A Code Layout Framework for Embedded Processors with Configurable Memory Hierarchy

Kaushal Sanghai Alex Raikman
Ken Butler
Analog Devices
Norwood, MA 02062, USA
kaushal.sanghai@analog.com
alex.raikman@analog.com
ken.butler@analog.com

David Kaeli
ECE Department
Northeastern University
Boston, MA 02115, USA
kaeli@ece.neu.edu

Abstract

Several embedded processors now support configurable memory hierarchies to exploit application specific workload characteristics. To take advantage of memory reconfigurability, automated software optimization techniques are generally lacking and application developers often resort to a hand tuned code layout. This not only increases the time to market of embedded products but may also result in an inefficient mapping.

To address this issue, we have developed a framework which incorporates profile guided code layout algorithms to efficiently map code on the available L1 code memory configurations. Given a L1 memory configuration, we show that the code layout problem can be mapped to a NP-complete Knapsack problem or to a combination of Knapsack problem and a graph coarsening problem.

In this paper we present the performance benefits of applying our framework to the multimedia applications from the EEMBC consumer benchmark suite. We evaluate the code layout algorithms on the ADI's Blackfin single core embedded processor.

1. Introduction

Configurable memory hierarchies are now commonly available in embedded processors. For example, the ADI's Blackfin family of processors [1] and LSI Logic's CW33000 Risc microprocessor core [2] support on-chip L1 memories which can be partitioned to behave as L1 SRAM and/or L1 Cache.

When the L1 memory is configured as L1 SRAM, the code is mapped statically on to the L1 memory. The code mapped to the L1 SRAM memory resides throughout the execution of the program and is never evicted from the L1 memory. This ensures that a core request to a L1 SRAM has guaranteed single cycle access and is not subject to a cache miss. A part of the L1 memory can be configured to behave as cache and this part of the L1 memory is termed as L1 Cache. In the case of L1 Cache, an instruction fetch may result in a cache miss and subsequently resulting in a core access to the external memory. The code which resides in the cache is dictated by the cache replacement policies. When the L1 memory is partitioned as L1 SRAM and L1 Cache, then we term it as L1 SRAM/Cache configuration.

Given such a memory subsystem, performance can be greatly increased by exploiting application specific workload characteristics. For example in case of L1 SRAM memory, one can map functions with real time criticality and most of the hot code sections in the L1 SRAM space. This eliminates most of the cache misses which in turn reduces the core access to the slower off-chip external memory. In effect, both, the average memory access time and the external memory bandwidth requirements are reduced. But sometimes not all of the critical code sections may be mapped in the available on-chip L1 SRAM space and therefore the remaining code sections have to be mapped to the off-chip external memory. To reduce the memory access latency in such cases a part of L1 SRAM can be configured to behave as cache.

To take advantage of a heterogeneous SRAM and cache memory architecture, automated software optimization techniques are generally lacking. Thereby, an application developer often resorts to a hand tuned code layout which not only increases the time to market of embedded products but may also result in an inefficient mapping. To address this issue, we have developed a framework which incorporates automated code layout techniques for the various L1 memory configurations. The framework uses the execution profile information and the L1 code memory configurations to produce an efficient code layout.

For just the cache memory, considerable amount of research has been done in the area of efficient code mapping [3, 4, 5, 6, 7]. Prior work [3] has developed code layout algorithms wherein, they map frequently called caller/callee functions in contiguous memory locations. More advanced algorithms for code reordering were developed by [4, 5, 6, 7]. For an L1 SRAM memory, we [8] mapped the code layout problem to a Knapsack problem [9]. Others have formulated the problem of mapping for SRAM memories as an Integer Linear Programming(ILP) problem [10, 11, 12, 13, 29].

For the heterogeneous SRAM and cache memory architecture we show that the code layout problem can be mapped to a combination of Knapsack and a graph coarsening problem. We also achieve better layout by ordering the code sections based on temporal reuse information.

For the L1 SRAM/Cache configuration we can partition the code and map part of the application code in the L1 SRAM memory and map the remaining in the external address space. To efficiently utilize both the SRAM and cache memory, we develop strategies such that they mutually benefit each other. For example,a cache memory can help to further save the L1 SRAM memory space. In the presence of cache, functions with high temporal locality which were originally mapped to L1 SRAM can be displaced to external memory. The functions with high temporal locality now mapped to the external memory would be friendly to cache and thus would not degrade the performance. Also, by placing hot functions and low temporal code in SRAM most of the cache conflicts are eliminated. Thus, for the remaining code that has to be mapped to the external memory very simple code layout techniques [3] for cache memories can be adopted.

We combine the code layout algorithms for the L1 SRAM configuration [8] and the L1 SRAM/Cache configuration in one common framework. We evaluate the code layout algorithms on six multimedia encoder/decoder algorithms from the EEMBC [14] consumer benchmark suite. In this work we use a ADI Blackfin 533 processor installed on a EZkit hardware board [1]. We show performance improvements ranging from 6% to 35% for different L1 SRAM/Cache configurations using our code layout algorithms.

The rest of the paper is organized as follows. In the next section we describe the memory architecture we are targeting. Section 3 presents the code layout algorithms for different L1 memory configurations. In section 4 we discuss the framework and its major components and section 5 describes the methodology of our evaluations. We present the results for the code layout algorithm for the L1 SRAM/Cache configuration in section 6 and some discussion in section 7. Finally, we discuss some related work and conclude in sections 8 and 9, respectively.

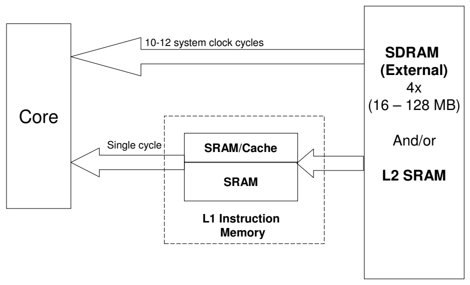

Figure 1. Instruction memory architecture for ADI's Blackfin family of processors.

2. Memory Architecture

In this section we will discuss the memory hierarchy model of the more recent embedded processors. We also describe the possible memory configurations and the tradeoffs involved in selecting a particular layout.

2.1 Memory model

A typical memory hierarchy model for some of the modern embedded processors is shown in figure 1. The on-chip L1 SRAM provides single cycle access to the core. Part of L1 SRAM can be configured to behave as cache. The L2 memory is optional and some processors may not implement it. If implemented, an L2 memory can be either configured as SRAM or cache. L2 access times are typically much longer than L1, but provide better performance than accessing off-chip SDRAM. For the rest of this paper we will omit L2 memory from the discussion, and focus on optimizations for cores with a single level of on-chip memory. The external memory is SDRAM and access to SDRAM generally results in a delay of several system clock cycles. A similar architecture for L1 data memory is discussed in [11] in more details.

2.2 Memory configurations

Given such a L1 memory model, we can have three possible configurations, although some processors could provide additional options.

1. **L1 SRAM:** In this configuration there is no cache and the entire L1 memory is available as SRAM memory. If the entire code fits in the L1 SRAM then this would provide the best performance. For code size greater than the L1 SRAM size, the code that spills over to the external memory will incur a delay of several system clock cycles if requested by core. For this configuration to be effective, only a very small number of code accesses should access external memory.
2. **L1 Cache:** This is similar to the L1 memory architecture prevalent in most general purpose microprocessors. All the code resides in the off-chip external memory and is cached in the L1 memory upon core request. The disad-

vantage of a cache memory is that it increases the *worst case access time* [15] which is critical for real time applications. It also increases the external bus bandwidth requirements which may prove costly for applications with streaming data buffers in external memory. Minimizing external bus bandwidth requirements is always a challenge in embedded systems. Also, performance may suffer if the application has poor locality behavior.

3. **L1 SRAM/Cache:** In this configuration most of the critical code sections can be mapped to the SRAM. The rest of the code which spills into external memory is cached. If most of the executed code can be placed in the L1 SRAM portion of the memory, then the majority of the compulsory and conflict misses are avoided. This should also reduce external bus bandwidth requirements. The cache can provide low latency access to infrequently executed code.

In the next section we describe the code mapping algorithms for the above L1 code memory configurations.

3. Code Mapping Algorithm

The problem of mapping is constrained to a combinatorial optimization problem based on the L1 memory configuration. In this section we discuss the code mapping algorithms for the L1 SRAM/Cache configurations in detail. The code mapping for L1 SRAM is described in [8] and L1 Cache configuration has been studied extensively in prior research and is not the targeted configuration for this work. Thus, we only summarize the algorithms and discuss only the relevance of L1 SRAM and L1 cache as they relate to a more heterogeneous L1 SRAM/Cache configuration.

3.1 L1 SRAM Code Mapping

For an efficient L1 SRAM code mapping, one approach would be to simply map the most frequently executed code (i.e., the hottest code) sections in L1 SRAM based only on their relative execution percentage. But the layout can be more effectively guided by also weighting the code sections by their corresponding binary size. This leads us to a very familiar NP complete Knapsack problem [9].

In the Knapsack problem every object is characterized by its value and weight. The objective is to maximize the sum of values of the objects in the Knapsack given a bound on the total weight of the Knapsack. This problem is also known as the 0/1 Knapsack problem. For the L1 SRAM code layout problem the goal is to maximize the execution percentage(value) of the code sections mapped to the L1 SRAM memory (Knapsack) given the constraint on the total size (weight) of the L1 SRAM memory.

If E_i is the execution percentage of each code section, and S_i is the size of the corresponding code section then the problem can be formulated as follows:

$$max\left(\sum_{i=1}^{n} E_i\right) \qquad (1)$$

in the L1 memory, with the constraint that:

$$\sum_{i=1}^{n} S_i \leq \text{L1 memory} \qquad (2)$$

The 0/1 Knapsack problem is a NP complete problem but by using greedy heuristics, near optimtal solutions can be obtained. We incorporate a simple greedy algorithm in the framework and it produces close to optimal solutions quickly.

Using the above equations, we use the execution percentage of each code section, code sizes, and the L1 memory size to drive the algorithm.

Greedy Algorithm: We compute the *hotness ratio* for a code section as its execution percentage in the overall program execution time over size. The code sections are sorted in decreasing order of hotness ratio. The sorted code sections are then added to the Knapsack until the L1 memory size bound is reached. As each new object is added, the L1 SRAM size is checked. If the L1 SRAM size is exceeded, the code section is not included and other objects are considered. In this way the algorithm obtains a solution as close as possible to the total L1 SRAM size bound.

3.2 L1 Cache Code Mapping

As discussed before, a L1 instruction cache memory is prevalent in most general purpose microprocessors and has been the focus of most prior work [3, 4, 5, 6, 7] in efficient code layout. In [3], Pettis and Hansen use the function caller/callee frequency count and a ”closest is best” strategy to map code sections. By mapping caller/callee function pairs to contiguous memory locations, cache conflicts are reduced. The intuition behind this is that there is high temporal locality exhibited by caller/callee function pairs. Thereby, mapping caller/callee in contiguous memory locations in the external address space, a core fetch to those functions would result in them going to different cache lines and thus should reduce cache conflicts.

Subsequent techniques have used more advanced algorithms such as cache coloring [4], temporal reference graphs [7, 5], conflict miss graphs [6] for procedure reordering. For the memory model we are targeting most of the conflict misses are eliminated by mapping the most valuable code sections in L1 SRAM as determined by the L1 SRAM code layout algorithm. Thus, we do not need to incorporate more sophisticated algorithms for L1 cache layout.

3.3 L1 SRAM/Cache Code Mapping

In the L1 SRAM/Cache configuration we can partition the code sections to be mapped in the L1 SRAM and external memory. The code layout algorithm should aim to effectively utilize the L1 SRAM space, as well as produce an efficient layout for the L1 Cache memory.

To take advantage of L1 SRAM and cache memories we define the following three objectives the code layout algorithms should achieve

1. For the L1 SRAM memory the objective is to map the most valuable functions in L1 SRAM such that we maximize the execution percentage from L1 SRAM memory.
2. To place functions with low temporal locality in L1 SRAM and higher temporal locality functions (cache friendly) in external memory.
3. For the L1 cache memory, the objective is to minimize cache conflicts. Therefore the code layout algorithm should place functions in external memory efficiently.

In the next section we further describe how we could use the temporal reuse measure to further save the L1 SRAM space and use a similar technique developed in [3] to produce an efficient cache layout.

3.3.1 L1 SRAM Mapping

By using the execution percentage and size of the code section, we would be able to map most of the executed code in L1 SRAM. But by using the temporal reuse information of the code sections we can further save space in L1 SRAM memory by only mapping functions with poor temporal locality in L1 SRAM. This also ensures the cache performance would be efficient since only functions which exhibit good locality have been mapped to the external off-chip memory.

There could be many code sections that possess poor temporal locality. Therefore, it is important to consider their hotness ratio in order to prioritize code sections to be mapped in L1 SRAM. We measure the temporal locality of a function by computing its reuse distance [16]. If a function has a large reuse distance (i.e., poor temporal locality) then it is accessed less frequently. We use the reuse distance metric to guide code layout as follows:

If a function has a large reuse distance, it is likely that it will be evicted from the cache and so mapping it to external memory should not degrade the performance. Similarly, if a function has a small reuse distance, there is less of chance for it to be replaced in the cache. Thus, if a function has low temporal reuse and high hotness ratio it should have higher priority for its placement in L1 SRAM. We can use the cache for managing cache-friendly code.

Greedy Algorithm: Functions responsible for less than 1% of relative execution are pruned from the call graph. We sort the remaining functions by further dividing their hotness ratio by temporal reuse measure. Then we apply our greedy layout algorithm as described in section 3.1 to identify the code sections that will be placed in L1 SRAM.

3.3.2 L1 Cache mapping

The remaining code is mapped to the external off-chip memory and is subsequently cached in L1 cache. By using the function call graph, we can formulate code layout algorithm as an undirected graph coarsening problem with weighted edges and nodes. Every function is assigned a node and the caller/callee pair is connected via an edge. Edges are weighted by the call frequency of the corresponding caller/callee pair. The nodes are weighted by their binary size.

We map the most called caller/callee function in contiguous memory locations [3]. But instead of using just the ”closest is best” strategy as is used in [3] we also take the cache configuration and the size of the functions as an input to our algorithm.

Greedy Algorithm: The input to the algorithm is the L1 cache configuration and the weighted call graph. To produce coarser sub graphs we merge connected nodes based on the relative edge weights. The algorithm can be briefly explained in the following three steps.

Step 1: Sort the edges of the call graph by the frequency count caller/callee pair (edge weight).

Step 2: Set the threshold on the maximum size of the merged node (coarser subgraph). This is equal to the cache line size (denoted as M_{th}).

Step 3: *For* all edges in the sorted list, start merging nodes. Given that the ith edge connects nodes A and B, and given that S_A and S_B denote the size of A and B, respectively, nodes that are grouped together are assigned a common merged node id.
Case 1: For both A and B, no merged ID is assigned and $S_A + S_B \leq M_{th}$. Merge nodes A and B by assigning them a common merged node id.
Case 2: A has already been assigned a merged node id (A is already part of a merged node).
If the total size of the merged node containing A would exceed M_{th} if B would be added,
then assign a new merged node id to B.
else merge B with the node containing A.
Case 3: Only B is assigned the id. This is the same as case 2, though just exchange A and B.
Case 4: Both A and B have assigned ids.
continue with the next edge;
end for;
end algorithm;

Thus, we map the functions as specified by the Knapsack problem to the L1 SRAM space, and the remaining functions are mapped to external memory based on the solution of the graph coarsening problem.

3.4 Conflicting optimization techniques

Compiler optimizations such as function inlining could also reduce the cache conflicts but it would also increase the

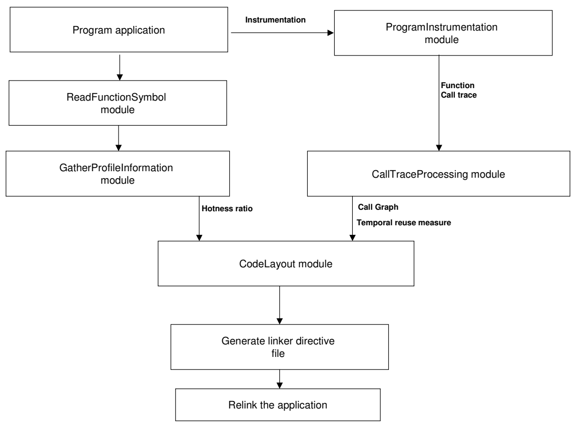
Program application
Instrumentation
ProgramInstrumentation module
Function Call trace
ReadFunctionSymbol module
GatherProfileInformation module
CallTraceProcessing module
Hotness ratio
Call Graph
Temporal reuse measure
CodeLayout module
Generate linker directive file
Relink the application

Figure 2. Framework for L1 code memory layout

code size. By changing only the mapping of the functions at link time, the code size remains the same. Also, statically analyzing the program behavior can help to improve code layout, though little control flow information is available at compile time [17]. Also hardware (i.e., cache organization) can be used to further reduce conflicts, though tends to be more appropriate for high performance versus embedded platforms.

4. The Framework

The previous sections described the code layout algorithms and the greedy algorithms for the available L1 memory configurations. The entire process from gathering profiles, selecting a particular layout algorithm to producing linker directive files is integrated in a framework. The framework allows for design space exploration to optimize the code memory layout for an embedded system implementation.

Figure 2 shows the major framework components and the process steps involved in producing the final layout. The current framework only supports mapping at a function granularity, but can be extended to finer granularities such as basic blocks. The framework starts with the ReadFunctionSymbol module. It reads the function symbol name, function size, function memory start address from the elf header embedded in the executable. The application is then executed on the target processor and the GatherProfileIinformation module stores the execution frequency of each function obtained from the profile run.

The ProgramInstrumentation module instruments the code at a function granularity to generate the function call graph. The collected call trace information is processed to build the call graph and provides temporal reuse information per function. Every function is assigned a node and the caller/callee frequency counts are represented by edge weights. It should be noted that if the user only wants an efficient L1 SRAM layout then the instrumentation and call graph construction steps can be skipped.

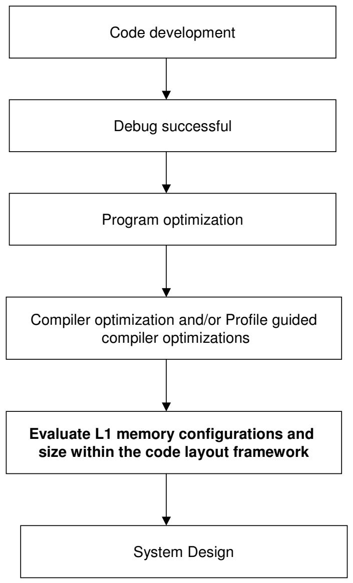
Code development
Debug successful
Program optimization
Compiler optimization and/or Profile guided compiler optimizations
Evaluate L1 memory configurations and size within the code layout framework
System Design

Figure 3. Framework usage in the overall embedded system implementation process.

The execution percentage and size of each function along with the call graph information (if instrumentation option is exercised), are then passed on to the CodeLayout module. The framework incorporates the algorithms for the two different L1 memory configuration options, L1 SRAM and L1 SRAM/Cache. We assume that a user would never select the L1 cache configuration since an L1 SRAM is always available for the processors targeted in this work.

A new set of directives are required to specify the memory mapping for each code section. The Output module produces a linker directive file where every function symbol name is labeled with a directive. This file is added to the project and the project is re-linked to produce an efficient layout.

To provide more information to the user, a plot of L1 memory size vs. the execution coverage out of L1 SRAM memory can also be displayed within the framework. Typically, different L1 memory sizes are available for a given family of processors and therefore an optimal L1 SRAM size could be determined from the graph.

Figure 3 shows where in the embedded system implementation, the framework can be exercised. It should be noted that if the sources or compiler switches are changed, then the entire process for code layout should be repeated. This is because optimization can affect the size of the code sections and the associated profile information.

5. Methodology

All the experiments are performed on ADI's Blackfin EZ-kit hardware board. VisualDSP++ 4.0 is used as the development environment for compiling, linking and building ex-

Benchmark	Code size(KB)	# of functions
JPEG2 encoder	56	380
JPEG2 decoder	61	388
MPEG2 encoder	84	330
MPEG2 decoder	68	351
MPEG4 encoder	197	480
MPEG4 decoder	131	404

Table 1. Code size (KB) and number of functions in each benchmark.

ecutables for the Blackfin processor. The tool provides a rich set of profiling tools and linking directives necessary for code mapping. The tool also provides a set of automation API's which help us to develop software utilities outside of the development environment (Visual DSP++) to extract information from the tool and process it automatically. These utilities are integrated within the framework and use the automation API's to collect the profiles and symbol information. They communicate to the VisualDSP++ tool via the Windows com interface.

Blackfin processors are designed specifically for consumer multimedia applications. Thus, we have evaluated our code layout algorithms on the EEMBC [14] consumer benchmark suite. If an application's code fits entirely in L1 SRAM, then mapping all of it to L1 SRAM would give the best performance. Therefore, from the benchmark suite we focus on applications which have large code sizes. We have selected the JPEG2, MPEG2 and MPEG4 encoder/decoder benchmark programs which have code size greater than 50KB. These benchmarks are standard video and image encoding/decoding algorithms widely used in multimedia applications.

Before using the framework, the programs are compiled with optimization turned on for maximum performance. The first step in the process is to collect the function symbol information by reading the elf header from the executable and obtaining the execution profile of the functions in the program by executing the program.

The instrumentation module is invoked to generate the call trace. The trace is post processed to produce the call graph. We use the standard inputs contained in the EEMBC benchmark suite for gathering the profile information.

We measured the performance in terms of the number of cycles consumed for program completion. Hardware performance counters available on the Blackfin processors are used to compute the cycle count.

To evaluate the code layout algorithms, we compare the performance for different L1 memory sizes. A part of L1 SRAM is used to map code sections (as determined from the code layout algorithms) and the rest can be used for mapping critical real time functions. In Blackfin, the instruction cache is a 16K 4-way set associative with a 32 bytes line size. It can be configured as 4K direct mapped or 8K 2-way set associative.

We choose a minimum of 8K L1 SRAM memory and a 4K direct mapped cache. We also perform our experiments with a 12K SRAM and 4K cache and a 8K SRAM and 8K cache. The above three L1 memory configurations are compared to the fully available 64K SRAM and 16K cache L1 memory on a Blackfin.

6. Results

Next, we present results for the 6 benchmark programs from the EEMBC consumer benchmark suite. Figure 4 shows the performance benefits obtained for the L1 SRAM/Cache configuration using our code mapping algorithm for the 6 different code layouts. A 12K (8K SRAM and 4K cache) of L1 memory space with no code layout optimizations is chosen as a baseline configuration. For all the six benchmarks the input used is the standard image provided in the EEMBC suite. Each data input comprised of at least fifteen CIF sized frames (352x240) for MPEG encoding or decoding, and a CIF sized image for JPEG encoding/decoding.

In the graph, the first bar shows the relative improvements after mapping the most executed (ME) code sections in the L1 SRAM space and using the default layout for the remaining code sections mapped to external memory. The next two bars show the performance benefits in using the Knapsack(KS) and the temporal reuse (TR) for the L1 SRAM along with the default linker layout for the functions in the external memory. The remaining 3 results are for the same L1 SRAM layout, but maps the remaining code sections using our graph coarsening technique (GC) with the help of a function call graph.

As can be seen in the figure, we achieve an average improvement of 19% for the KN-TR-GC algorithm (which combines the hotness ratio with temporal reuse) for the L1 SRAM layout and the graph coarsening for the external memory layout. While, most of the performance benefits are obtained by mapping the most frequently executed code in the L1 SRAM space, a further improvement of 4% is obtained by formulating the problem as a Knapsack problem. By incorporating temporal reuse information, a further improvement of 1% is possible for some of the benchmarks.

By efficiently mapping the code sections in the external memory, consistent improvement is seen over all the benchmarks. For the MPEG4 decoder, the benefits are as much as 7%. It should be noted that the performance improvements are more pronounced for the MPEG4 programs because of its larger code size as compared to the MPEG2 or JPEG2 programs.

In figure 5 we show the average performance improvement over all benchmarks for three different L1 SRAM/Cache memory size. The baseline is the default layout for the respective L1 memory configuration. The performance benefits for the 12K SRAM and 4K cache are slightly greater

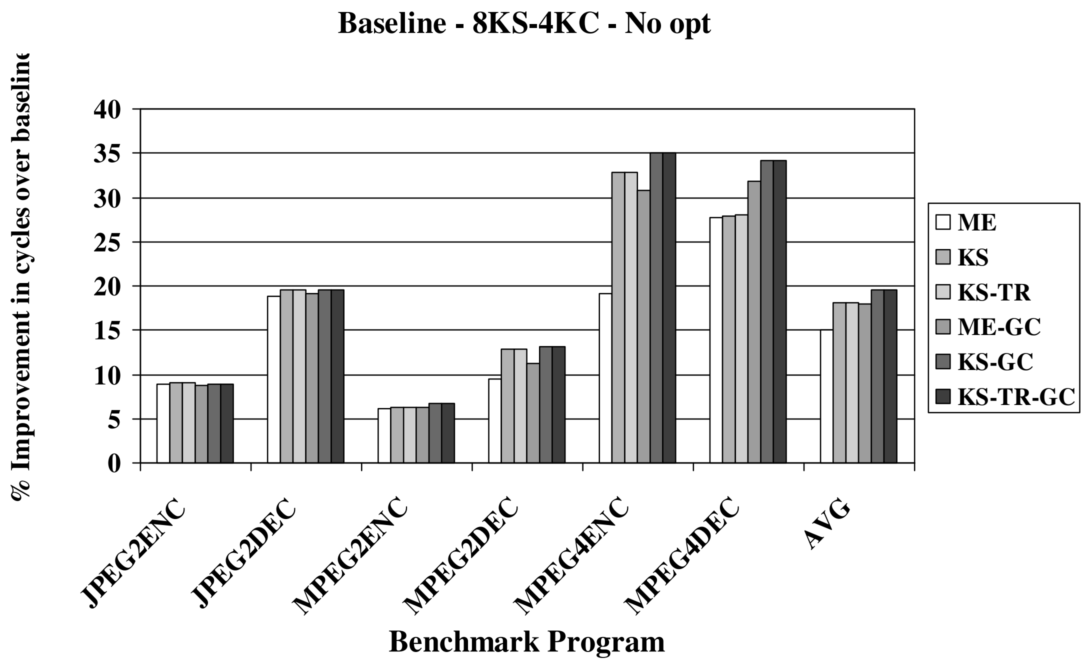

Figure 4. Figure shows the relative performance improvements for JPEG2, MPEG2 and MPEG4 encoder/decoder benchmark programs. A 8K SRAM and 4K Cache with no code layout optimization is taken as the baseline configuration. The relative performance improvements in terms of cycles over the baseline for 6 different layouts is shown. In the graph ME= Most Executed, KS=Knapsack,TR=Temporal Reuse and GC=Graph Coarsening.

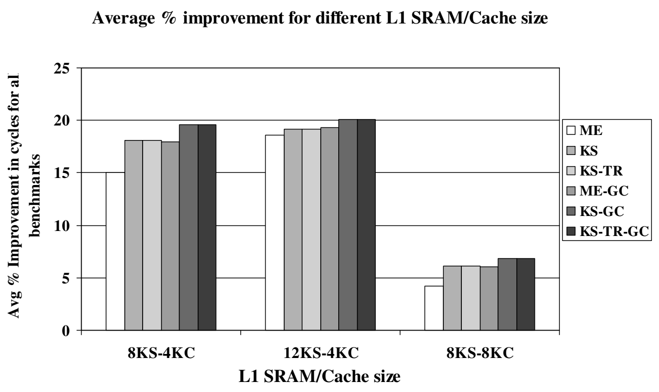

Figure 5. Figure shows the average performance improvements over all benchmarks for three different L1 SRAM/Cache memory size. In the graph ME= Most Executed, KS=Knapsack,TR=Temporal Reuse and GC=Graph Coarsening.

because the default layout does not make effective use of the increased L1 SRAM space. Also, in the case of 8K SRAM and 8K cache, the increased cache size exploits the locality even in the default layout and there is less than 7% benefit in using the code layout algorithms.

But as we can see for both the 12K SRAM/4K cache and 8K SRAM/8K cache L1 memory, the gains in using our advanced code layout techniques (i.e., KN, TR, or GP, as compared to ME) are diminishing. Most of the improvements are obtained by mapping just the hottest code sections or by applying the Knapsack algorithm. If most of the executed code can be mapped to the L1 SRAM space then there is little opportunity for improvements using more advanced algorithms.

To find an upper bound on performance, we used the maximum available L1 memory (i.e. a 64K SRAM and 16K cache) for the Blackfin family of embedded processors. We compare this to the L1 memory size we have already studied.

In Table 2 we show the performance benefit of the default non-optimized and fully optimized layout (i.e., KN-TR-GC) for four different L1 SRAM/Cache sizes. Five out of six of the benchmark programs achieve less less than 1% benefit for a 64K SRAM and 16K cache as compared to a 8K SRAM and 8K cache. Also, for the MPEG4 encoder benchmark program we have found that the performance of 12K SRAM and 8K cache is close to a 64K SRAM and 16K cache L1 memory.

Thus, our code layout techniques would prove beneficial in embedded systems which have low memory budgets. We show that by carefully mapping code we can achieve similar performance using a smaller L1 memory size, which will re-

Benchmark	8KS-4KC-NO-OPT	8KS-4KC-FULL-OPT	12KS-4KC-NO-OPT	12KS-4KC-FULL-OPT	8KS-8KC-NO-OPT	8KS-8KC-FULL-OPT	64KS-16KC-NO-OPT	64KS-16KC-FULL-OPT
JPEG2 encoder	1.41	1.28	1.32	1.27	1.28	1.28	1.27	1.27
JPEG2 decoder	1.39	1.12	1.34	1.11	1.15	1.11	1.10	1.10
MPEG2 encoder	11.76	10.97	11.69	10.89	10.92	10.81	10.87	10.77
MPEG2 decoder	54.41	47.27	53.25	45.92	48.03	44.82	44.37	44.33
MPEG4 encoder	72.45	47.04	70.66	44.79	49.87	44.60	43.01	42.19
MPEG4 decoder	73.31	48.22	73.31	45.43	55.68	45.31	45.29	45.05

Table 2. Table shows the executed cycles (in 10 millions) for default non-optimized and fully optimized layouts for four different L1 SRAM/Cache size.

duce the overall cost and power of the system. The demands for a better multimedia experience are always growing and applications are becoming more complex, resulting in larger code size. Our layout techniques should be extremely helpful in such cases. Our framework can also provide for possible design space exploration in L1 memory size and configurations.

7. Discussion

As a general rule of thumb, our advanced code layout techniques should be used for L1 memory configuration exploration when it is possible to only map 70-90% of the total code in L1 SRAM. If more than 90% of the code can be placed in L1 SRAM memory then the suggested techniques will not have a large impact.

To effectively capture the program characteristics, we profile the application to obtain the execution percentage of each code section, temporal reuse distances, and a function call graph. We present our analysis by treating each of these pieces of information during different algorithm steps to guide our layout. But it should be noted that the program characteristics overlap between these measurements. For example, hot code is generally frequently called code, should also have high temporal reuse. There will be few cases where a hot code has low temporal locality or a hot code is less frequently called. But as we have seen by targeting these cases and equipped with the temporal reuse measure and a function call graph we could improve performance by 3-5% on average.

During our analysis, we also noted cases where it is possible that a relatively cold function is placed in external memory and is frequently calling a hot function placed in L1 SRAM. In such cases the caller/callee functions are separated in the memory address space greater than the address range architected to make a short call. This results in an indirect call in the generated code. This introduces added instructions and requires more cycles to complete each call. If the call is frequent enough, it can degrade performance. We specifically observed this case in the MPEG2 decoder and we placed the cold caller function in L1 SRAM space (which was frequently calling a hot function already placed in L1 SRAM). Although placing the cold function in L1 SRAM displaced a relatively hot function in external memory, we observed a 2% improvement in performance for a 8K SRAM and 4K cache L1 memory size.

Optimizations based on profile guidance tend to suffer when input data sets change. Optimizations need to remain robust and work for changing program inputs. For the EEMBC consumer benchmark suite, when we ran experiments with different input data sets, we find that our results are not impacted. The execution percentages were somewhat insensitive to input changes. In general, instruction profiles are less susceptible to change due to data input changes versus data profiles.

8. Related work

Producing cache conscious code mapping which exploits the spatial and temporal locality within programs has been proposed as an effective way to reduce cache misses. Researchers over the last couple of decades have proposed different techniques for code and data remapping [18, 19, 20, 3, 4, 5, 6, 7].

Accurate program characteristics can be obtained by looking at the execution profile of a program. By carefully choosing a representative input dataset, efficient profile guided optimizations can be developed to increase program performance. Statically analyzing the program [21, 22] for compiler optimizations makes the reordering input data independent but several times it is hard to completely understand program behavior statically.

Basic block profiles and function caller/callee frequency counts have been used in code reordering algorithms. Prior work [3, 4, 5, 6, 7] has shown that procedure reordering during link time can reduce cache conflicts. Therefore, in our framework we use link time procedure reordering to effectively guide the layout but incorporating different algorithms and heuristics for the memory model we are targeting was used in [4, 5].

In [4], Hashemi et al. used a graph coloring algorithm to guide the code layout and in later work by the same group and others [6, 23] temporal reference information is considered. In our approach we use a simple graph coarsening problem instead of a graph coloring algorithm because we have the L1 SRAM memory at our disposal. Also, for the memory model we are targeting, it is not necessary to generate a temporal reference graph. Instead, our algorithm uses a temporal reuse distance measure to partition the code sections and separate cache friendly nodes from the unfriendly ones to guide the placement in L1 SRAM .

In the embedded space, Kumar et al. [12] formulate the code and data layout problem as an integer linear programming (ILP) problem for fixed SRAM memories. But they do not address any code or data placements for effective cache layout. In [24, 25, 11], the authors present data layout techniques for embedded multimedia applications. In particular, Panda et al. [11] present a data layout algorithm for a similar configurable memory hierarchy to the systems we are targeting, but our study focuses on code layout optimizations. Also, as opposed to [11], we present our analysis for a variety of L1 memory sizes.

In [10], Tomiyama et al. formulate the code layout algorithm as an integer linear programming (ILP) problem for direct mapped caches. Our work differs from all of the above in that we address the problem of code layout for a configurable memory hierarchy system. Also, we map the code layout problem to commonly known NP-complete problems and use time-proven greedy heuristics to solve the problem efficiently. In [10], the authors have used local search algorithms to solve the ILP problem. Local search is time consuming and has been noted in their paper as one of the shortcomings of their methodology.

Researchers have proposed techniques to reduce cache conflicts with a variety of techniques as discussed above, but an on-chip L1 SRAM can be effectively used to eliminate all cache misses. In [8], we showed the benefits of using on-chip L1 SRAM for the MPEG-2 encoder benchmark program for both code and data management. The code layout algorithm discussed in our previous work [8] used cache misses as a metric to estimate the cycles lost and guide the mapping. Given that we have caller/callee frequency counts and reuse distances, we do not have to infer the relationship between different functions. Also, in [8] we evaluated function overlays for managing code sections. Our evaluation of our overlay manager was hand-tuned. But by solving the graph coarsening problem, the resulting merged nodes will be automatic candidates for implementation with function overlays.

9. Conclusions

In this paper, we presented a code layout framework for configurable code memory systems now typical in embedded processors. Our tool allows for automatic code layout and extensive design space exploration. This should reduce the time to market embedded products. We presented performance improvements for industry standard multimedia benchmark programs, evaluating them on real hardware using performance counters. We show performance benefits ranging from 6% to a maximum of 35%, when using the default linker layout as a baseline. We achieve a 19% improvement on average for the 6 benchmark programs. Our future work will focus on developing algorithms for data layout and considering tradeoffs in code and data layout in shared memory spaces.

References

[1] Analog Devices Incorporation. Blackfin processor hardware reference manual. Norwood, MA, USA, 2003.

[2] LSI Logic Corporation. Cw33000 risc microprocessor core hardware manual. 1992.

[3] Karl Pettis and Robert C. Hansen. Profile guided code positioning. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 16–27, New York, NY, USA, 1990. ACM Press.

[4] Amir H. Hashemi, David R. Kaeli, and Brad Calder. Efficient procedure mapping using cache line coloring. In *PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 171–182, New York, NY, USA, 1997. ACM Press.

[5] Nicholas C. Gloy, Trevor Blackwell, Michael D. Smith, and Brad Calder. Procedure placement using temporal ordering information. In *MICRO*, pages 303–313, 1997.

[6] John Kalamatianos, Alireza Khalafi, David R. Kaeli, and Waleed Meleis. Analysis of temporal-based program behavior for improved instruction cache performance. *IEEE Trans. Comput.*, 48(2):168–175, 1999.

[7] John Kalamatianos and David Kaeli. Accurate simulation and evaluation of code reordering. In *Proceedings of the IEEE International Symposium on the Performance Analysis of Systems and Software*, pages 45–54, April 2000.

[8] Kaushal Sanghai, David Kaeli, and Richard Gentile. Code and data partitioning on the blackfin 561 dual-core platform. In *ODES-3: Digest of the third workshop on Optimizations for DSP and Embedded Systems*, pages 92–100, 2005.

[9] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[10] Hiroyuki Tomiyama and Hiroto Yasuura. Optimal code placement of embedded software for instruction caches. In *EDTC '96: Proceedings of the 1996 European conference on Design and Test*, page 96, Washington, DC, USA, 1996. IEEE Computer Society.

[11] Preeti Ranjan Panda, Nikil D. Dutt, and Alexandru Nicolau. On-chip vs. off-chip memory: the data partitioning problem in embedded processor-based systems. *ACM Trans. Des. Autom. Electron. Syst.*, 5(3):682–704, 2000.

[12] T.S. Rajesh Kumar, R. Govindarajan, and C. P. Ravi Kumar. Optimal code and data layout in embedded systems. In *16th International Conference on VLSI Design*, pages 573–579, 2003.

[13] Manish Verma, Lars Wehmeyer, and Peter Marwedel. Cache-aware scratchpad allocation algorithm. In *DATE '04:*

Proceedings of the conference on Design, automation and test in Europe, page 21264, Washington, DC, USA, 2004. IEEE Computer Society.

[14] EEMBC. Embedded microprocessor benchmark consortium.

[15] David A. Patterson and John L. Hennessy. *Computer architecture: a quantitative approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.

[16] Thierry Lafage and Andre Seznec. Choosing representative slices of program execution for microarchitecture simulations: a preliminary application to the data stream. pages 145–163, 2001.

[17] Amir Hashemi. Efficient program mapping for improved cache performance. *Ph.D. thesis,Northeastern University, Boston, MA*, 1997.

[18] S. McFarling. Program optimization for instruction caches. In *ASPLOS-III: Proceedings of the third international conference on Architectural support for programming languages and operating systems*, pages 183–191, New York, NY, USA, 1989. ACM Press.

[19] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Cache-conscious structure layout. *SIGPLAN Not.*, 34(5):1–12, 1999.

[20] Shai Rubin, Rastislav Bodik, and Trishul Chilimbi. An efficient profile-analysis framework for data-layout optimizations. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 140–153, New York, NY, USA, 2002. ACM Press.

[21] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Cache miss equations: a compiler framework for analyzing and tuning memory behavior. *ACM Trans. Program. Lang. Syst.*, 21(4):703–746, 1999.

[22] Abraham Mendlson, Shlomit S. Pinter, and Ruth Shtokhamer. Compile time instruction cache optimizations. *SIGARCH Comput. Archit. News*, 22(1):44–51, 1994.

[23] Nikolas Gloy and Michael D. Smith. Procedure placement using temporal-ordering information. *ACM Trans. Program. Lang. Syst.*, 21(5):977–1027, 1999.

[24] Chidamber Kulkarni, Francky Catthoor, and Hugo De Man. Advanced data layout optimization for multimedia applications. In *IPDPS '00: Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing*, pages 186–193, London, UK, 2000. Springer-Verlag.

[25] C. Kulkarni, C. Ghez, M. Miranda, F. Catthoor, and H. de Man. Cache conscious data layout organization for embedded multimedia applications. In *DATE '01: Proceedings of the conference on Design, automation and test in Europe*, pages 686–693, Piscataway, NJ, USA, 2001. IEEE Press.

[26] Sundaram Anantharaman and Santosh Pande. An efficient data partitioning method for limited memory embedded systems. In *LCTES '98: Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 108–222, London, UK, 1998. Springer-Verlag.

[27] Christophe Guillon, Fabrice Rastello, Thierry Bidault, and Florent Bouchez. Procedure placement using temporal-ordering information: dealing with code size expansion. In *CASES '04: Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 268–279, New York, NY, USA, 2004. ACM Press.

[28] Steve Carr, Kathryn S. McKinley, and Chau-Wen Tseng. Compiler optimizations for improving data locality. *SIGPLAN Not.*, 29(11):252–262, 1994.

[29] Manish Verma, Lars Wehmeyer, and Peter Marwedel. Dynamic overlay of scratchpad memory for energy minimization. In *CODES+ISSS '04: Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'04)*, pages 104–109, Washington, DC, USA, 2004. IEEE Computer Society.