REMcode: relocating embedded code for improving system efficiency

A. Janapsatya, S. Parameswaran and J. Henkel

Abstract: The memory hierarchy subsystem has a significant impact on performance and energy consumption of an embedded system. Methods which increase the hit ratio of the cache hierarchy will typically enhance the performance and reduce the embedded system's total energy consumption. This is mainly due to reduced cache-to-memory bus transactions, fewer main memory accesses and fewer processor waiting cycles. A heuristic approach is presented to reduce the total number of cache misses by carefully relocating selected sections of the application's software code within the main memory, thus reducing conflict misses resulting from the cache hierarchy. The method requires no hardware modifications i.e. it is a software-only approach. For the first time such a method is applied to large program traces, and the miss rates and corresponding energy savings are observed while varying cache size, line size and associativity. Relocating the code consistently produces superior performance on direct-mapped cache. Since direct-mapped caches, being smaller in silicon area than caches with higher associativity (for the same size), cost less in terms of energy/access, and access faster, using direct-mapped instruction cache with code relocation for performance-oriented embedded systems is recommended. A maximum cache miss rate reduction from 71% down to less than 1% is achieved, with energy reductions of up to 63% with only a small increase in main memory size.

1 Introduction

It is well known that the memory subsystem of any computing system has a significant impact on performance and power of a whole system. As far as general-purpose microprocessors are concerned, from one processor generation to another, the on-chip cache is increasing significantly in size [1]. Often, on-chip memory will account for the largest resource of transistors, larger than the processor core. This is well-invested chip area since any access to the main memory is time consuming; depending on the memory subsystem more than 100 cycles might be necessary when the main memory is accessed after a cache miss occurs, whereas the L1/L2 cache have 1–2 cycle access time. The access cycle time is far lower for cache due to the smaller distances between processor and L1/L2 cache compared with the distance between processor and main memory, and also the smaller effective capacitance of the L1/L2 cache because of its size, silicon and memory technology.

Hardwarewise, there are many means to increase cache hit ratio, such as increasing cache size, modifying cache line sizes and cache policy (this might work when specific characteristics of an application are exploited), and introducing tiny caches for loops [2]. Other recent hardware modification techniques to improve the instruction memory and/or power performance are the inclusion of scratchpad memory [3, 4], the addition of extra levels of cache [5–7], increasing bus widths [8, 9], and shutting down sections of cache to improve energy efficiency [10]. In addition, there are compiler-based approaches for reducing the number of cache accesses by optimising the executable software. These software modification methods, called code placement or code relocation, involve careful rearrangement of instructions within the main memory. Code relocation is performed so that when these instructions are brought into cache they cause minimal conflict misses [11–17].

This paper is closely related to the code relocation methodology. Relocating the code in the main memory is a technique that is applied after the code is compiled. Relocation is unaware of the semantics of the code that is going to be relocated and thus any method using this principle is orthogonal to any compiler technique i.e. it can be applied in addition to compilation-based approaches and might further increase cache hit ratio.

In this paper we apply code relocation to a variety of applications. For the first time we analyse the effect of varying the cache size, associativity, and linesize on both the performance and energy consumption. Previous work looked at either only the performance benefit of different cache associativities and linesises on small programs [11–15, 17] or only at performance and energy gains for direct-mapped cache [16]. Throughout this paper, cache refers to instruction cache unless otherwise stated.

1.1 Contribution and characteristics of our approach

Our approach for code relocation uses a simple though effective heuristic that shows its usefulness over a broad range of cache parameters (cache size, line size, and associativity) with significant improvements for any application size/domain. This is achieved by supplying cache

IEE Proceedings online no. 20040942

doi: 10.1049/ip-cdt:20040942

Paper first received 11th December 2003 and in revised form 14th July 2004. Originally published online 2nd November 2004

A. Janapsatya and S. Parameswaran are with the School of Computer Science and Engineering, The University of New South Wales, Sydney, NSW 2052, Australia

J. Henkel is with Department of Computer Science, University of Karlsruhe, Germany

parameter information, and reducing the search space by eliminating program paths that are rarely or never executed.

Compared with general (noncode-relocation) approaches, our approach distinguishes itself in two ways:

- no additional hardware is necessary
- the approach is orthogonal to compiler-based memory optimisation strategies. It is applied after the compilation phase and yields additional (nonsemantic-sensitive) improvements whereas compiler-based approaches optimise with regard to the code semantics only. Thus our approach complements compiler-based approaches.

The basic target architecture considered in this paper is as follows: a single processor core that has exclusive access to the caches, a single unified memory, an instruction cache, a data cache, and possibly a custom hardware unit (without direct cache access). The cache can either be an L1 or L2 cache.

2 Related work

Hwu *et al.*, in [11], effectively reduced the cache miss rates in a compiler called IMPACT-1 by function inline expansion, trace selection, function layout and global layout. Function inline expansion replaces the function calls with the functions in higher execution calls. Trace selection groups the basic blocks that are often executed together, which then reduces the compulsory and conflict misses. Function layout places the most important descendant after the function entrance, thus reducing the conflict misses. Global layout places functions, which are executed together in close proximity to each other. McFarling [14, 15], analysed functions such that the dependencies amongst functions were exposed and exploited to reduce misses. Chow [12] reduced cache conflicts by sorting functions by their execution frequencies, and then grouping functions together to reduce conflict misses. All of these methods worked at the functions level.

The first of the global methods was proposed in [17] by Tomiyama and Yasuura, where an ILP formulation was applied to two differing methods to reduce the cache misses. The first method applied trace selection, which reduced the cache size but increased memory size. The second method was a refined method and applied trace selection, trace merging and trace placement to reduce the total misses. The ILP formulation reduced the speed of application and is not a suitable method for large program optimisations. Performance estimation of such caches has also been reported in the literature [18]. Kirovski *et al.* [19] use the frequency of execution to reduce cache misses in a system to be synthesised.

In [20, 21] the authors used a scheme where half the cache was assigned for high-priority tasks and the other half was allocated for nonhigh-priority tasks. This method was applied to instruction cache optimisation in multi-processor systems by Li and Wolf [22], though they later abandoned [23] for random placement of instructions in memory and doubling of cache sizes until deadlines were met.

In [16], code is reordered by examining the execution frequency of basic blocks, and placing code segments with high execution frequency next to each other within the cache. Their methodology works only for a single instruction block sizes and direct-mapped cache. Our work differs from this by having a methodology able to be applied for configurations with cache line size greater than one and cache associativity greater than one.

A different code placement method is known as branch alignment [24]. Branch alignment methods placed the most likely follower of an instruction at successive locations within the memory. Other approaches that are relevant to our work stem from the area of system-level power estimation and optimisation: Dave *et al.* [25] introduce a task-level codesign methodology that optimises for power consumption and performance. The influence of caches is not taken into consideration. The procedure for task allocation is based on estimations for an average power consumption of a processing element. The approach described by Hong *et al.* [26] uses a multiple-voltage power supply to minimise system-power consumption. Another system-level power estimation approach that focuses on peripheral cores within SOCs is described by Givargis *et al.* [27]. Their hybrid approach uses one-time-obtained gate-level power data and propagates it to an executable specification to speed up power estimation. Simunic *et al.* [28] simulated the power consumption of an ARM processor plus a cache hierarchy and a main memory using a cycle–accurate approach. Lajolo *et al.* [29] have conducted research on a cycle-based cosimulation environment for power estimation.

3 REMcode relocation strategy

The code relocation methodology is shown in Fig. 1. The inputs are cache parameters and application parameters. The REMcode strategy consists of a cache allocation algorithm and a memory allocation algorithm. The results are a list of basic blocks and their new locations in the memory. Given the new memory locations of each basic block, it is then possible to recompile the executable and adjust the branching destinations of all the relocated codes.

3.1 Assumptions and definitions

For a particular application, the following are given:

- A set of instructions $I = \{i_0 \ldots i_{q-1}\}$, where q is the number of all instructions. These individual instructions are always executed in groups called basic blocks $B = \{b_0 \ldots b_m\}\}$ and every instruction i_r (within basic block b_r) is executed e_r times.
- A program trace S of the application with its subset $L = \{l_0 \ldots l_p\}$, where each element l_i is a sequence of basic block accesses from b_k back to b_k itself. Each element l_i is called a loop. A basic block can be part of multiple loops.
- The instruction width is defined as I_w.
- Cache parameters are given by cache size $= N$, cache line size $= N_L$, cache associativity $= N_A$. Thus the cache location (set) of each memory location can be determined by

$$N_{loc} = M_{loc} \text{mod}(N/N_A) \qquad (1)$$

where M_{loc} is the memory location.

The following assumptions are also made:

- For RISC machines each instruction is of the same width, i.e. I_W is constant. This assumption of fixed instruction size is made to simplify the reallocation of code in memory. For machines with varied instruction widths it is still possible to implement our algorithm with careful allocation of instructions into the memory to ensure that instructions are aligned with each other in memory.
- Cache line size $N_L \geq$ instruction width I_W.
- The size of the cache is known *a priori*. This assumption is made to allow for greater optimisation. However, if a number of processors with differing sizes of cache have to be serviced by the same binary source it is possible to ship

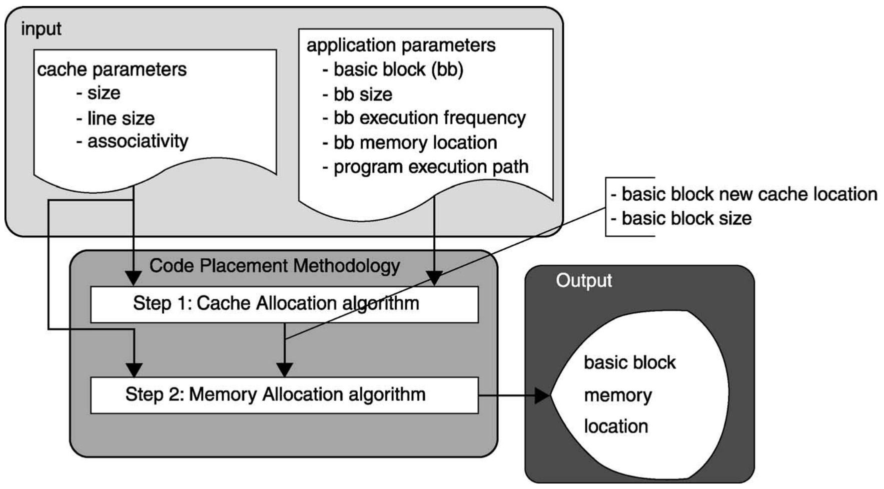

Fig. 1 *Flow of REMcode (code relocation strategy)*

several binaries, and the suitable one can be applied to the particular embedded system.

- Each basic block is smaller than the cache size divided by the cache associativity. This assumption is typically fulfilled. However, if the basic block is too large it can always be broken into smaller basic blocks.
- The methodology targets the low-power embedded system, where the cache size is tailored to suit the application and silicon area restricts the use of large cache sizes.

3.2 Problem statement

Let the size of the cache associated with processor P be equal to N. The basic blocks to be placed in the cache be $b_1, b_2 \ldots\ldots b_m$ all of which belong to the set B. Each block b_i is associated with size c_i. Let a loop l_r be defined as being a subset of B and the loops be numbered from $l_1, l_2, \ldots\ldots l_p$. Note that while multiple loops can share a basic block, the set of basic blocks within a loop will be unique to that loop. Some basic blocks will not belong to any of the loops. The problem is to arrange the basic blocks of each of the loops (and other basic blocks which do not belong to any of the loops) in main memory so that it can be brought into the cache such that the conflict misses are reduced, and the total main memory used is minimised. This problem is known to be NP-complete [16].

3.3 REMcode cache allocation

The cache allocation algorithm is used to find locations to map each basic block to locations in cache to minimise total conflict misses. If a number of basic blocks belonging to the same loop overlap in cache (located at same address space), cache misses will occur repeatedly.

3.3.1 Cache allocation process:

We illustrate the cache allocation process using a sample program for allocation into a 64 bytes cache, Fig. 2*a*. Each vertex represents one basic block. The directed edges indicate the traversal of the program and the number beside each edge indicates the number of times each path is executed. In addition to the program flow graph shown, cache parameters and application parameters are given, Table 1. Column 1 gives the basic block name, column 2 the execution frequency of the basic blocks, and column 3 the effective execution frequency (calculated in step 1 of cache allocation algorithm). Column 4 shows the size of each basic block, and column 5 shows the original location of each basic block. Column 6 shows where each basic block maps into

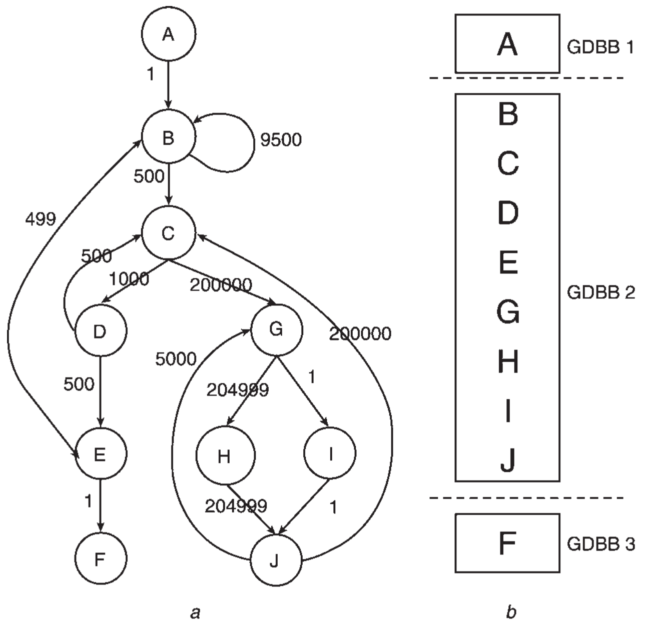

Fig. 2 *Sample program flow graph*

cache using (1). For this example, we assume a direct-mapped cache.

Inputs to the cache allocation algorithm are the program flow graph (such as the one given in Fig. 2*a*), cache parameters, and the application parameters shown in Table 1 columns 1, 2, 4, 5, and 6. The cache allocation algorithm is displayed in Fig. 3.

Step 1: Basic block B shown in Fig. 2*a* has a path to itself that is executed 9,500 times. This loop back path of a basic block B will never cause a conflict (given that our cache is always larger than any single basic block). The first step adjusts for this loop back path. Basic block B's effective execution frequency is reduced to 500 and shown in column 3 of Table 1. The effective execution frequency is the number of times a cache conflict can occur.

Step 2 identifies the loops within the application. In this work the loops are identified by searching for backward paths in the program trace. In the example given, the following loops have been identified: BCDEB, CDC, CGHJC, CGIJC, GHJG, GIJG, BCGHJCDEB, BCGIJCDEB, BCGHJGIJCDEB, BCGHJGIJCDEB, BCDCDE.

Step 3 separates the program into groups of dependent basic blocks (*GDBB*), created by partitioning the program into smaller groups of basic blocks. If one basic block

Table 1: Cache allocation example (for a 64 byte cache)

bb	Exe. freq.	Effective exe. freq.	Basic block size	Mem. map	I-cache map	Allowed I-cache map	New I-cache map	New mem. map
A	1	1	22	0–21	0–21	0–21	0–21	0–21
B	10,000	500	10	22–31	22–31	0–9	23–32	86–95
C	201,000	201,000	64	32–95	32–31	0–63	22–21	22–85
D	1000	1000	7	96–102	32–38	10–16	33–39	96–102
E	500	500	18	103–120	39–56	17–34	40–57	103–120
F	1	1	9	121–129	57–1	0–7	22–30	150–157
G	205,000	205,000	8	130–137	2–9	47–54	5–12	197–204
H	204,999	204,999	17	138–154	10–26	47–63	5–21	133–149
I	1	1	9	155–163	27–35	35–42	58–0	121–128
J	205,000	205,000	7	164–170	36–42	55–61	13–19	205–211

```
STEP 1
For each basic block, $b_i$ in set B
  Find and remove basic blocks that have a loop back path
STEP 2
Identify Loops in program
STEP 3
Create Group of Dependent Basic Block (GDBB)
STEP 4
For each GDBB {
  Identify path that exceeds a threshold $T_L$ into a separate
    group, called dominant loops DL within GDBB
  Remove basic blocks identified as DL from the GDBB
  Add identified DLs as new GDBB
}
For each GDBB {
  STEP 5
  Rank basic blocks in descending order of execution
    frequency
  STEP 6
  While Cache is not full AND list of basic block in the
    GDBB is not empty
    Allocate basic block from lowest address to highest
      address in cache.
  If there exists unallocated basic block {
    Allocate the largest in size to the bottom of cache and
      marked the start of this basic block as A(ls).
    While unallocated basic block list is not empty
      · Allocate basic blocks within A(ls) and the bottom of
        cache.
  }
}
```

Fig. 3 *Cache allocation algorithm*

belongs to two or more separate loops we partition all basic blocks in the different loop into a single *GDBB*. For example, in Fig. 2*a*, basic block A, will never cause a conflict miss with basic blocks 'B, C, D, E, G, H, I, J' or with basic block F. Thus we group A into one *GDBB*; 'B, C, D, E, G, H, I, J' into another and finally F into a third *GDBB* (Fig. 2*b*). Basic blocks in separate *GDBB*s will not cause a conflict miss. Therefore we can consider each *GDBB* separately, greatly simplifying the cache allocation problem.

Step 4 is used to identify dominant loops *DL* within a *GDBB*. A *DL* is any instruction group with a maximum effective executed frequency greater than the threshold percentage T_L. Basic blocks identified as part of a *DL* is

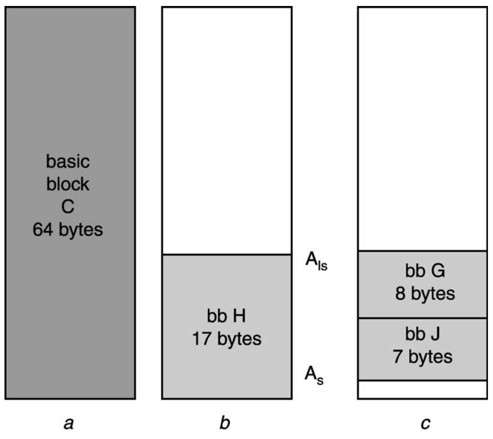

Fig. 4 *Illustration of cache allocation mapping*

removed from the *GDBB*. We set T_L to be 5%. In this example only the loop CGHJ is above this threshold. Each *DL* is then added as new *GDBB* to the list of *GDBB*s, and each *DL* will then be considered as another *GDBB*. By treating the *DL*s as *GDBB* we further simplify the cache allocation problem. Even though separate consideration of the new *GDBB*s (previously *DL*) does not guarantee zero conflict miss, the effect on the total miss rates would be negligible due to small threshold percentage T_L, chosen.

Step 5 ranks basic blocks within each *GDBB* in descending order of execution frequency. If two basic blocks have execution frequencies within the same order of magnitude, the larger one is ranked higher than the smaller one (this reduces capacity misses). In the example, basic blocks CGHJ within the *GDBB* identified in the previous step will be ordered as CHGJ.

Step 6 allocates individual instructions into a location in cache. Consider each *GDBB* separately. The basic blocks within each *GDBB* are taken from the ordered list given in step 5 (in the given order). These basic blocks (only whole basic blocks are allowed) are allocated to the cache from the lowest address to the highest until the cache is completely filled or there are no remaining basic blocks within that *GDBB*, Fig. 4*a*.

If there are any remaining basic blocks, we find the largest basic block from the remaining basic blocks in the ordered list. This large basic block is allocated to the starting address A_{ls} and ending at address A_s of the cache, Fig. 4*b*. After this we take the next largest basic block and allocate its starting address in the cache to A_{ls}. The ending address will be less than the final address of the cache. Thus, if another unallocated basic block can be found which fills the space (below the basic block we just allocated and above the last cache address A_s), we allocate that basic block to the available space, Fig. 4*c*. We keep doing this until no more basic block can fit to the remaining available space. We take the next largest unallocated basic block and start it at address A_{ls} and repeat the process until all basic blocks are allocated. We do this step for all *GDBB*s. The complexity of this algorithm is $O(S + B^2)$, where B is the number of basic blocks and S is the size of the program trace. For a cache with associative greater than one, the algorithm is modified to allow the allocation of code in every one of the ways in the associative cache. This allows basic blocks to overlap in multiple identical cache locations on different sets.

3.3.2 Worked example:

Consider only the list C, H, G, and J for allocation into a 64-byte direct mapped cache. Basic block C is allocated first. Since block C is 64 bytes long, it occupies the entire cache as seen in Fig. 4*a*. Allocation of basic block H will force part of basic block C to be replace from the cache. The allocations of the other two basic blocks G and J will force further replacement of part of basic block C and/or H from the cache. To minimise the total cache misses it is best to allow basic block H, G, and J to overlap each other; hence causing only minimal portion of basic block C to be replace out from the cache. Basic block H is to be allocated to replace a portion of basic block C as shown in Fig. 4*b*. The start of basic block H is then marked as A_{ls}, which indicates the location for overlapping the remainder of the unallocated basic blocks; G and J. The next step is to allocate basic block G into location A_{ls} and allocate basic block J into location '$A_{ls} + sizeofbasicblockG$' shown in Fig. 4*c*.

Figure 4 shows the resulting allocation of basic blocks CHGJ into the cache. Output of the cache allocation algorithm gives the location at which each basic block should be within the cache; this is given in column 7 of Table 1.

3.4 Memory allocation

The memory allocation algorithm is used to map basic blocks into memory. Inputs to the algorithm are the application parameters shown in columns 1, 3 and 4 of Table 1, and output from cache allocation algorithm shown in column 7 of Table 1). The memory allocation algorithm given in Fig. 5.

```
Allocate first basic block to start of memory
Set $Z_r$ to be the size of the first basic block
Order each GDBB in descending order of the total size

For each basic block $b_i$ in GDBB {
  Repeat until each basic block is allocated to memory {
    Allocate first basic block in the first available
      contiguous memory block ($M_x$ to $M_Y$), which will
      hold the basic blocks

    Calculate $Z_r = (M_x \bmod N/N_A) - t_x$, where $N_A$ is the
      cache associativity
    While basic block not allocated do {
      If memory locations $t_x + Z_r + i \times N$ to $e_x + Z_r +$
        $i \times N$ is free then

        ·Map basic block to address $t_x + Z_r + i \times N$ to
          $e_x + Z_r + i \times N$
      Else
        ·$i{+}{+}$
    }
    Remove basic block from unallocated list
  }
}
```

Fig. 5 *Memory allocation algorithm*

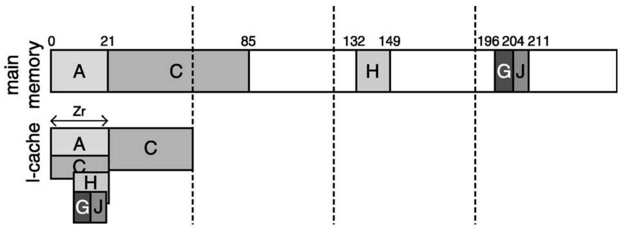

Fig. 6 *Memory allocation example*

Figure 6 shows an example of how basic blocks are mapped directly into memory. In this Figure only allocation of basic blocks A, C, G, H, and J from the example program Fig. 2*a* are shown. Basic block A is allocated to the start of memory because it indicates the start of the program. Basic block C, G, H, and J are allocated such that when they are loaded into cache, they overlap as shown in Figure. Looking at the allocation of basic blocks C, G, H, and J, note that they will not map directly to the cache location as predicted in column 7 of Table 1 but will map to cache as indicated in column 8 of Table 1. This is acceptable because the overlap in cache between blocks within the same *GDBB* is as predicted in the cache allocation algorithm. Thus if a basic block is mapped to the location from t_x to e_x in cache, the basic block can be placed in memory in any one of the address ranges from addresses $t_x + Z_r + i \times N$ to $e_x + Z_r + i \times N$, where i is a positive integer, Z_r is the cache offset, and N is the total cache size divided by the cache associativity. This introduction of an offset value Z_r allows a reduction in size of the total memory needed for the system.

The memory algorithm starts by allocating the first basic block into the start of the memory (i.e. the start of program). In the example it is basic block A. It cannot be moved from the initial location, since, changing the location of A will cause the calling program directed to the wrong location. The next step is to sort the *GDBB*s in descending order of size (i.e. the total size of all the basic blocks within each). Then, for each *GDBB* in the ordered list, allocate the first basic block in each group to the first available contiguous location in memory that is large enough to accommodate the basic block. The offset value Z_r is then calculated for the current *GDBB* based on the equation shown in the algorithm in Fig. 5. Subsequent basic blocks in the same *GDBB*, are allocated to the cache by finding free space in memory between $t_x + Z_r + i \times N$ to $e_x + Z_r + i \times N$. Results of the memory allocation algorithm for the sample program are shown in column 9 of Table 1. The runtime complexity of the memory allocation algorithm is $O(BM)$, where B is the number of basic blocks and M is the size of the memory.

4 Experimental procedure and discussions

The experimental setup is shown in Fig. 7. Application programs written in C were run on a SPARC machine (Sun UltraSparc with four processors, 400 MHz, 2GB, Solaris 2.6). Instruction traces of the applications are obtained using QPT[30]. A custom profiler was used to gather application parameters (shown in Fig. 7).

Step 2 reads the cache parameters. Cache allocation and memory allocation algorithms are applied to design the new instruction code layout and the corresponding new program trace. These two execution traces are applied as two separate inputs in step 3. The cache simulator, dineroIV [31], is deployed to simulate and calculate the cache miss rates whereas the memory calculator is a custom program calculating the memory size increase. An analytical power model estimates the energy consumption. The output gives the cache miss rate comparison of both the original and the modified program trace, the increase in memory size due to the code relocation algorithm, and the percentage energy savings.

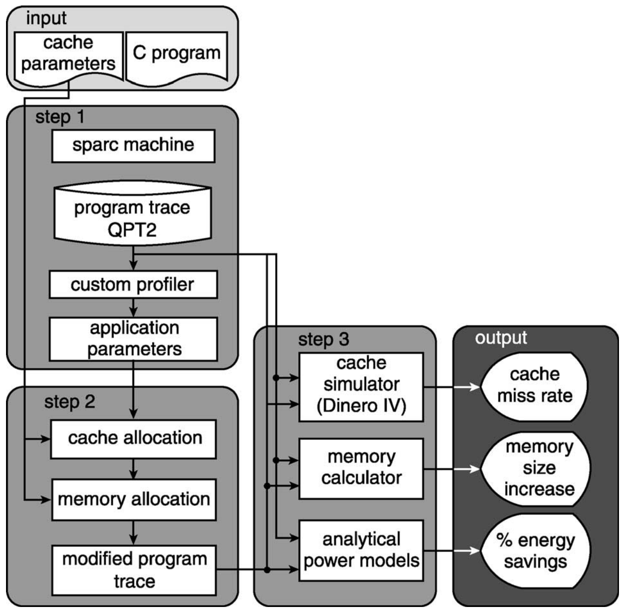

Fig. 7 *Experimental setup*

Six different benchmarks were used to validate our approach. The trace size is shown in column 1 on Table 2. Program size in bytes is shown in column 2 of Table 2. The experiment was conducted for varying associativity (1, 2, and 4), varying instruction cache size from 64 to 2048 bytes, and varying cache line sizes from 4 to 8 bytes. The larger size of instruction cache was not evaluated owing to most results already showing a cache miss rates of less than 1% for cache size of 2048 bytes. Data cache size was fixed at 1024 bytes and the least recently used (LRU) replacement policy was used in the experiment. The runtime of the cache allocation algorithm and the memory allocation algorithm are only a few seconds in all the experiments.

Table 2 shows the results of the six benchmarks for a direct mapped cache with cache line size of 4 bytes. Column 3 shows the instruction cache size in bytes. Columns 4 and 5 show the number of instruction misses and miss rate of the original code layout. The relocated code misses are given in column 6 and its miss rate is in column 7. Columns 8, 9, and 10 show the energy consumption of the original code, the modified code, and the percentage energy savings, respectively. Column 11 shows the percentage main memory size increase.

Table 2: Table of results

Application & trace size	Program size	I-cache size	Original miss	Miss rate (%)	Modified miss	Miss rate (%)	Original energy(J)	Modified energy(J)	Energy saving (%)	Size increase (%)
compress	4215320 bytes	128	39686614	74.49	31235336	58.62	7.494007	6.327155	15.57	0.18
(53280973)		256	29161991	54.73	10481347	19.67	6.0605139	3.566770	41.15	0.36
		512	7102452	13.33	1813645	3.40	3.0002462	2.413922	19.54	0.67
		1024	1272890	2.39	820511	1.54	2.2959965	2.281830	0.62	0.50
		2048	443168	0.83	228738	0.43	2.4338514	2.203121	9.48	0.39
mpeg	129603 bytes	128	16343210	72.94	15901083	70.97	23.966566	23.345371	2.59	42.13
(22406459)		256	12592796	56.20	12182514	54.37	18.91491	18.354527	2.96	101.16
		512	10721491	47.85	8359079	37.31	16.424782	13.222937	19.49	194.82
		1024	7378424	32.93	4858766	21.68	11.969864	8.525023	28.78	372.54
Q8	8888 bytes	64	30307448	67.63	23369636	52.15	4.7998051	3.833297	20.14	0.54
(44814000)		128	27500972	61.37	11299336	25.21	4.4282823	2.165848	51.09	0.05
		256	4538800	10.13	3426923	7.65	1.21218817	1.078315	11.04	0.05
		512	241	0.0	241	0.0	0.62180677	0.604937	2.71	0
trick1	9312 bytes	128	103104784	100	103104784	100	19.514151	19.709437	−1.00	12.7
(103104786)		256	102669904	99.58	96873072	93.96	19.582832	18.869561	3.64	26.7
		512	90421320	87.70	61623088	59.77	18.081579	14.118756	21.92	22.9
		1024	73378136	71.17	39632	0.04	16.152711	5.818868	63.98	12.5
		2048	18248122	17.70	486	0.00	8.673339	5.813592	32.97	0
tsp	23076 bytes	64	2657486	78.08	2642578	77.64	0.51087595	0.4996175	2.20	6.1
(3403735)		128	1931083	56.73	1815230	53.33	0.41021391	0.3945525	3.82	13.3
		256	1240095	36.43	1208763	35.51	0.31553183	0.3175372	−0.64	27.5
		512	856395	25.16	755447	22.19	0.26643111	0.2599706	2.42	39.8
		1024	488049	14.34	787801	23.15	0.22323306	0.2640792	−18.30	90.5
		2048	288144	8.47	365898	10.75	0.212027719	0.2105018	0.72	43.1
whetston	11408 bytes	64	1749402	100	1714912	98.03	0.30186870	0.297138	1.57	3.4
(1749402)		128	1701363	97.25	1500996	85.80	0.29518329	0.267796	9.28	3.8
		256	1215194	69.46	775510	44.33	0.22842737	0.1682862	26.33	0.8
		512	108059	6.18	106530	6.09	0.073333428	0.0765265	−4.35	0.1
		1024	993	0.06	993	0.06	0.062205727	0.062050705	0.25	0

Table 3 shows the cache miss rates in column 2 to 5 (to read Table 3, the column titled 'cache size' is referred to as column 1) for *compress, mpeg, and trick1* applications for differing cache line sizes (other results are not presented due to lack of space). Results shown on Table 3 are divided into three the applications (*compress*, *mpeg*, and *trick1*).

4.1 Effect on system performance

Reduction in cache miss rates will improve the system's performance. Cache miss rates are shown in Table 2. The best cache miss rate results for each application is shown in Fig. 8*b* (before and after code relocation). The results in Table 2 show that using a smaller cache size with code relocation can provide similar or better cache miss rates compared with using larger cache sizes with the original code placement as shown in [16, 17] The results show that cache miss rates can be reduced from 71.17% to 0.04% for *trick1*.

Cache miss rates for varying cache line sizes are shown in Table 3 for three applications. Comparing columns '2 and 3', and '4 and 5' of Table 3, we see that code relocation does reduce miss rates in most cases. However, as we go to larger line sizes with greater associativities, cache miss rates do not always reduce with code relocation (for example in Table 3, *trick1* with associativity of two and line size of 4 and 8 for 512 byte cache). Since the cache replacement policy dictates which of the sets are replaced, and we have no control over the replacement, the effectiveness of code relocation is diminished.

Larger cache line sizes can decrease the cache miss rate and cache access time but will increase the cache penalty since more bytes will have to be brought in to cache for every miss. Cache penalty is the cost (both time and energy) of bringing elements from the memory into the cache. In calculating the CPU time (using the equation given in [17]), it is assumed that the memory bus width is fixed at 32 bits (4 bytes). Thus a cache line size of 8 bytes requires two memory accesses for each cache miss (other memory types such as RDRAM can access memory faster for subsequent data bytes, but this is beyond the scope of this paper). Column 6 of Table 3 gives the % speedup of applications with line size of 4 bytes. Column 7 of Table 3 shows the % speedup of code relocation for cache line size of 4 bytes against cache line size of 8 bytes. The positive value for all the values in column 7 of Table 3 indicates that

Table 3: Results for different cache configuration

			Instruction cache miss rate (%)				speedup (%)	
			line size = 4		line size = 8		line size comp.	
		Cache size	old	new	old	new	4	4vs8
compress	1	128	74.49	58.62	39.34	34.87	20.86	15.6
cache assoc.		256	54.73	19.67	29.41	11.96	62.31	16.7
		512	13.33	3.40	7.44	1.78	66.76	3.2
		1024	2.39	1.54	1.28	0.79	21.62	3.3
	2	128	64.28	60.40	33.72	34.42	5.9	12.0
		256	26.58	28.18	16.29	17.98	−5.7	20.7
		512	12.43	3.49	6.31	1.86	64.0	4.4
		1024	1.33	1.56	0.73	0.83	−7.9	3.2
	4	128	70.04	63.50	36.88	43.10	9.1	25.9
		256	32.80	42.58	18.42	23.75	−28.5	10.0
		512	4.01	3.65	2.09	2.33	6.6	16.4
		1024	0.93	1.72	0.55	0.93	−31.8	4.5
mpeg	1	256	56.20	54.73	28.67	29.73	3.17	8.3
cache assoc.		512	47.85	37.31	24.16	20.66	21.36	9.4
		1024	32.93	21.68	16.87	12.21	32.62	10.6
	2	256	57.31	57.44	29.29	31.87	−0.2	9.7
		512	48.01	44.03	24.24	24.37	8.0	9.4
		1024	46.37	37.64	23.64	19.70	18.2	4.3
	4	256	58.26	58.46	29.83	31.69	−0.3	7.6
		512	47.99	48.09	24.24	26.60	−0.2	9.4
		1024	47.21	47.43	23.83	25.35	−0.5	6.3
trickl	1	256	99.58	93.96	52.60	52.17	5.56	9.8
cache assoc.		512	87.70	59.77	46.86	35.28	31.30	15.0
		1024	71.17	0.04	39.20	0.02	97.83	0.4
	2	256	100	99.99	52.21	52.61	0.0	4.9
		512	98.76	99.96	50.99	52.20	−1.2	4.2
		1024	28.60	0.05	16.52	0.03	94.7	0.5
	4	256	100	100	52.21	52.99	0	5.6
		512	99.96	99.96	52.19	51.83	0	3.5
		1024	27.22	0.06	17.64	0.04	94.4	0.8

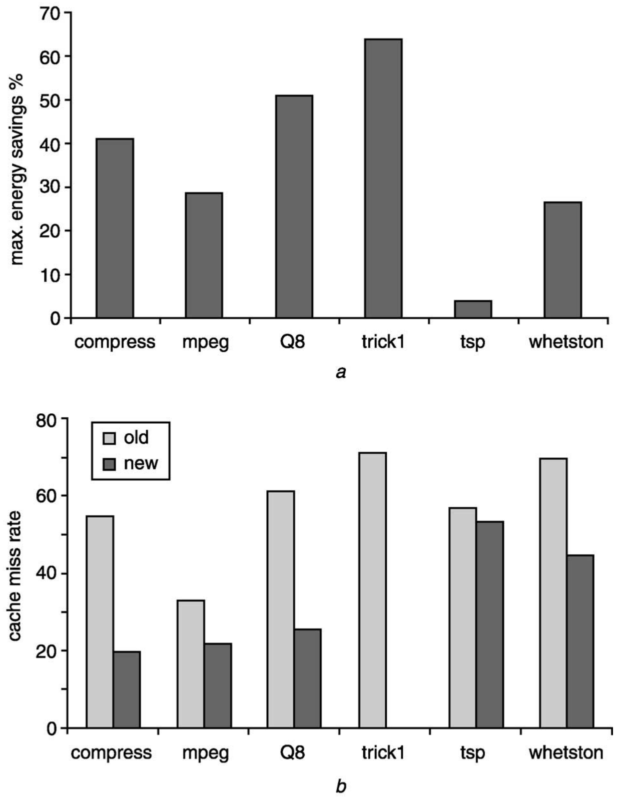

Fig. 8 *Final results showing best energy savings and best cache miss rates for each application*

with line size of 4 bytes, performance is always better compared to a larger line size cache.

4.2 Impact on energy consumption

Energy estimation (see columns 8–10 of Table 2) shows a reduction in energy consumption due to reduced conflict cache misses. Energy measurements are performed using analytical power models [32] to estimate the energy consumption of I-cache, D-cache, buses, and main memory. CPU energy is estimated using an instruction set simulator (ISS). Results of energy estimation for direct mapped caches are shown in columns 8–10 of Table 2. The best energy savings gain for each application is shown in Fig. 8*a*. Results shown in Table 2 show that energy reduction is obtained whenever lower instruction cache miss rates are observed.

4.3 Effect on area of memory hierarchy

Higher cache associativity (columns 2–5 of Table 3: each column contains figures for associativities 1, 2 and 4) can decrease cache miss rates but increases the complexity of the cache architecture, increasing cache access times, cache area (silicon) and energy/access [33].

The increase in main memory due to the code relocation algorithm is shown in column 11 of Table 2. The average memory size increase is 13% if we exclude the *mpeg* application. The *mpeg* application was written for a general purpose computer system, though our system assumes a single application embedded system. Our trace generator does not reflect the system instructions that were executed and therefore had a large number of short segments of code which were overlapping. This was due to one system call with changing pointers as arguments. Since the size of the loop is larger than the cache size, the overlapping of instructions in cache causes the memory area to be large. Memory is usually available in sizes which are multiples of 2^n (1 K, 2 K, 4 K etc), and therefore it is possible that for most applications there is no need for extra memory, since usually there is sufficient unused main memory available to implement this increase of 13%.

4.4 Summary of results

Experimental results show that code relocation can increase performance of the system and decrease its energy consumption. For cases with very small cache sizes, it is seen that in most cases code relocation has very little or no effect on the cache miss rates. This is because most of the cache misses are due to capacity misses. At the other end of the problem, where cache sizes are very large, code relocation also shows little or no effect. This is due to most loop sizes being smaller than the cache and the whole loops are able to fit within the cache concurrently.

Results of the *trick1* application shows that the cache miss rate is reduced from 71.17% down to 0.04% with 1024 bytes of cache. Analysing the statistical information for the *trick1* application reveals that 99.98% of the time, it is executing a single loop (the size of this loop is below 1024 bytes). The code relocation method increases the spatial locality of the code resulting in reduced cache miss rates.

In the case of the *tsp* application, for any cache size the code relocation method shows little reduction in cache miss rates (even an increase in cache miss rates is seen with cache size of 1024 and 2048 bytes). The statistical information for *tsp* application shows that a single big loop of size larger than 7000 bytes is executed 99% of the time. Hence any cache size up to 2048 bytes will not contain the whole loop in the cache.

One of the most interesting observations is that the direct mapped cache with code relocation will always perform better than caches with greater associativity (with or without code relocation). The use of a higher associativity cache exploits temporal locality in programs, while code relocation increases the spatial locality of the code. Thus the two methods through different means result in improved performance. Since direct-mapped caches have smaller cache areas (silicon) and faster access times, any embedded code requiring high performance should use direct-mapped cache with code relocation. The unwanted side-effect of code relocation is the increased main memory size.

5 Conclusions

A heuristic code placement algorithm for minimizing conflict cache misses has been presented. For the first time it has investigated the effect of code placement algorithms on performance and energy consumption for varying cache sizes, cache associativities and cache line sizes. We have also identified that code placement methods increase the spatial locality of codes. Results from experiments have shown that cache conflict miss rate reduction can be up to 71% and energy consumption savings of 63%.

It is seen that direct mapped cache with code relocation always performs (lowest miss rates) better than caches with higher associativity (with or without code relocation). Since direct-mapped caches are faster, smaller in silicon area, and consume less energy/access than caches with higher associativities, we recommend the use of direct-mapped caches with code relocation for performance-oriented embedded systems.

6 References

1 Intel. Intel pentium 4 processor: Tech specs and features. http://www.intel.com.apac/eng/home/desktop/pentium4/tech_info.htm, 2003

2 Cotterell, S., and Vahid, F.: 'Synthesis of customized loop caches for core-based embedded systems'. Presented at Int. Conf. on CAD, 2002

3 Banakar, R., Steinke, S., Lee, B.S., Balakrishnan, M., and Marwedel, P.: 'Scratchpad memory: a design alternative for cache on-chip memory in embedded systems'. Proc. CODES, 2002, pp. 73–78

4 Panda, P.R., Dutt, N.D., and Nicolau, A.: 'Efficient utilization of scratch-pad memory in embedded processor applications'. Proc. Conf. on Design and Test in Europe (DATE), 1997, pp. 7–11

5 Bellas, N., Hajj, I., Polychronopoulos, C., and Stamoulis, G.: 'Architectural and compiler support for energy reduction in the memory hierarchy of high performance microprocessors'. Proc. ISLPED, 1998, pp. 70–75

6 Tang, W., Gupta, R., and Nicolau, A.: 'Power savings in embedded processors through decode filter cache'. Presented at Conf. on Design and Test in Europe (DATE), 2002

7 Wang, W., Veidenbaum, A.V., and Nicolau, A.: 'Reducing power with an 10 instruction cache using history-based prediction'. Presented at IWIA, 2002

8 Zhang, C., and Vahid, F.: 'A power-configurable bus for embedded systems'. Presented at IEEE Int. Symp. on Circuits and Systems, 2002, vol. 5

9 Zhang, C., Vahid, F., and Najjar, W.: 'Energy benefits of a configurable line size cache for embedded systems'. Proc. IEEE Computer Society Annual Symp. on VLSI, 2003, pp. 87–91

10 Powell, M.D., Yang, S.H., Falsafi, B., Roy, K., and Vijaykumar, T.N.: 'Reducing leakage in a high-performance deep-submicron instruction cache', *IEEE Trans. Very Large Scale Integr. Syst.*, 2001, pp. 77–89

11 Chang, P.P., Mahlke, S.A., Chen, W.Y., Warter, N.J., and Hwu, W.W.: 'IMPACT: an architectural framework for multiple-instruction-issue processors', *Comput. Arch. News*, 1991, vol. 19

12 Chow, F.: 'A portable machine-independent global optimizer - design and measurement'. PhD thesis, Computer Systems Lab, Stanford Univ., 1983

13 Hwu, W.W., and Chang, P.P.: 'Achieving high instruction cache performance with an optimizing compiler'. Proc. 16th Annual Int. Symp. on Computer Architecture, 1989, pp. 242–251

14 McFarling, S.: 'Program optimization for instruction caches'. Proc. 3rd Int. Conf. Architectural Support for Programming Languages and Operating Systems, 1989, vol. 17, pp. 183–191

15 McFarling, S.: 'Procedure merging with instruction caches', *SIGPLAN Not.*, 1991, **26**, p. 71

16 Parameswaran, S., and Henkel, J.: 'I-CoPES: fast instruction code placement for embedded systems to improve performance and energy efficiency'. Proc. Conf. on ICCAD, 2001, pp. 635–641

17 Tomiyama, H., and Yasuura, H.: 'Code placement techniques for cache miss rate reduction', *ACM Trans. Des. Autom. Electron. Syst.*, 1997, **2**, pp. 410–429

18 Li, Y.T., Malik, S., and Wolfe, A.: 'Performance estimation of embedded software with instruction cache modeling', *ACM Trans. Des. Autom. Electron. Syst.*, 1999, **4**, pp. 257–279

19 Kirovski, D., Lee, C., Potkonjak, M., and Mangione-Smith, W.H.: 'Application-driven synthesis of memory-intensive systems-on-chip', *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, 1999, **18**, pp. 1316–1326

20 Kirk, D.B.: 'SMART (strategic memory allocation for real-time) cache design'. Proc. Symp. on Real-Time Systems, 1989, pp. 229–237

21 Kirk, D.B., and Strosnider, J.K.: 'SMART (strategic memory allocation for real-time) cache design using the mips r3000'. Proc. Symp. on 11th Real-Time Systems, 1990, pp. 322–330

22 Yanbing, L., and Wolf, W.: 'Hardware/software co-synthesis with memory hierarchies'. Presented at Conf. on Design Automation, 1998

23 Yanbing, L., and Wolf, W.: 'Hardware/software co-synthesis with memory hierarchies', *IEEE Trans. Computer Aided Design Integr. Circuits Syst.*, 1999, **18**, pp. 1405–1417

24 Young, C., Johnson, D.S., Karger, D.R., and Smith, M.D.: 'Near-optimal intraprocedural branch alignment'. Proc. Conf. on PLDI, 1997, pp. 183–173

25 Dave, B.P., Lakshminarayana, G., and Jha, N.K.: 'COSYN: hardware-software co-synthesis of embedded systems'. Proc. Conf. on Design Automation, 1997, pp. 703–708

26 Hong, I., and Kirovski, D., *et al.*: 'Power optimisation of variable voltage core-based systems'. Proc. Conf. on Design Automation, 1998, pp. 176–181

27 Givargis, T., Vahid, F., and Henkel, J.: 'A hybrid approach for core-based systems-level power modeling'. Proc. ASP-DAC, 1999, pp. 141–145

28 Simunic, T., Benini, L., and De Micheli, G.: 'Cycle-accurate simulation of energy consumption in embedded systems'. Proc. Conf. on Design Automation, 1999, pp. 867–872

29 Lajolo, M., Raghunathan, A., Dey, S., and Lavagno, L.: 'Efficient power co-estimation techniques for system-on-chip design'. Proc. Conf. on Design and Test in Europe (DATE), 2000, pp. 27–34

30 Hill, M.D., Larus, J.R., and Lebeck, A.R., *et al.*: 'Warts:wisconsin architectural research tool set', Computer Science Department, University of Wisconsin, http://www.cs.wisc.edu/~markhill/DineroIV/

31 Edler, J., and Hill, M.D.: Dinero iv trace-driven uniprocessor cache simulator. Computer Science Department, University of Wisconsin

32 Li, Y., and Henkel, J.: 'A framework for estimating and minimizing energy dissipation of embedded hw/sw systems'. Proc. Design Automation Conf., 1998, pp. 188–193

33 Shivakumar, P., and Jouppi, N.P.: Cacti 3.0: An integrated cache timing, power, and area model, Compaq WRL Report, August 2001