Frequency-Based Code Placement for Embedded Multiprocessors

Corey Goldfeder
Columbia University
500 West 120th Street, M.C. 0401
New York, New York, 10027
coreyg@cs.columbia.edu

ABSTRACT
Multiprocessor embedded systems often have processor-local caches and a shared memory. If the system's code is available at design time we can maximize cache hits by rearranging code in memory so that frequently executed tasks reside in reserved areas of the caches and are not overwritten by less frequent tasks.

Categories and Subject Descriptors
D.4.2 [**Operating Systems**]: Storage Management - Main memory; C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems

General Terms
Algorithms, Performance, Design.

Keywords
Embedded Systems, Multiprocessors, Caching, Memory, Code Placement, Frequent Code

1. INTRODUCTION
A common setup for an embedded system is a system composed of several commodity processors, each with local cache, sharing a single main memory. For various reasons, mostly economic, the processors in embedded systems are often direct-mapped. The inclusion of more than one processor in an embedded system, which is likely to be already restricted in terms of power and memory, creates a host of new problems such as resource sharing and memory synchronization. This paper focuses on how to optimally place code in the shared memory. Although an equivalent problem can be formulated for a single processor both the problem and its solutions are expressed more naturally in regards to a multiprocessor system.

The usual relationship between memory and cache is that code is placed in memory and then dynamically mapped into cache at run-time. The location of the code in memory determines where it will map in the cache. With a direct-mapped cache, however, a different paradigm is possible. Direct mapping is deterministic, and so we can choose where we wish the task to map to in the *cache* and then work backwards and determine a suitable placement in memory. This is somewhat non-intuitive, but placing data in the cache first is a useful technique, as it allows us to avoid collisions between tasks likely to be in the cache at the same time by ensuring that they map to different cache locations.

DAC 2005, June 13–17, 2005, Anaheim, California, USA.

1.1 Reserving Cache Lines
The goal of code placement is to ensure that high frequency code is in the cache as often as possible. Aside from improving the hit ratio, this has an important side benefit of increasing the energy efficiency of the entire system, as cache hits consume far less energy than cache misses [2]. For embedded systems, this energy savings may be of greater importance than the speedup.

We break code into individual tasks, or sub-programs, which can fit entirely into a cache. We assume that the system we are optimizing has been analyzed carefully and that each task has been assigned a frequency rating based on how often it is likely to be executed [5]. We then reserve part of the cache for the highest frequency code so that it is rarely if ever overwritten.

1.2 Prior Work
How much of the cache should be reserved for the high frequency code? Li and Wolf [3] proposed a system in which the cache is evenly split between high and low frequency tasks. This somewhat simplistic heuristic was chosen to enable the algorithm to do run-time placement. In fact, most of the work done in code placement has been done either at the run time level or at the compiler level [4]. Only rarely has code placement been addressed at the system design level, at which point tasks may be rearranged (although code within a task may not), and the more flexible bound on execution time allows for somewhat more complex solutions. The advantage of design time code placement over compilation time algorithms is that design time placement does not alter the size of the code, while compilation time cache-optimization algorithms may increase code size significantly. A major goal of this paper is to reduce cache-misses *without* significantly increasing the required memory.

This paper proposes an extension of the design-time code placement algorithm proposed by Parameswaran in [1], and our problem statement in Section 3 follows the formulation presented in that work.

2. MOTIVATIONAL EXAMPLE

Consider a two-processor system, where each processor has an instruction cache but both processors share a main memory. We are given a set of tasks to be run by each processor and a static task graph, in which each task has a directed edge to any other task that can possibly follow it in execution order. The weight of edge (u,v) is the probability that task v will follow task u. From this graph we can calculate the expected frequency of execution of each task. Suppose that Processor 1 has a 500 line cache, and Processor 2 a 1000 line cache, and that the following tasks were assigned to each processor:

Table 1: Example System Load

	Task #	Size	Frequency
Processor 1	Task1	300	10
	Task2	400	5
	Task3	200	1
Processor 2	Task4	800	10
	Task5	300	1
	Task6	900	10

So our system has a total of 6 tasks in it. If we place the tasks in memory first come first served, we will get the following placement of code in memory, and resulting cache mappings:

Table 2: Unoptimized code placement

	Task	Memory	Cache Lines
Processor 1	Task1	0-299	0-299
	Task2	300-699	300-499 & 0-199
	Task3	700-899	200-299
Processor 2	Task4	900-1699	900-999 & 0-699
	Task5	1700-1999	700-999
	Task6	2000-2899	0-899

In Processor 1, Task1 has the highest frequency. Suppose it is executed first, and fills lines 0-299. If Task2 is executed, lines 0-199 of Task1 will be overwritten, and if Task3 is executed, the remainder of Task1 will be overwritten. When Task1 is executed, which we expect to happen frequently, it may not be in the cache and might to be copied from memory. In fact, there are only 100 lines in the cache of Processor 1 (lines 400-499) that are never overwritten, and they are not assigned to the highest frequency task. In Processor 2, there are no lines that are never overwritten.

The algorithms described in this paper suggest an alternate way of placing this code in memory. We allow each cache to have an offset which is added to a memory address before it is mapped into that cache, as will be explained below. Without explaining the method here, we present an alternative memory table, with offsets of 400 for Processor 1 and 100 for Processor 2:

Table 3: Optimized Code Placement

	Task	Memory	Cache Lines
Processor 1	Task1	2100-2399	0-299
	Task2	1700-2099	100-499
	Task3	3200-3399	100-299
Processor 2	Task4	900-1699	0-799
	Task5	2700-2999	100-399
	Task6	0-899	100-999

In this optimized memory layout, the first 100 lines of Task1 and the last 200 lines of Task3 in the cache of Processor 1 are never overwritten. The same is true for the first 100 lines of Task4 and the last 200 lines of Task6 in Processor 2's cache. This gain comes at the cost of increased memory fragmentation, with 500 unused lines in the optimized table, necessitating a 17% larger memory. The algorithm in this paper attempts to minimize memory bloat, but it cannot be entirely avoided.

3. PROBLEM STATEMENT

Suppose there are n processors labeled each with a cache. The sizes of the caches need not be uniform, but we assume that only Level 1 caches are available, both for simplicity and because more than one level of cache is unusual in the commodity processors usually used in embedded systems [1]. We also have m tasks, each with an associated size and frequency. A processor has an assigned set of tasks, each of which can fit in its cache. When a task is run it is direct-mapped from memory into the cache of the processor it will run on. We can relax the direct-mapping condition slightly by assigning to each cache an offset that is added to all memory addresses before mapping. Such an offset is simple to implement, and allows for more powerful algorithms.

For each cache, we wish to reserve for the highest frequency tasks a maximum number of cache lines to never be overwritten. To determine how many lines we can possibly reserve, we fill the cache with as many highest frequency tasks as will fit and call the total number of cache lines used '*HighFrequency*'. The size of the largest remaining task limits how many cache lines can be reserved. Let us call the size of the largest remaining task '*Largest*' and the size of the second largest remaining task '*2ndLargest*'. For a cache of size '*CacheSize*', we can reserve exactly *CacheSize* - *Largest* + min(*CacheSize* - *HighFrequency*, *Largest* - *2ndLargest*) cache lines for the high frequency tasks *HighFrequency*, and still have room to place the remaining tasks.

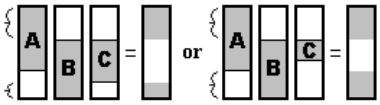

Figure 1: Maximum number of never-overwritten cache lines (the gray areas of the fourth bins are never overwritten)

Addresses in memory that map to the reserved area cannot be used for low frequency code, and so there will of necessity be unused gaps between tasks in memory. Tasks from other processors' tasksets, since they are never written into the current cache, can be placed into the gap areas. Our goal is to find the mapping of tasks into memory that has the least total fragmentation in memory without placing any low frequency code where it would map to the reserved section of its cache.

3.1 Local Problem vs. Global Problem

We need an algorithm for mapping code into the cache that maximizes the reserved area. This algorithm is run once for each cache, and so we refer to it as the local problem. The local problem is to assign to each task a set of positions in the cache where it can be mapped without violating the reserved area. We need a second algorithm to take tasks from all the caches and

place them in the shared memory so that each task maps into a legal cache position as computed by the local algorithm, and memory fragmentation is minimized. We call this the global problem.

3.2 Difficulty of the Problem

If the offsets of all the caches are set to 0 then the global problem reduces to bin-packing, [1] which is NP-Hard [7]. Assigning an offset to a cache is equivalent to rearranging the contents of the bins in the standard bin-packing problem in the middle of packing, since it alters the set of legal positions where the task may be placed. Rearranging the bins of the bin-packing algorithm makes the problem harder, and so the global problem with offsets is at least NP-Hard as well. The identification of the global problem (with the offsets set to 0) with bin-packing might imply that we should employ some form of the best-fit heuristic, which is known to have a worst-case 22% deviation from the optimal solution for bin-packing [7]. Unfortunately, the mapping constraints on the global problem make this standard solution inapplicable in its proven form.

4. LOCAL PROBLEM SOLUTIONS

The local problem can be solved exactly in θ(nlogn)time. We have already shown in the problem statement that the maximum possible size of the reserved area = *CacheSize – Largest* + min (*CacheSize – HighFrequency*, *Largest – 2ndLargest*)

4.1 Parameswaran's Solution

Parameswaran in [1] gave an $\theta(n^2)$ algorithm for the local problem, which presented two difficulties. First, while it reserves the *CacheSize-Largest* area of the cache, it fails to reserve the min(*CacheSize-Frequent*, *Largest-2ndLargest*) area. As well, this algorithm finds only a single legal position in the cache for each task, while in fact there may be a range of legal positions.

4.2 Proposed Solution

The algorithm describe below addresses both of these concerns:

- Sort the tasks in order of descending frequency
- Until the cache is full, allocate the remaining task with the highest frequency to the first unused cache position
- Set the *flex* value of all the tasks allocated so far to 0
- Set *HighFrequency* = the total size of the placed tasks so far
- Sort the remaining tasks in order of descending size
- Allocate the largest remaining task so that it maps to the bottom of the cache, and set its *flex* to 0
- Set *Top* = the cache position where that task begins
- Set *Bottom* = the maximum of *HighFrequency* and (*Top* – *size* of the next largest remaining task)
- For each remaining task:
  - Map the task to *Top*
  - Set the *flex* of the task to *Bottom* – *Top* – (*size* of the task), or to 0 if this is a negative value

This algorithm is θ(nlogn) since unlike Parameswaran's algorithm, the final step does not attempt to fit the remaining tasks into existing bins. The nlogn cost is from sorting.

The *flex* term introduced in this algorithm is the range of legal positions for a task so that the task lies between *Top* and *Bottom*. A task will be allowed to map into the cache starting anywhere from address *Top* to (*Top*+*flex*). By forcing tasks to lie above *Bottom* and not just above the actual bottom of the cache, we have reserved the min(*CacheSize-Frequent*, *Largest-2ndLargest*) area. The never-overwritten code in this area will be from the bottom of the largest remaining task after the high frequency code has been placed.

5. GLOBAL PROBLEM SOLUTIONS

After running the local algorithm on each cache, every task has been mapped to an address in its cache, and assigned a flex, or legal range that the start address can be shifted downwards by. The global problem is to place every task in memory so that its address satisfies the formula:

Equation 1: *MemoryAddress* = *CacheAddress* + *C* * (*CacheSize*) + (*CacheOffset*)

where *C* is a nonnegative integer, and *CacheOffset* is an integer in the range of 0 to *CacheSize*, referring to the cache associated with this task. We want to place in memory the tasks from all the caches, according to the constraints given by the local algorithm, while minimizing the amount of memory used. The minimization clause is the difficulty, since given an unlimited amount of memory the tasks from each processor could be placed far apart from each other and satisfy the local constraints.

5.1 Choosing an Offset

For each cache there are as many possible offsets as cache lines, and while we could go through each of them for every cache, we would have to place every task in memory with each possible offset for each of the caches to determine which offset is best. This would take exponential time, so we need to choose an offset heuristically. If we assume that the largest task in a cache is the most difficult to place (since it requires the largest available gap in memory to be placed into) then it might make sense to choose the offset that helps place the largest task, without considering whether this is the best offset for the remaining tasks in the taskset. This is the approach Parameswaran uses, and although we experimented with several other heuristics it is the approach which we eventually adopted as well.

5.2 Proposed Solution

In the algorithm proposed in [1], all of the tasks associated with a particular cache are placed in memory before any tasks from the next cache, and the algorithm only considers one possible cache position for each task. The algorithm described here addresses both of these issues. First, when we place a task we consider the full range of *Top* to (*Top* + *flex*). Second, we place tasks in descending order of size, regardless of associated processor. When implementing the algorithm, this requires some extra care to ensure that each cache's offset is properly applied to all tasks in that cache, since tasks from a given cache are no longer placed at once. The algorithm is as follows:

- Place all of the tasks in a single list and sort in order of descending size
- For each task:
  - If this is the first task from its cache to be placed:
    - Place the task at the start of the first memory gap large enough to hold it

        - Solve Equation 1 for the offset term, and set the *CacheOffset* of the task's cache to this value
    - Else
        - $C = 0$
        - Until we find an unused memory gap with a start address that satisfies Equation 1:
            - Increment C and loop through *CacheAddress* = *Top* to (*Top* + *flex*), testing Equation 1 with each set of values
        - Place the task at this address

The run time of this algorithm is $\theta(n^2k)$ where n is the number of tasks and k is the size of the largest cache. This is because each of the n tasks may need to be compared to as many as $\theta(n)$ gaps in the worst case where each task creates a new gap when it is placed, and for each gap the algorithm must consider up to *flex* positions, where *flex* is bounded above by the size of the cache and therefore by k.

6. SIMULATION AND RESULTS

We chose to test the algorithms on a system with four processors. The caches were allowed to vary randomly from 1K to 32K in size, and the number of tasks was varied systematically from 28 to 168. Task sizes were random, but always chosen to fit inside the cache. The largest task in each cache was given the highest frequency, to give worst-case performance. We ran the simulation 20 times for each system load and averaged the results. On each run of the simulation, a random dataset was created and both Parameswaran's and our algorithm were run on the same data. We then calculated the total size of all tasks, which is the minimum amount of memory needed to hold all the code, and calculated how much more memory the placement algorithms required. We give the results in the form of percent wasted memory. The mean results for 20 runs of the simulation are presented in **Figure 2** and **Table 4**. Parameswaran's algorithm is roughly invariant to system load, while the proposed algorithm *improves* very fast as system load grows, performing better on large tasksets than on smaller. In addition, our algorithm is more consistent, with a smaller standard deviation than Parameswaran's algorithm.

Table 4: **Overall results**

	Parameswaran	Proposed
Mean % Wasted Memory	15.815	4.403
Worst Case % Wasted Memory	29.198	13.526
Mean Standard Deviation	4.061	2.018

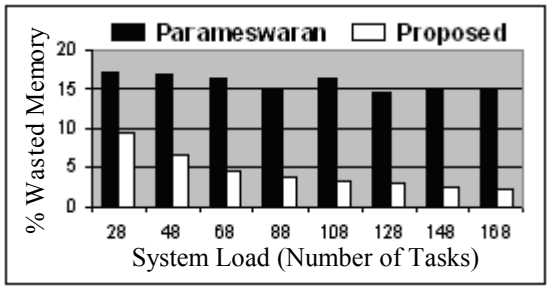

Figure 2: Average results over 20 runs

7. CONCLUDING REMARKS

Although we focused on embedded multiprocessors, this work has some broader applications. Large specialized multiprocessor systems such as scientific clusters are often programmed using a paradigm called *Distributed Shared Memory* in which each processor maintains a local memory, and a logical shared memory is implemented over the network by having local memories update each other when necessary. A similar code placement algorithm for these systems could ensure that code remains in the faster local memory whenever possible. Even for single processor embedded systems, our algorithm can serve a useful purpose. Parameswaran [2] outlines a method for isolating "superloops," or section of code which references code within the section frequently but other code only rarely. A superloop can be placed in memory so that higher frequency code within the superloop remains in the cache whenever the superloop is active. In practice, isolating superloops is very difficult and for a given program may not be possible, which is one of the reasons why we chose to focus on the multiprocessor case.

8. ACKNOWLEDGMENTS

My thanks to Dr. Bhaskar Sengupta of ExxonMobil for guiding me as I researched this problem, and to Dr. Luca Carloni of Columbia University for his help in assembling this paper.

9. REFERENCES

[1] Parameswaran, S. "Code placement in hardware/software co-synthesis to improve performance and reduce cost." In *Proceedings of the Conference on Design, Automation and Test in Europe*. pages 626-632, 2001

[2] Parameswaran, S. "I-CoPES: fast instruction code placement for embedded systems to improve performance and energy efficiency." In *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*. pages 635-641, 2001

[3] Yanbing, L. and W. Wolf. "A Task-Level Hierarchical Memory Model for System Synthesis of Multiprocessors." In *Proceedings of the 34th annual conference on Design automation - Volume 00*. pages 153-156, 1997.

[4] Tomiyama, H. and H. Yasuura. **"**Size-Constrained Code Placement for Cache Miss Rate Reduction." In *Proceedings of the 9th International Symposium on System Synthesis*. pages 96-104, 1996.

[5] McFarling, S. "Program Optimization for Instruction Caches." In *Proceedings of 3rd International Conference on Architectural Support for Programming Languages and Operating Systems,* pages 183-191, 1989.

[6] Hennessy, J.L. and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc. 2nd edition, 1996.

[7] Corman, T., et. al. *Introduction to Algorithms*. MIT Press, 2001.

[8] Tannenbaum, A. *Structured Computer Architecture*. Prentice Hall, Inc. 4th edition, 2001