graph the centrality measure is computed (compute_centrality) which considers the in- and outgoing edge weights which are the calling frequencies. Then the node with highest centrality measure is merged with the neighbor it shares the highest edge weight (combine_neigbors). This is repeated until no more functions can be merged into blocks (size limit). The first step only merges directly linked nodes/functions. **Step2**: In the next step (merge_direct_children) the children of every node are merged in sequence of their centrality measure. **Step3**: In the last step (bubble_merge) the algorithm goes even further and considers distant nodes. First, the parents' children are considered. Then the parents' parents are recursively considered. Then the children of the children and as a last resort a recursive decent from the root node takes all other nodes of the graph into consideration. Finally the block linker file is created.

The graph started out as function call graph and is converted into a block call graph by merging. The designer specified block size has to be obeyed within each step.

B DESIGN CONSIDERATIONS IN BLOCK COMPOSITION

Both run-time trace and static analysis are limited in accuracy. For static analysis meta-information such as relocation and symbol tables can be used to create an approximate call graph. The results however cannot be relied on if function pointers are passed around in data structures. This is quite common in the C-language to implement software interfaces. Therefore the designer needs to adjust the component's parameters and input data carefully, possibly even merging and weighting multiple execution traces to finally obtain an execution trace close enough to the ones encountered in the field. So that all frequently executed code portions are included in the dynamic call graph. The merging algorithm exploits the knowledge about function calling relations to group strongly related functions into one block. This is important because every call outside a block may require a new block to be fetched and another to be evicted. The block size determines the latency encountered for every miss in our block cache.

One has to keep in mind that a good block composition and a sufficiently sized block cache cause the system to spend most of the time executing code from on chip memory.

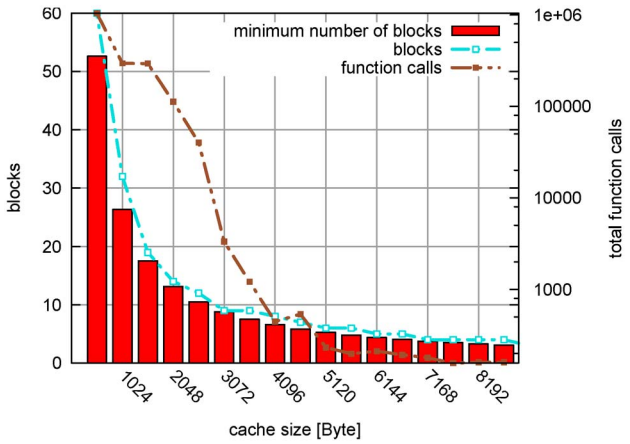

Fig. 3. Internal block fragmentation in CJPEG

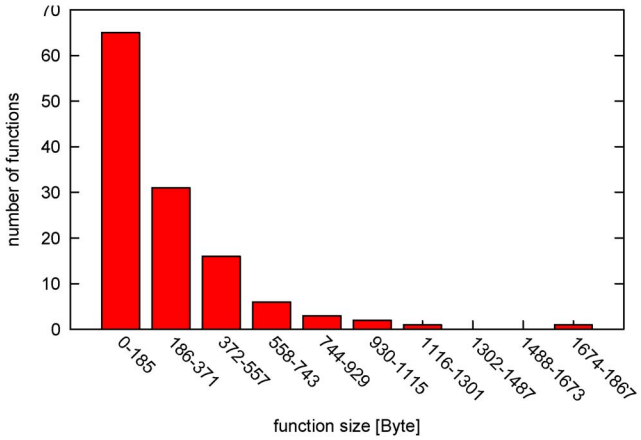

Fig. 4. Function size distribution in CJPEG

C RESULTS OF BLOCK COMPOSITION

Figure 3 shows the (theoretical) minimum amount of blocks, the amount of attained and the number of functional calls between blocks in total for different block sizes (512-8192 bytes) in CJPEG. This benchmark has been used throughout the paper. The other benchmarks (Table I) perform quite similar and would convey little additional value and are therefore omitted.

The merging algorithm presented in A reduces inter-block calls exponentially for block sizes up to 4096 bytes.

The minimum amount of blocks can be achieved if (internal) fragmentation is non-existent. In configurations with small block sizes (<1024 byte) internal fragmentation is more severe than in bigger blocks. This can be explained by the relation between block size and average function size (Table I) which is 216 bytes for CJPEG. The function size distribution is shown in Figure 4. Most functions in CJPEG are already smaller than 512 bytes, the smallest block size we chose in our benchmarks. Bigger functions were split into smaller ones to compose blocks with a maximum size of 512 bytes.

The function sizes are given for the Intel architecture (32bit, x86) which has been used for all simulations. Other than different function sizes there is nothing which is architecture depended in our analysis tools and in block composition. If a designer wants to compose code blocks for the ARM/THUMB architecture e.g. he/she only needs a (cross) compiled binary

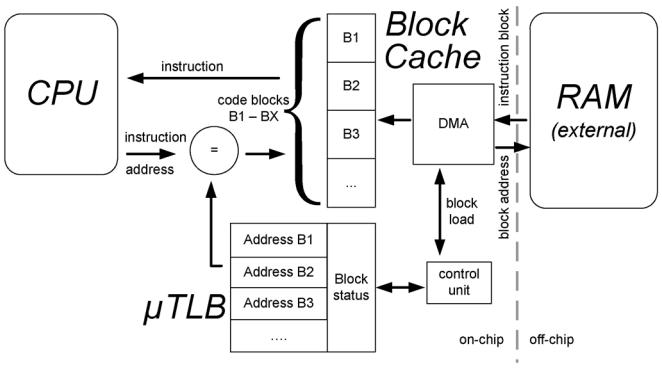

Fig. 5. High level hardware overview

	Control Logic Area		
	Slices	Equivalent gates	Percentage
32 Bit DSP-Processor (OR1200)	4530	10192	89.9
DMA / Control Unit	492	1107	9.8
TLB	16	36	.30

TABLE II
AREA IN A FPGA IMPLEMENTATION

to determine the architecture specific function sizes in order to build a new block composition. The function trace can be taken from the Intel architecture because it is application specific. This allows different architectures to be analyzed without actually running code for them.

IV HARDWARE OVERVIEW

Figure 5 shows a high level hardware overview. The number of block slots need to be fixed at design time and determine along with the block size the overall cache size. The fixed block size has been specifically chosen with an efficient hardware implementation in mind.

The block cache uses a μTLB [9] to map code blocks into on chip memory (a MMU is optional). It is also necessary to detect function calls/returns to other blocks efficiently. Software based approaches [10, 11] suffer from complexity and need time consuming checks to catch function calls/returns to unloaded/evicted code.

The DMA unit transfers new blocks from external memory to the on chip memory. It exploits burst transfers in the external memory to transfer blocks and is therefore much faster than a software copy implementation. The copying and eviction process is controlled by the block cache control unit.

A HARDWARE COSTS

The area for processor, DMA/control unit and μTLB are shown in Table II (FPGA implementation). The majority of the area almost 90% - is taken by the processor.

Performance wise a block cache miss causes the processor to stall until the requested block is in place. The time for a miss is fixed and depends on the time necessary for the block transfer.

V EVICTION STRATEGIES

Every time a new block is loaded into on chip memory, the block cache needs to evict an old block. In [9] a round robin (RR/FIFO) algorithm has been implemented in hardware. We evaluated the performance of RR, LRU and ARC (Adaptive Cache Replacement) [12]. For reference we use the (optimal) Belady algorithm which requires future knowledge to base its decisions.

Block size [Byte]	Block cache size for 6,8 and 12 slots [Byte]			Instruction Cache
	6 slots	8 slots	12 slots	
512	3072	4096	6144	512
1024	6144	8192	12288	1024
1536	9216	12288	18432	
2048	12288	16384	24576	2048
2560	15360	20480	30720	
3072	18432	24576	36864	
...	.	.	.	4096
8192	49152	65536	98304	8192-131072

TABLE III
CACHE SIZES

LRU is well researched algorithm and myriads of improved versions have been published. Only one deficiency shall be mentioned here: in loops LRU tends to overwrite old entries which are subsequently needed, in case cache size and loop access patterns mix poorly. Under such circumstances it is better to keep some entries fixed for the time being. However, if such situations arise in our block cache it can be a good indication for a too small cache and/or a suboptimal block composition. None of our benchmark suffered from this phenomenon because looping mostly takes place inside blocks. In CJPEG loops never spanned more than four blocks. Our smallest block cache already uses 6 block slots which are more than sufficient.

ARC balances recency and frequency by dynamically dividing the cache size into recent and frequent accesses. ARC is scan-resistant, meaning that one time access sequences have limited impact in terms of cache pollution.

FIFO (First in first out, round robin) is easy to implement in hardware but suffers from Beladys anomaly meaning that bigger caches can lead to worse performance.

VI FUNCTION CACHE

Additionally we are also interested in comparing the block cache with a fictional function cache of the same size. A function cache caches at function granularity instead of block granularity. A function cache therefore can approximate the working set more precisely than a block cache. Ideally a function cache would be used for on chip memory handling. Unfortunately the runtime costs for dynamic function relocation is too high. Such work is usually done offline by a linker.

The function cache uses LRU for eviction because it is also the algorithm we favored for hardware-implementation.

The block cache in comparison is a trade-off between flexibility and effort. The usage of block slots allows runtime adaptation similar to the function cache. This also increases robustness.

VII INSTRUCTION CACHE

For assessing the relative performance of our block cache, we use a 4x associative instruction cache with a 32 byte cache line. The code cache size always grows to the power of two and its size is shown Table III along with the sizes of different block caches.

For block caches the size depends on the number of block slots and the block size. We increased the block size in 512 byte steps until 8192 bytes.

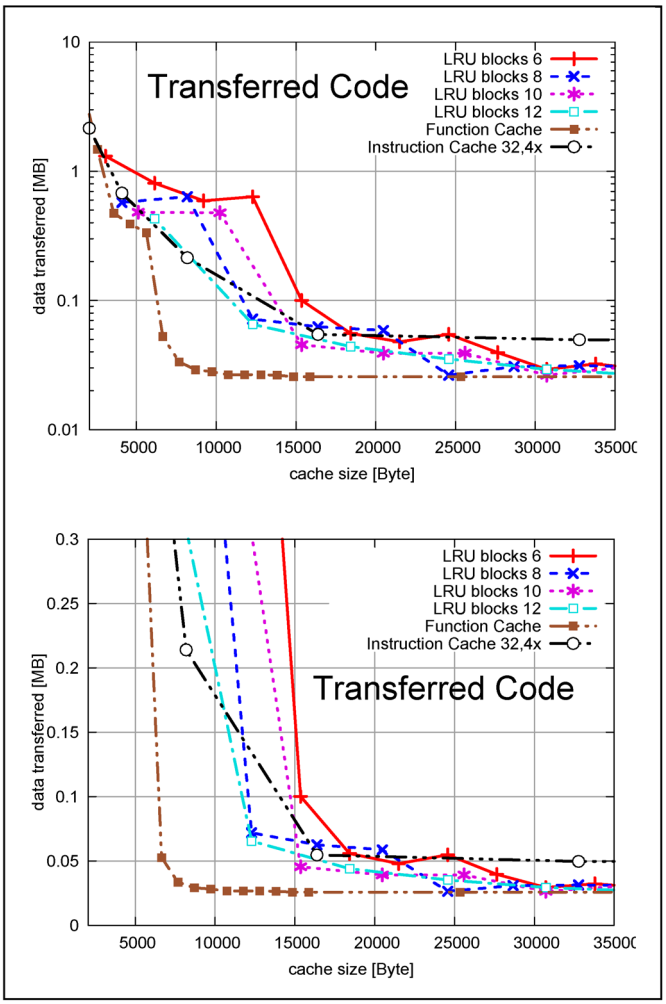

Fig. 6. Transferred data for 6-12 blocks, using LRU (upper graph uses log scale, the lower (zoomed) graph a linear scale)

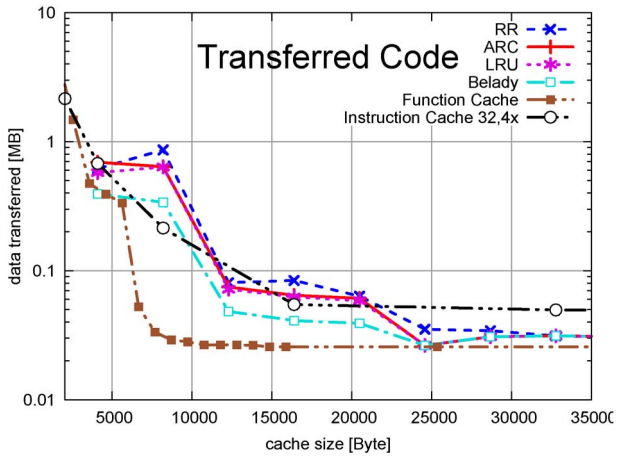

Fig. 7. Transferred data for 8 blocks, using RR, ARC, LRU and Belady

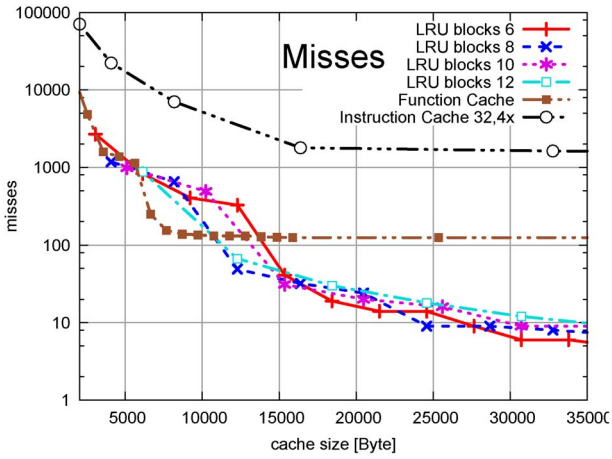

Fig. 8. Misses for 6-12 blocks, using LRU

VIII BLOCK CACHE PERFORMANCE TRANSFERRED DATA

For assessing the runtime performance of different block cache configurations we used our block cache simulator (Figure 2). The configurations evaluated span different block sizes and thus block compositions, ranging from 512 bytes up to 8192 bytes in 512 byte increments (Table III). For every block size we simulated the block caches with 6,8,10 and 12 block slots. Furthermore we evaluated the performance of different evictions strategies presented in Section V. For reference the performance of the function and instruction cache are included. We determined the amount of data transferred (external memory to on chip memory), the amount of misses and cycles in a specific hardware implementation.

Figure 6 shows the amount of code transferred from external memory to the block cache. For a cache size of about 5 kB which is 19% of the executed code size the block caches are at their smallest block size of 512 bytes. At this size they perform as good as the instruction cache. The function cache outperforms both slightly. Between 5 and 10 kB transferred code the function caches curve sharply drops below 0.05 MB closing on the 26935 bytes executed code totally (Table I). For bigger or equally sized caches the function caches only encounters fill misses and does not evict any code. The reason for the function cache coming close to the 26935 bytes is that code eviction affects only a small part of the entire cache. The average function size is only 216 bytes. Even tough the instruction cache has a smaller granularity it cannot fully exploit its size because of its limited associativity.

For the block cache the performance varies depending on the block size and number of slots available. Block size affects the function composition and may in unfortunate cases decrease performance even for a bigger cache. This can be seen for the cache with the fewest slots (6). The amount of transferred data rises between 10-15 kB.

Twice as many slots allow for more runtime adaptation and perform best in this benchmark. In between, for 8 and 10 slots the results are mixed and depend on the quality of the block composition for a specific block size. For block caches bigger than 15 kB (lower Figure 6) all block caches stay below 0.1 MB transferred code while stabilizing for cache sizes nearing the size of the total executed code at 26 kB depending on their internal fragmentation sooner or later. Interestingly the instruction cache performs quite similar to the 12 slot block cache independently of the individual block size.

Figure 7 shows the transferred code amount the block cache with 8 slots for different eviction strategies. As expected round robin (RR) / first in first out performs worst, followed by ARC and LRU. Belady the optimal algorithm is the lower bound and shows the remaining space for improvement. ARC which

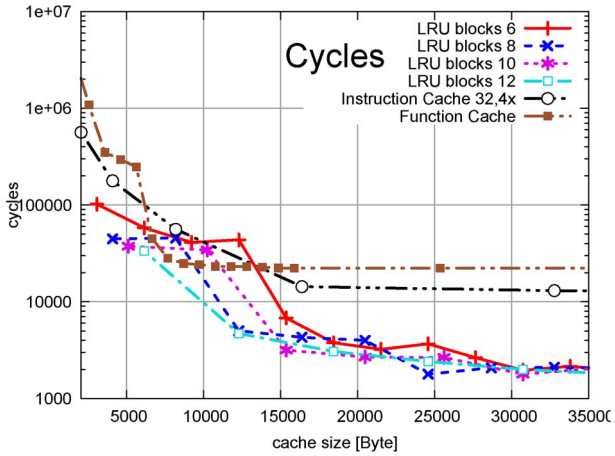

Fig. 9. Memory copy cycles for 6-12 blocks, using LRU

dynamically trades cache space between recent and frequent block accesses closely tracks LRU because it mostly favors recency over frequency.

From a hardware perspective LRU and RR are well suited for implementation. ARC is out of question considering the complexity and LRU-like performance (in our case).

A BLOCK CACHE PERFORMANCE - MISSES

Figure 8 shows the amount of misses. The instruction cache exhibits the highest number of misses in accordance to its granularity. For the function cache the curve climbs down to 125 misses, the total number of executed functions which is reached for cache sizes bigger or equal to 27766 bytes (Table I). For small block sizes the block cache is close to the function cache in performance. For 12 slots the number of misses is less than for the function cache after about 10 kB. For cache size bigger than 15 kB, there is little variation among all block cache configurations. For different cache granularities, basically different miss rates can be expected.

B BLOCK CACHE PERFORMANCE - CYCLES

The performance of a memory hierarchy can be described by its latency and bandwidth. For block caches latency matters not as much as for the instruction cache e.g. Bandwidth however can be used to its advantage. Figure 9 reflects this. It shows the number of cycles needed to transfer the code from external to on chip memory (7 times faster). We assume a double data rate dynamic memory chip with tRCD=3, CL=3 and a 64 bit interface. For every block access we need to change the row address and encounter the full latency of tRCD+ CL. The parameter choice is arbitrary but realistic and if changed will reflect the aforementioned inflictions. For the given parameters the block cache outperforms the instruction cache. The 12 slot block cache even outperforms the function cache for all block and cache sizes. Similar to the instruction cache, the function cache cannot exploit the high bandwidth as well as the block cache and suffers more from the access latencies.

IX CONCLUSION AND OUTLOOK

Block caches are a viable method to cache code in on chip memories. In comparison to instruction caches on chip memories need less area and power for the same size [3, 4, 5] and can take advantage of high bandwidth memory links as we have shown. In the past memory latencies were going up and down [13] unlike capacity and bandwidth which rise steadily with time. For future embedded systems we expect to see an increasing number of multi-core systems equipped with local memories in the megabyte range. Block caches are one opportunity to put them to a use without manual programming efforts which are time consuming and costly for the large software systems deployed nowadays. In research block caches still have a lot of potential. We are currently investigating new block compositions by identifying distinct working sets in time. Additionally, we plan to store selected functions into more than one block, trading space against performance. On the hardware site we plan to support step-wise granularities for code blocks thereby increasing the design space.

References

[1] A.Raghunathan S.Ravi S.Hattangady J.Quisquater " Securing Mobile Appliances: New Challenges for the System Designer," *In DATE '03*: Proceedings of the conference on Design, Automation and Test in Europe, pp. 176-181, 2003.

[2] P.Kocher R. Lee Gary McGraw C. Dulles A.Raghunathan S. Ravi " Security as a New Dimension in Embedded System Design," *DAC '04*: Proceedings of the 41st annual conference on Design automation, pp. 753-760, 2004.

[3] R.Banakar S. Steinke B.Lee M. Balakrishnan P. Marwedel " Scratchpad memory: design alternative for cache on-chip memory in embedded systems," *CODES '02*: Proceedings of the tenth international symposium on Hardware/software codesign, 2002.

[4] F. Angiolini F. Menichelli A. Ferrero L. Benini M. Olivieri " A post-compiler approach to scratchpad mapping of code," *In CASES '04*: Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems, pp. 259-267, 2004.

[5] S. Steinke L. Wehmeyer B. Lee P. Marwedel " Assigning Program and Data Objects to Scratchpad for Energy Reduction," *In DATE '02*: Proceedings of the conference on Design, automation and test in Europe, pp. 409, 2002.

[6] S. Steinke N. Grunwald L. Wehmeyer R. Banakar M. Balakrishnan P. Marwedel " Reducing energy consumption by dynamic copying of instructions onto on-chip memory," *In ISSS '02*: Proceedings of the 15th international symposium on System Synthesis, pp. 213-218, 2002.

[7] Nils Grunwald. Energy minimization of embedded applications by using scratchpad memory. Thesis, University Dortmund, 2002.

[8] P. Francesco P. Marchal D. Atienza L. Benini F. Catthoor J. Mendias " An integrated hardware/software approach for run-time scratchpad management," *DAC '04*: Proceedings of the 41st annual conference on Design automation, pp.238-243, 2004.

[9] B.Egger J. Lee H. Shin " Scratchpad memory management for portable systems with a memory management unit," *In EMSOFT '06*: Proceedings of the 6th ACM & IEEE International conference on Embedded software, pp. 321-330, 2006.

[10] B. Egger C. Kim C. Jang Y. Nam J. Lee S. Lyul Min " A dynamic code placement technique for scratchpad memory using postpass optimization," *In CASES '06*: Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems, pp. 223-233, 2006.

[11] C. Park J. Lim K. Kwon J. Lee S. Lyul Min " Compiler-assisted demand paging for embedded systems with flash memory," *In EMSOFT '04*: Proceedings of the 4th ACM international conference on Embedded software, pp. 114-124, 2004.

[12] N. Megiddo D. Modha " Outperforming LRU with an adaptive replacement cache," *IEEE Computer*, pp. 58-65, 2004.

[13] David A. Patterson " Latency lags bandwidth," *Commun. ACM*, 2004.