Dynamic Code Mapping for Limited Local Memory Systems

Seung chul Jung
Compiler Microarchitecture Lab
Arizona State University
Tempe, AZ 85281, USA
sjung@asu.edu

Aviral Shrivastava
Compiler Microarchitecture Lab
Arizona State University
Tempe, AZ 85281, USA
Aviral.Shrivastava@asu.edu

Ke Bai
Compiler Microarchitecture Lab
Arizona State University
Tempe, AZ 85281, USA
Ke.Bai@asu.edu

***Abstract*—This paper presents heuristics for dynamic management of application code on limited local memories present in high-performance multi-core processors. Previous techniques formulate the problem using call graphs, which do not capture the temporal ordering of functions. In addition, they only use a conservative estimate of the interference cost between functions to obtain a mapping. As a result previous techniques are unable to achieve efficient code mapping. Techniques proposed in this paper overcome both these limitations and achieve superior code mapping. Experimental results from executing benchmarks from MiBench onto the Cell processor in the Sony Playstation 3 demonstrate upto 29% and average 12% performance improvement, at tolerable compile-time overhead.**

I. Introduction

Multicore architectures are becoming popular since they provide a way to improve peak performance without much increase in the power consumption. In addition, several other parameters, e.g., power, temperature and reliability can be more easily managed at coarser granularity of thread- and core- level [15]. As we transition from few- to many-core processors, scaling the memory becomes one of the most important challenges. Managing the memory automatically in the hardware, transparent to the application, is increasingly becoming infeasible. This is not only because caches consume a lot of power (almost 1/2 of the processor power may be consumed by the caches [16]), but also because ensuring coherency between the caches makes them slow, and is difficult to scale to a large number of cores [7]. Consequently, distributed memory systems, in which each core has access to only a limited local memory seems to be the only memory scaling option. High-end and futuristic processors are being designed with such memory architecture. Examples are the network processors [17] and the experimental 80-core processor [24] from Intel, and the IBM Cell processor [7]. The synergistic processing units or SPUs in the IBM Cell have access to a limited 256 KB of local memory, but can access the global memory through explicit DMA requests.

In processor cores with limited local memories, if the whole application fits into the limited local memory, it executes extremely efficiently. If not, then memory management – which must be done in software – is needed. Code or data which is needed next must be explicitly brought into the limited local memory before it can be used. This may include evicting the not-so-urgently needed code and data out of the limited local memory. While management is needed for all code and data, in this paper we focus on code management, since efficient code management can very significantly impact the performance of the system. To facilitate this code management, the IBM Cell processor provides an overlay mechanism. In a linker script, user specified regions and specified the mapping of functions into regions. Functions mapped to one region are mapped to the same physical location in the limited local memory, and replace each other when called. , the size of region is equal to the size of the largest object mapped to the region, and the total code space required is the sum of the sizes of the regions. The goal of code mapping problem is then to generate a linker script, which minimizes the swapping of functions so that performance can be improved due to reduced data transfer between the global memory and local memory.

The inputs to this code mapping problem are traditionally, i) the maximum size of code region, and ii) a call graph, in which the nodes represent functions, and a directed edge between two functions denotes a caller-callee relationship. The weight on the edge is the number of times the caller calls (or is expected to call) the callee. Finding the number of regions, and the mapping of objects to regions, that will minimize the data transfers, both are shown to be intractable [19], [25]. Consequently, several heuristics have been proposed. These heuristics define a metric of *interference* between two functions as the amount of data that needs to be transferred when two functions are mapped to the same region. All existing heuristics are greedy, and place one function into a region in each step.

The approach in this paper removes several limitations of previous approaches.

- While call graphs capture the aggregate information about how many times a function calls another, it does not capture the temporal information about the sequence in which the functions are called. To accurately estimate the data transfers, this temporal information is essential. We formulate our problem using GCCFG, which captures both the aggregate and temporal information about the function calls.
- The interference between two functions, i.e., the amount of data transfer required when two functions are mapped in the same region is dependent on where the other functions are mapped. Therefore even though our heuristic is also greedy, but we update the interference cost between functions as the other functions are mapped.

As a result of these two fundamental reasons, our heuristics result in superior (better runtime) mapping at minimal and

tolerable compile-time overhead. We apply our heuristics to solve the code mapping problem for several benchmarks from the MiBench suite [9], and demonstrate an average of 12% runtime improvements on the IBM Cell processor.

II. Motivating Example

This section provides an example to illustrate two ideas i) interference cost between two functions depends on where the other functions are mapped, and ii) updating interference costs can lead a code mapping to minimize the data transfers between the limited local memory and the global memory.

Figure 1 (a) shows a simple call graph in which function F1 calls F2, F2 calls F3, and F3 calls F4, and then they all return. The function nodes also indicate the sizes of each of the functions. Assume that we have to map all these functions into a local memory of 3.5 KB. Figure 1 (b) shows the interference graph for this application. In this graph, nodes are functions, and the edge weights represent the amount of data transfers required in the worst case, when the two functions are mapped to the same region. For example, if the functions F1 and F2 are mapped to the same region, then they will replace each other two times. First when F1 calls F2, 1.5 KB of function F2 will need to be brought into the local store. Note that F1 does not need to be committed back to the main memory, since this is code, and it does not get "dirty". When function F2 returns to F1, 2 KB of the code of F1 needs to be brought into the local store. Therefore a total of 2 KB + 1.5 KB = 3.5 KB of data transfer is required when functions F1 and F2 are mapped to the same region. Similarly the interference of the direct edges, e.g., F2-F3, and F3-F4 are calculated.

For the indirect edges, the weight calculation is slightly tricky. Consider F1-F3; if F1 and F3 are mapped to the same region, the interference between them depends on where F2 is mapped. For example, if F2 is mapped to a different region (other than that of F1 and F3), then the interference between F1 and F3 is just sum of their sizes, i.e, 2 KB + 0.4 KB = 2.4 KB. However, if all F1, F2 and F3 are mapped to the same region, then the interference cost between F1 and F3 is 0. This is because, when F3 is called, F1 is already replaced with F2, and when the program returns to F1, F3 is already replaced. In a sense, there is interference between F1 and F2, and between F2 and F3, but there is no interference between F1 and F3.

Previous approaches computed the worst case interference cost, i.e., 2.4 KB for F1 - F3, and never updated it, and therefore obtained inferior mapping. To explain this, Figure 1 (c) shows a state in mapping when F1, F2 and F3 have already been mapped. F1 is alone in the first region, F2 and F3 are together in the second region. Now is the time to map function F4. Size of F4 is 0.2 KB, therefore it can be mapped to either region, without violating the size constraint. The interference cost between region 1 and F4, i.e., between F1 and F4 is 2.2 KB. The interference cost between region 2 and F4 is traditionally computed as the sum of interferences between the functions in region 2 and F4, i.e., 1.7 KB between F2 and F4, and 0.6 between F3 and F4, totalling to 2.3 KB. Consequently traditional techniques will map F4 to region 1 with F1 (shown in Figure 1 (d)).

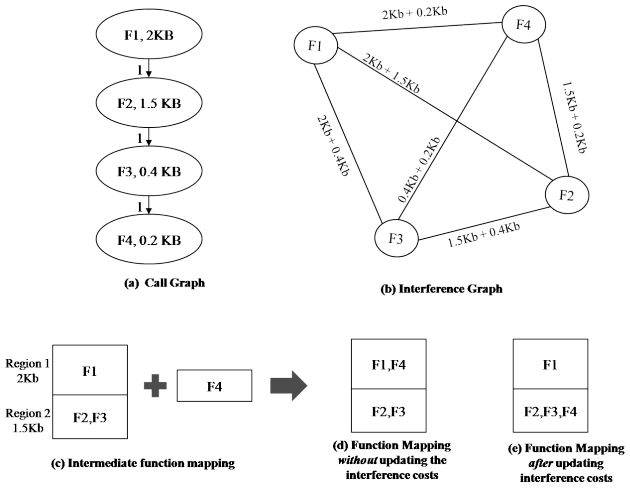

Figure 1. Interference cost between functions depends on where other functions are mapped, and updating the interference costs as we map the functions can lead to a better mapping.

Clearly there is a discrepency in computing the interference cost between region 2 and function F4. If F3 is also mapped to the same region, the interference cost between F2 and F4 should be estimated as 0. Otherwise, the interference costs between region 2 and function F4 are incorrectly (over)estimated. With this fixed, the interference between region 2 and F4 is just the interference between F3 and F4, which is just 0.6 KB. As per this correct interference calculation, F4 should be mapped to region 2 with functions F2 and F3 (shown in Figure 1 (d)). The required total data transfer between the main memory and the local storage, in this case 6KB, as compared to 7.6KB with the previous mapping, resulting in a 21% savings in data transfers.

III. Background and Problem Definition

For code management, both the number of regions and the mapping of functions to regions needs to be specified. Functions in a region replace each other, and therefore, the size of a region is the size of largest function in the region, and the total code space required is equal to the sum of the sizes of the regions. From the performance perspective, it is best to place each function into a separate region, so that it does not interfere with any other object, but that may increase the code segment of the local memory too much. On the other hand, mapping all the application functions into one region utilizes the minimum amount of code space, but incurs too much data transfer and therefore runtime overhead. The task of optimizing code management is therefore to organize the application functions into regions, i.e., how many regions, and function-to-region mapping that will obtain a balance between the code space used, and the data transfers required for code management.

To do this, it is essential to capture the number of times a function will swap each another as accurately as possible when they are mapped in the same region. This has been traditionally achieved through the use of Call Graph. Figure 2 (a) shows the outline of a program, in which function F1 calls functions F2 and F3. F2 calls F4 in a loop, and then calls F5 in a loop. F3 calls F6 and F7 in a common loop. The

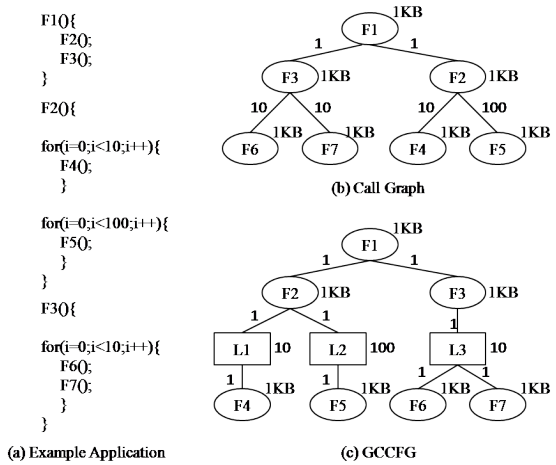

Figure 2. GCCFG captures both the aggregate and the temporal information about the execution of the application.

call graph for this example code is shown in 2 (b). Notice that while call graph captures that F1 calls F2 and F3, it does not capture the time-order in which they are called. Also, the fact that F6 and F7 are called one after another in a loop, and therefore will have a much higher interference than F4 and F5 cannot be interpreted from the call graph.

Incidentally this temporal and loop information is present in the control flow graph. Global Call Control Flow Graph or GCCFG [19] is a hybrid of control flow graph and a call graph, and capture this information, which is vital in estimating the interference between functions. As shown in Figure 2 (c) GCCFG is an ordered directed graph $G = (V_f, V_l, E, S, W)$, which contains two kinds of nodes, $v_f \in V_f$ represents function nodes, and $v_l \in V_l$ represents loop nodes. An edge $e_{i,j} \in E$ between function nodes represents a function call, If one of nodes is a loop node, the edge represents a control flow. If both are loop nodes, the edge represents a nested control flow. The edges are ordered, in the sense that if there are two or more child nodes of a parent node, then the left child is executed before the right child. Weights $w_i \in W$ are assigned to each node $v_i \in V_f \cup V_l$. Weight of the function node is the size of the function, while the weight on a loop node is the number of times the loop is executed. Weights on edges $w_{ij} \in W$ represent how many times the edge is taken. In summary GCCFG is an approximation of the control flow and call graph of the application, which intends to capture the function call sequence of the application as closely as possible. Our problem now can be stated as:

Inputs: GCCFG (V_f, V_l, E, S, W) of an application, and the maximum code size on SPM, $\mathbb{S}$

Outputs: The number of regions n_R, and mapping of functions to regions, $M : V_f \rightarrow (1, n_R)$

Constraints:

- For each region $r \in (1, n_R)$, size of region $S_r = MAX\ [(M(v_f) == r) \times s_f]$
- $\sum_{r=1}^{n_R} s_r \leq \mathbb{S}$

Objective:

- Minimize data transfer. $min\ \sum_{v_i, v_j \in V_f} I^M(v_i, v_j) \times (M(v_i) == M(v_j))$, where $I^M(v_i, v_j)$ is the interference of V_i and v_j under the mapping M.

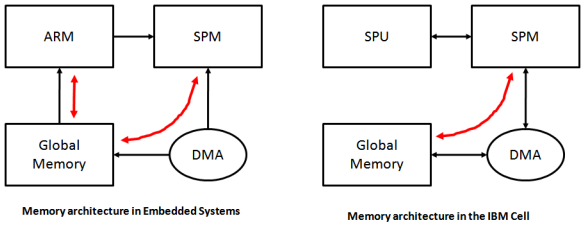

Figure 3. In embedded systems, typically using the SPM was a choice, and therefore the code mapping problem was just an optimization problem. In the cell architecture, all code has to reside in the SPM or local store.

IV. Related Work

Limited local memory architectures have been popular in embedded systems for some time now under the name of Scratch Pad Memories or SPMs [4]. They have been extensively used in embedded systems, e.g., the ARM architecture [2] for power reduction, and techniques have been developed to manage code [1], [5], [6], [10], [18], [19], [21]–[23], [25]–[27], global variables [3], [11], [12], [14], [22], [23], [25], [26] stack [3], [8], [14], [18], [23], and heap data [8] on SPMs. In this paper, we concentrate only on code mapping on SPMs.

Figure 3 illustrates the major difference between the two use-cases of scratch pad memories. Figure 3 shows that in the embedded systems e.g. the ARM architecture, the SPM was present in addition to the regular cache hierarchy of the processor. Programs could execute correctly without the use of SPM; they could however, use SPM to improve power and performance. Consequently, a lot of SPM research has focused on the question of "what to map" on the SPM [1], [3], [8], [18], [22], [26], [27]. Static mapping techniques essentially selects a portion of code, loaded it onto the SPM at the initialization stage, and it does not change during the execution of the program. Dynamic code mapping resolves this inefficiency by changing the code mapping onto the SPM at runtime [5], [6], [10]–[12], [14], [19], [21], [23], [25]. Bringing new code may require replacing some existing code, and therefore, "where to map" question is also important for dynamic techniques.

The "what to map" question is not valid for limited local memories present in present high-performance multi-core processors. For example, in the Synergistic Processing Unit (SPU) of IBM Cell, all code/data must go through the SPM, or the local memory. "Where to map" is the only question for code mapping on SPUs. To facilitate code management, the local store can partitioned into regions, and functions can be mapped onto regions. Only the functions mapped to the same region will replace each other as they are called. Consequently, the size of region is the size of the largest function in the region, and the total code space needed is the sum of the functions. In the SPUs, the local store is shared by all code and data, therefore it is important to conserve space occupied by code. In addition there is a clear trade-off between the code space needed and the potential performance. Consequently, there are two questions to be addressed in code mapping i) "how many regions", and ii) "mapping of functions to regions."

[23] proposes a method to allocate code, stack and global variables and apply a best-fit heuristic to map those into a local storage at the specific locations. However, they formulate their problem from call graph, which does not capture temporal information. [25] formulates an ILP for the problem of dynamic code mapping to determine memory object to be mapped in the local storage. Since ILP solution is impractical even for small-sized applications, they propose the first-fit heuristic to assign memory addresses. Even though [25] formulate their problem using GCCFG, both [23] and [25] assume that the "how many regions" has already been solved. Note that this problem in itself is NP-complete, and consequently, there is a phase ordering problem between determining the number of regions and finding the mapping. The closest to our work is [19], in which authors formulate the code mapping problem from GCCFG, and solve both the problems of "how many regions" and "mapping of functions to regions". Because of simultaneously solving both these problems they achieve better mapping than previous techniques. However they assume conservative interference costs. They calculate these costs between every pair of functions at the beginning, and do not update them as their heuristics maps functions to regions. In this paper, we alleviate this problem, and develop two heuristics and a combination of them to achieve much better mapping.

V. Our Approach

When two functions are mapped in the same region, they replace each other when called. The data transfer cost of mapping two functions in the same region is termed as the interference cost between the two functions. The objective of code mapping is to cluster functions into a number of regions such that the sum of interference between functions in a region, over all regions is minimized. This is equivalent to the max k-cut problem [13], which is NP-hard even for $k = 2$. Note however, that the min-cut problem is solvable in polynomial time for a given k. The max-cut problem however is APX-hard [20], which implies that there is not even a good way to approximate it in polynomial time.

To partition functions into k clusters, we take two approaches. First is the FMUM (or Function Mapping by Update and **Merging**). In this we first map all the functions to different regions. Then we merge two functions that will cause minimal increase in the runtime, until the regions fit in the code space $\mathbb{S}$. The second scheme FMUP (or Function Mapping by Update and **Partitioning**) is just the reverse of this, in which we start will all the functions in one region. In each step we create a new partition that maximizes performance improvement. We continue this until we do not overstep the code space constraint $\mathbb{S}$. In order to correctly compute the interference costs, we note that it is not possible to find the interference cost without the mapping. Therefore, note that in the algorithms we always find the interference cost for a given mapping. The next two subsection describe the details of the two heuristics.

A. Function Mapping by Updating and Merging (FMUM) Heuristic

Algorithm 1 outlines a simplified version of the FMUM heuristic. It starts with M, in which all functions are mapped in separate regions (lines 01-02). Now we try to merge all combinations of two regions until we are within space constraints (while loop, lines 04 - 29). To do this, we choose a region pair (r_1, r_2) (line 08), and create a new mapping M' by moving all the functions from region 2 to region 1 (for loop, lines 09-15). We compute the cost of this mapping M' (line 17), and thereby find out the mapping with the least interference cost (lines 18-21). Merging only the regions that do not result violation of code size constraint is allowed (line 16). If the mapping with the minimum interference cost M'' is better than the original mapping M, then the mapping is changed to the one with minimum interference cost (lines 24-25). Otherwise, the algorithm terminates (line 27).

The outermost while loop merges two regions at a time. Since in the worst case, all regions may have to be merged into one, this loop can execute V_f times. Inside this, the for loop (lines 08-23) runs for each pair of regions. This adds $O(V_f^2)$ complexity to the time. Inside this loop, there is a for loop (lines 09-15), which generate a mapping, and take $O(V_f)$ time, and there is a Interference cost calculation, which, assuming for now, has complexity I. Thus the worst case timing complexity of FMUM is $O(V_f^4 + V_f^3.I)$.

B. Function Mapping by Updating and Partitioning (FMUP) Heuristic

Algorithm 2 outlines a simplified version of the FMUP heuristic. It starts with M, in which all functions are mapped into the first region (lines 01-02). In each iteration of the outer while-loop (lines 05-31), we create a new region $R++$, and then move functions to fill up the new region. The inner while-loop (lines 07-30) picks one function at a time $v_i \in V_f$ (for loop, line 11), creates a new mapping M' in which this function is in the new region (lines 12-18). It then computes the difference in interference costs between the initial mapping M and new mapping M' (line 20); the difference implies how much it is beneficial to move one function into the new region. The function, whose moving to the new region does not violate code size constraints (line 19), and which has the maximum difference in cost of moving (lines 19-21), is finally picked, and moved (line 21). When no more function can be moved, the inner while loop ends. The outer while loop will continue creating new regions until the code size constraint stops it, and stops the algorithm.

The inner most for loop (lines 12-18) generates a mapping and takes $O(V_f)$ times. The for loop in lines 11 - 24 is over all functions, and therefore is repeated $O(V_f)$ times. Each time interference cost calculation is performed. Assume for now, that the interference cost calculation has complexity I. The inner while-loop in lines 07-30 tries to move all the functions, one at a time. This adds a factor of $O(V_f)$ to the complexity. The outer while loop creates new regions. In the worst case, it will create V_f regions. Therefore the overall complexity of this algorithm is $O(V_f^4 + V_f^3.I)$.

Algorithm 1 FMUM : Updating and Merging $(V_f, \mathbb{S})$

for all functions $v_i \in V_f$ **do**
  $M[v_i] = r_i$
end for
while $(Size(M) < \mathbb{S})$ **do**
  $init\_cost = Interference(M)$
  $min\_cost = MAX\_COST$
  $M'' = \phi$
  for all combination of regions $r_1, r_2 \in R$ **do**
    for all functions $v_i \in V_f$ **do**
      if $(v_i \notin r_2)$ **then**
        $M'[v_i] = M[v_i]$
      else
        $M'[v_i] = r_1$
      end if
    end for
    if $(size(M') < \mathbb{S})$ **then**
      $cost = Interference(M')$
      if $(cost < min\_cost)$ **then**
        $min\_cost = cost$
        $M'' = M'$
      end if
    end if
  end for
  if $(min\_cost < init\_cost)$ **then**
    $M = M''$
  else
    Return
  end if
end while

Algorithm 2 FMUP : Updating and Partitioning $(V_f, \mathbb{S})$

for all functions $v_i \in V_f$ **do**
  $M[v_i] = r_1$
  $R = 1$
end for
while $(Size(M) < \mathbb{S})$ **do**
  $R++$
  while **do**
    $init\_cost = Interference(M)$
    $max\_cost = MIN\_COST$
    $M'' = \phi$
    for all functions $v_i \in V_f$ **do**
      for all functions $v_j \in V_f$ **do**
        if $(v_i \neq v_j)$ **then**
          $M'[v_i] = M[v_i]$
        else
          $M'[v_i] = r_R$
        end if
      end for
      if $size(M') < \mathbb{S}$ **then**
        $cost = init\_cost - Interference(M')$
        if $(cost > max\_cost)$ **then**
          $max\_cost = cost$
          $M'' = M'$
        end if
      end if
    end for
    if $(cost > 0)$ **then**
      $M = M''$
    else
      Break from while loop
    end if
  end while
end while

C. Interference Cost Calculation

Since the interference cost calculation is very frequently required by both FMUM and FMUP, making interference cost calculation as fast as possible is important. Given a GCCFG, and a mapping M, a naive way to compute interference cost can be by traversing the GCCFG, (much like simulation) and adding the the function sizes, as we visit function nodes. However, the complexity of this will be very high. Therefore, we develop an algorithm to compute the interference cost using just 2 Depth First Search (DFS) traversals of the GCCFG. While we are unable to explain it in detail here due to space considerations, the main idea hinges on the fact that the function switches depend on the frequency of the edges of the least common ancestor of the two functions. The runtime complexity of interference cost calculation is O(V_f).

VI. Empirical Evidence

A. Experimental Setup

We evaluate the effectiveness of all our techniques on the IBM Cell processor in Sony Playstation 3 running Linux Fedora 7. Our software development environment is on a laptop with 2GHz Intel Celeron processor with 1 GB memory. We have set up a GNU cross compiler ver. 4.1 and PS3 SDK ver. 3.1, which supports using linker script for code management. We compile several benchmarks from MiBench [9] suite on the desktop, transfer them to PS3, and execute them there to measure the runtime. We tried to execute all the MiBench applications on the SPUs, but only the ones in our set executed; the data and code size of the other applications were too large to fit on the local storage of the SPU. This further underlines the need of data and code memory management schemes, such as the ones presented in this paper.

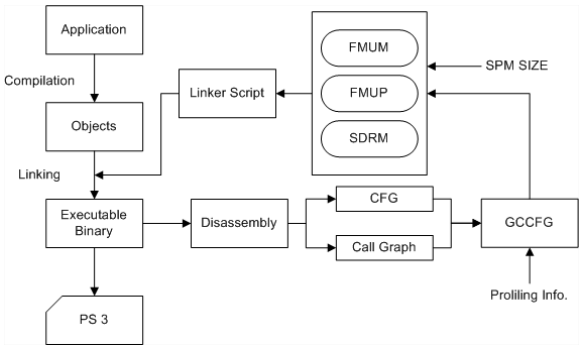

Figure 4. We perform our experiments on the IBM Cell processor in Sony Playstation 3. Our software framework is developed as a post-pass to the GCC compiler.

We have implemented our code mapping heuristics as a post-pass to the GCC compiler. We first compile the benchmark to executables and capture their *objdump*. We analyze the objdump to re-create the call graph and the control flow graph. From these we create the GCCFG of the application. At this point, we execute the application, and gather more accurate profile information about the edge costs, and integrate them manually into the GCCFG to best represent the application execution patterns. Performing this post-pass analysis at the objdump, instead of inside the compiler, or on assembly, allows us to analyze library functions too, and consequently, our GCCFG contains library functions too. Another important clarification here is that in the linker script, object to region mappings are specified, rather than function to region mappings. For the source code, each function can be compiled into a separate object (e.g., by placing them in a separate file, and compiling them), but library objects that contain multiple functions cannot be easily broken into individual objects. Therefore, all our techniques are at function-level granularity for source code, but at object level granularity level for the library functions. However, this in no way affects the generality of problem definition and effectiveness of heuristics.

Table I
BENCHMARKS FOR EXPERIMENTS

Benchmarks	Size(Bytes)	Description
Stringsearch	800 - 3792	Word Searching (Office)
Dijkstra	2496 - 8080	Shortest Path (network)
Basicmath	3880 - 11360	Math Calculation (Automotive)
FFT (encoding)	2496 - 11520	Signal Processing (telecom)
Susan (smoothing)	8300 - 19144	Image smoothing (Automotive)
Susan (edge)	8300 - 19144	Edge Detection (Automotive)
Adpcm (encoding)	1496 - 4648	Audio Compression (telecom)
Adpcm (decoding)	1496 - 4648	Audio Compression (telecom)

Once we have all the functions and objects (for source code, and library respectively), we can compute the minimum and maximum size of code region required. Table I shows the size limits for all our benchmarks from MiBench. Note that the maximum code size of the benchmarks are much smaller than the size of the local store on the SPUs, which is 256 KB. However, remember that the local store is shared between all the data (global, stack, heap), and the code. Therefore using minimum amount of space for code is extremely crucial.

After generating the GCCFGs for a benchmark, and given code size constraint we used our heuristics of FMUM, FMUP, and previous technique SDRM [19], to our knowledge, SDRM technique is the latest, and it leads to the near-optimal solution for the code mapping problem, to generate code mapping in the linker script. This linker script is then compiled with the object files again to create the binary. The binary is executed 10 times on PS3, and the average runtime value is used so as to drown the variations due to OS etc. We did these experiments for all our benchmarks, and for all code size constraints from the minimum to the maximum, in increments of 100 bytes.

B. Experimental Results

Figure 5 shows the runtime of the binary compiled using the code mapping obtained from each heuristic for a representative application, *stringsearch*. The X-axis shows the size of the given code size constraint. In the graph, the diamond markers represents the total number of execution cycles with Function-To-Region Mapping by FMUM heuristic, square markers represents that by FMUP heuristic, and triangle markers represents that by SDRM heuristic. When the code size constraint is very tight, all heuristics achieve the same mapping, i.e., mapping all the functions in one region. However, as we start relaxing the code size constraint, FMUP typically gives better mapping than FMUM. This is because FMUP has to do very few steps, while FMUM needs to do a lot of merges. The more steps a heuristic has to take, the errors in each step accrue, and we get a worse mapping. The reverse effect is also visible. When the code size constraint is extremely relaxed, e.g., greater than 2000 bytes, FMUM gives a much better mapping. Note that code mappings created by the FMUP do not always lead to a better mapping with the relaxed code sizes compared to that with tight sizes. This is because the FMUP moves one function at a time if the given code size accommodates the function moving, which may not be the best decision for functions that are mapped later, therefore, it ends up with a totally new code mapping with local maximum code mapping. None of the heuristics gives consistently good results, however our techniques give better results most of the times.

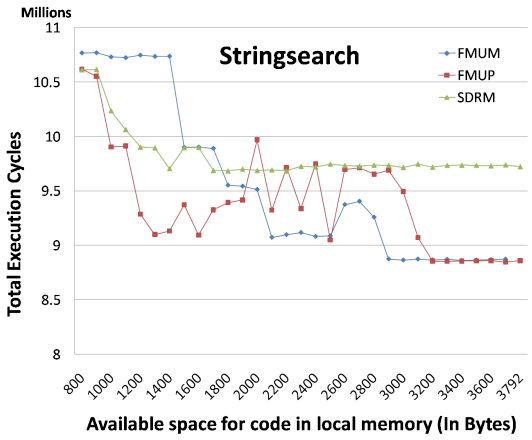

Figure 5. FMUP gives better results for tighter code size constraints, while FMUM gives better results for relaxed code sizes.

We tested the three heuristics for all code size constraints, from minimum to maximum in increments of 100 bytes. Figure 6 plots the number of times our techniques obtain a better result than SDRM. For some benchmarks like *dijkstra*, FMUM in general, gives worse results than SDRM, however, on average over all benchmarks, FMUM gives a better result than SDRM 57% of time, but FMUP gives a better result 80% time. In a sense FMUM and FMUP are complimentary heuristics, because FMUP is likely to give better result for tight code size constraints, and FMUM for relaxed code size constraints. Therefore we define a third composite heuristic, FMUP+FMUM, which essentially mean: try both of them, and pick the better one. FMUM+FMUP gives us better results than SDRM about 87% of the time. For some applications, like *fft* and *adpcm* FMUP+FMUP gives better results 100% of the time.

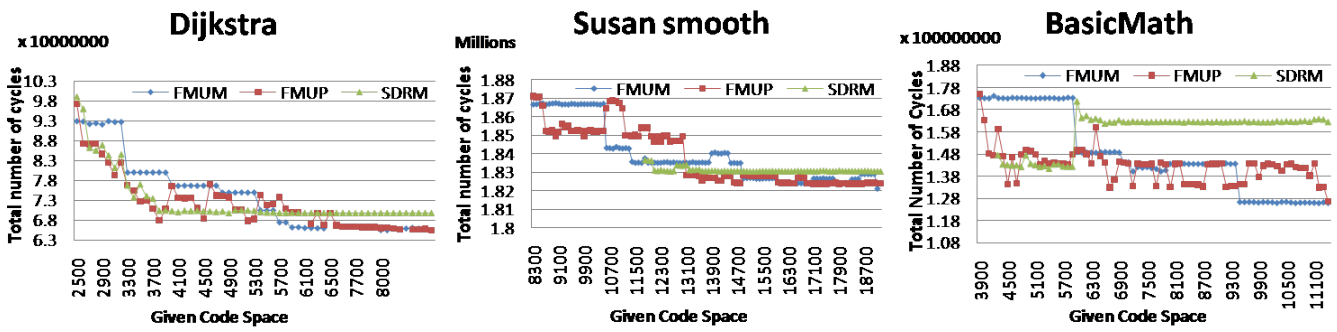

Figure 8. Limitations of SDRM; Less utilization of given code space, Inefficient object mapping for in-loop functions, Less map-ability

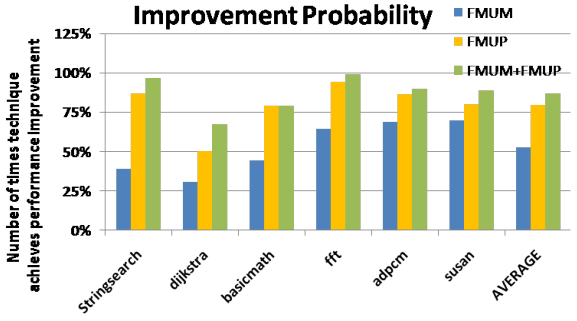

Figure 6. As an average, FMUM gives a better results than SDRM 57% of time, FMUP gives a better results 80%of time, and FMUM+FMUP gives better results 87% of the time.

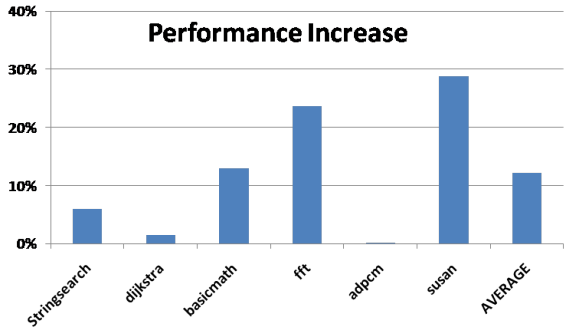

Figure 7. On an average the mapping generated by FMUM+FMUP executes 12% faster than that generated by SDRM

Figure 7 plots the average performance improvement (or runtime reduction) we achieved for each benchmark, over all the SPM sizes. Depending on SPM size and the application, the performance improvements can be as high as 50%. At the application level, the average improvement for any SPM size can be expected to be as high as 29%. Finally, for any given application and any given SPM size, the performance improvement compared to SDRM can be expected to be around 12%. This performance increase is definitely not for free. As compared to SDRM, we spend a lot more time during the compilation, trying to search for a better mapping. Even with the increase however, the compile-time for our set of benchmarks is always less than a minute. This is definitely tolerable for several applications which are compiled once, distributed as binaries, and executed several times: this profile closely fits the kinds of applications that are intended for PS3, e.g. games, multimedia and office applications.

C. Analysis

To get more insight into the behavior of FMUM, FMUP, and SDRM, we look at detailed results for a couple of benchmarks. The following are some of the limitations of SDRM, that FMUM and FMUP overcome.

1) Utilization of given code space: All the plots in Figure 8 show that towards the tail-end, when the code size constraint is relaxed, SDRM achieves worse mapping than FMUM and FMUP. Even though SDRM initially makes good use of local store, but once it gets a decent mapping, it does not relax it for small benefits.

2) Inefficient object mapping for in-loop functions: For the *basicmath* results in Figure 8, the total execution time by function mapping from the SDRM heuristic increases after code size constraint of 5700 Bytes. This is because some functions that are called in a loop are mapped together in the same region due to conservative estimates of interference costs. FMUM and FMUP correctly compute the interference costs, and map the functions separately, drastically reducing the data transfers and runtime.

3) Less map-ability: For susan smoothing, if code size constraint of less than 11800 Bytes is given, SDRM is unable to generate a mapping. Recall from Table I that the minimum code size required for susan smoothing was 8300 Bytes. SDRM maps one function at a time with the highest interference cost. As the mapping process goes on, if large sized objects are left, they cannot be mapped in existing regions or a newly created region since several regions have already been created for smaller regions. FMUM and FMUP heuristics avoid this problem. Similar effect can be observed for the basicmath benchmark when code size constraint is less than 4300 Bytes.

VII. Conclusion

Multi-core architectures with limited local memories are becoming popular. In these processors, e.g, IBM Cell, each core has access to only a limited amount of local memory, and access to any other memory has to be explicitly specified in the application. To manage code in the limited code space on the local memory, different functions share the same address in the local store. The problem of code mapping is therefore to place functions into regions, so that the replacement is minimized. Previous works have formulated this problem using call graphs, which do not capture the temporal information about the execution, and also do not

update the estimate of interference cost between two functions as the other functions are mapped. In this paper, we formulate the code mapping problem using GCCFG, which captures the temporal information of the application execution, and also update the interference cost as we generate mapping. As a result, our heuristic can create more efficient Function-To-Region mapping with the given code size in local storage.

Our experiments on solving the code mapping problem for several benchmarks from MiBench for the IBM Cell processor demonstrate a speedup of about 12% as compared to previous approaches at tolerable compile-time overhead. While the experiments in this paper are on a single SPU and PPU, the performance improvement due to our scheme is likely to scale with the number of cores since reducing the memory transfers becomes ever more important in a multicore setting. An area of improvement is prefetching code objects. Right now, we fetch the function code at the last moment. However, based on profile or analysis, the function code can be fetched beforehand, overlapping computation and communication, and further improving performance. We have automated our technique in the GCC cross compiler for SPUs. We have made a software release including all benchmarks, and it can be downloaded from http://www.public.asu.edu/~sjung/thesis/software_release_Seungchul Jung.zip

ACKNOWLEDGMENT

We would like to thank all who supported our work, and this research was partially funded by grants from National Science Foundation CCF-0916652, IIP-0856090, NSF I/UCRC for Embedded Systems, Microsoft Research, Raytheon, SFAz and Stardust Foundation.

REFERENCES

[1] F. Angiolini, F. Menichelli, A. Ferrero, L. Benini, and M. Olivieri. A post-compiler approach to scratchpad mapping of code. In *CASES '04: Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 259–267, New York, NY, USA, 2004. ACM.

[2] ARM. "ARM Architecture Reference Manual".

[3] O. Avissar, R. Barua, and D. Stewart. An optimal memory allocation scheme for scratch-pad-based embedded systems. *ACM Trans. Embed. Comput. Syst.*, 1(1):6–26, 2002.

[4] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In *CODES '02: Proceedings of the tenth international symposium on Hardware/software codesign*, pages 73–78, New York, NY, USA, 2002. ACM.

[5] B. Egger, C. Kim, C. Jang, Y. Nam, J. Lee, and S. L. Min. A dynamic code placement technique for scratchpad memory using postpass optimization. In *CASES '06: Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 223–233, New York, NY, USA, 2006. ACM.

[6] B. Egger, J. Lee, and H. Shin. Scratchpad memory management for portable systems with a memory management unit. In *EMSOFT '06: Proceedings of the 6th ACM & IEEE International conference on Embedded software*, pages 321–330, New York, NY, USA, 2006. ACM.

[7] A. E. Eichenberger, J. K. O'Brien, K. M. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, M. K. Gschwind, R. Archambault, Y. Gao, and R. Koo. Using advanced compiler technology to exploit the performance of the cell broadband enginetm architecture. *IBM Syst. J.*, 45(1):59–84, 2006.

[8] P. Francesco, P. Marchal, D. Atienza, L. Benini, F. Catthoor, and J. M. Mendias. An integrated hardware/software approach for run-time scratchpad management. In *DAC '04: Proceedings of the 41st annual Design Automation Conference*, pages 238–243, New York, NY, USA, 2004. ACM.

[9] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *WWC '01: Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society.

[10] A. Janapsatya, A. Ignjatović, and S. Parameswaran. A novel instruction scratchpad memory optimization method based on concomitance metric. In *ASP-DAC '06: Proceedings of the 2006 Asia and South Pacific Design Automation Conference*, pages 612–617, Piscataway, NJ, USA, 2006. IEEE Press.

[11] M. Kandemir, J. Ramanujam, and A. Choudhary. Exploiting shared scratch pad memory space in embedded multiprocessor systems. In *DAC '02: Proceedings of the 39th annual Design Automation Conference*, pages 219–224, New York, NY, USA, 2002. ACM.

[12] M. Kandemir, J. Ramanujam, J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh. Dynamic management of scratch-pad memory space. In *DAC '01: Proceedings of the 38th annual Design Automation Conference*, pages 690–695, New York, NY, USA, 2001. ACM.

[13] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[14] L. Li, L. Gao, and J. Xue. Memory coloring: a compiler approach for scratchpad memory management. pages 329 – 338, sept. 2005.

[15] P. Michaud, A. Seznec, D. Fetis, Y. Sazeides, and T. Constantinou. A study of thread migration in temperature-constrained multicores. *ACM Trans. Archit. Code Optim.*, 4(2):9, 2007.

[16] J. Montanaro, R. T. Witek, K. Anne, A. J. Black, E. M. Cooper, D. W. Dobberpuhl, P. M. Donahue, J. Eno, G. W. Hoeppner, D. Kruckemyer, T. H. Lee, P. C. M. Lin, L. Madden, D. Murray, M. H. Pearce, S. Santhanam, K. J. Snyder, R. Stephany, and S. C. Thierauf. A 160-mhz, 32-b, 0.5-w cmos risc microprocessor. *Digital Tech. J.*, 9(1):49–62, 1997.

[17] T. J. Network, M. Adiletta, M. Rosenbluth, D. Bernstein, G. Wolrich, and H. Wilkinson. The next generation of intel ixp network processors. *Intel Technology Journal*, 6, 2002.

[18] N. Nguyen, A. Dominguez, and R. Barua. Memory allocation for embedded systems with a compile-time-unknown scratchpad size. In *CASES '05: Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems*, pages 115–125, New York, NY, USA, 2005. ACM.

[19] A. Pabalkar, A. Shrivastava, A. Kannan, and J. Lee. Sdrm: Simultaneous determination of regions and function-to-region mapping for scratchpad memories. In *Int'l Conference on High Performance Computing (HIPC)*, pages 569–582, 2008.

[20] C. Papadimitriou and M. Yannakakis. Optimization, approximation, and complexity classes. In *STOC '88: Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 229–234, New York, NY, USA, 1988. ACM.

[21] S. Steinke, N. Grunwald, L. Wehmeyer, R. Banakar, M. Balakrishnan, and P. Marwedel. Reducing energy consumption by dynamic copying of instructions onto onchip memory. In *ISSS '02: Proceedings of the 15th international symposium on System Synthesis*, pages 213–218, New York, NY, USA, 2002. ACM.

[22] S. Steinke, L. Wehmeyer, B. Lee, and P. Marwedel. Assigning program and data objects to scratchpad for energy reduction. In *DATE '02: Proceedings of the conference on Design, automation and test in Europe*, page 409, Washington, DC, USA, 2002. IEEE Computer Society.

[23] S. Udayakumaran, A. Dominguez, and R. Barua. Dynamic allocation for scratch-pad memory using compile-time decisions. *ACM Trans. Embed. Comput. Syst.*, 5(2):472–511, 2006.

[24] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, A. Singh, T. Jacob, S. Jain, V. Erraguntla, C. Roberts, Y. Hoskote, N. Borkar, and S. Borkar. An 80-tile sub-100-w teraflops processor in 65-nm cmos. *Solid-State Circuits, IEEE Journal of*, 43(1):29 –41, jan. 2008.

[25] M. Verma and P. Marwedel. Overlay techniques for scratchpad memories in low power embedded processors. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 14(8):802 – 815, aug. 2006.

[26] M. Verma, K. Petzold, L. Wehmeyer, H. Falk, and P. Marwedel. Scratchpad sharing strategies for multiprocess embedded systems: a first approach. pages 115 – 120, sept. 2005.

[27] M. Verma, L. Wehmeyer, and P. Marwedel. Cache-aware scratchpad allocation algorithm. volume 2, pages 1264 – 1269 Vol.2, feb. 2004.