A Performance Model and Code Overlay Generator for Scratchpad Enhanced Embedded Processors

Michael A. Baker†‡, Amrit Panda†, Nikhil Ghadge†, Aniruddha Kadne†, Karam S. Chatha†

†Computer Science and Engineering Department
Arizona State University
Tempe, Arizona 85281

‡Sensors and Electron Devices Directorate
U.S. Army Research Laboratory
Adelphi, Maryland 20783

{mike.baker, amrit.panda, nghadge, aniruddha.kadne, karam.chatha}@asu.edu

ABSTRACT
Software managed scratchpad memories (SPMs) provide improved performance and power in embedded processors by reducing required hardware resources. Performance depends strongly on the scheme used to map code and data onto the SPM, but generating optimal mappings can be extremely difficult. Here we address instruction mapping on SPMs and present a performance model and algorithm, “Code Overlay Generator” (COG), for producing high performance dynamic SPM code mappings. Our heuristic does not require profiling information, and is suitable for generating mapping solutions for large programs which are otherwise infeasible using previously proposed Integer Linear Programming (ILP) techniques.

We compare our algorithm with a published heuristic and the code overlay mapping algorithm provided with the Cell Broadband Engine (CBE) Synergistic Processing Unit (SPU) compiler from IBM, spu-gcc. We find an average performance advantage of 34% compared to the previous algorithm, and 87% with respect to spu-gcc. We additionally show that our performance model enables improved tools for offline evaluation of code overlay performance and mapping selection.

Categories and Subject Descriptors
B.1.4 [Control Structures and Microprogramming]: Microprogram Design Aids—Optimization

General Terms
Performance, Algorithms, Design, Experimentation

Keywords

1. INTRODUCTION
Performance and power consumption are critical design considerations in embedded systems. System memories may consume half of the limited power budget [7], which has motivated a tremendous amount of work aimed at improving memory performance and efficiency. Scratchpad Memories have grown dramatically in importance in recent years, particularly as performance requirements for embedded systems become more like those of power hungry general purpose devices.

Scratchpad Memories (SPMs) are software managed data and instruction storage structures integrated into a processor or System on Chip (SoC) architecture in much the same way as system cache. The SPM enables performance and silicon area advantages over the system cache by shedding hardware required for cache management and relying on compile time and programmer directives to adequately manage its use. How best to utilize the SPM is a difficult problem which has been heavily explored over the past decade.

There are a number of approaches for selecting what to place into the SPM and when to place it there. In this paper we consider the problem in which a code partition is defined on the SPM, and a large code set must be dynamically mapped into the available space by assigning multiple segments of code to the same address in memory. An example of this situation is the IBM Cell Broadband Engine (CBE) which provides eight Synergistic Processing Units (SPUs) each with a software managed 256KB Local Store SPM. In the CBE model, all code to be executed in an SPU must be mapped to the local SPM [9]. This model promises to become increasingly important as multicore SoCs make their way into multitasking handheld devices. Here each processing core may be assigned a compute intensive task such as video coding in a real-time resource constrained environment.

We introduce a Code Overlay Generator (COG) Algorithm for identifying high performance overlay mappings designed to minimize overhead. We present two extensions to the COG algorithm and compare the performance of our implementations against a previously published algorithm and the automatic overlay mapping generator in the IBM Cell SPU compiler, spu-gcc.

In the next section we will discuss previous and related work. We give details on code overlays in Section 3 and introduce our overlay performance and cost models in Section 4. The COG algorithm and extensions are presented in Section 5 followed by an analysis of expected performance compared to the previous algorithms in Sections 6 and 7. Our experimental setup and results are presented in Section 8, followed by Conclusions.

2. PREVIOUS WORK
Existing work may be categorized into either instruction mapping, data mapping schemes, or both, as well as either static or dynamic allocation techniques. We have developed a code overlay mapping generator for dynamically mapping instructions to memory at runtime. Our implementation does not require profiling information at compile time. We also present an improved scheme for modeling relationships between code objects at runtime to be-
ter predict instruction memory misses based on structural analysis of the the program.

Several previous efforts focus on temporal locality of code. Early work in this area investigate the concurrency of code modules based on branches of the execution call graph [4] [11], but do not consider the effects of control structures such as loops. Similar to our effort, Steinke et al. [12] view the problem in terms of minimizing memory accesses, and evaluate the structure of the program in terms of basic blocks and functions to formulate an ILP problem. Udayakumar et al. [13] present an algorithm which looks at timestamps in code sections to determine temporal locality, Janapatsaya et al. [7] introduce a concomitance metric which relies on profiling trace data, and Angiolinie et al. [3] present a dynamic programming algorithm which also requires trace data. Egger et al. [5] implement a paged SPM management and prefetching scheme. These schemes rely on profiling information which can be impractical or inflexible particularly where program execution paths vary widely on different input data.

Verma et al. [14] and Pabalkar et al. [8] are most similar to our work. Verma uses a first-fit heuristic to assign objects to the SPM, but their scheme results in a static mapping which omits objects that do not fit by placing them back in main memory. Pabalkar’s algorithm faces a related problem when attempting to assign all code modules to the SPM due to potential deadlock which is not handled. More details on SDRM deadlock are provided in Section 6.2. Additionally, the existing overlay mapping generation algorithm included in the spu-gcc compiler provided by IBM for the CBE addresses our problem [1][2][10], but the algorithm gives limited performance. Our results are compared with the SDRAM and spu-gcc solutions in this paper.

3. CODE OVERLAY

Code Overlay is a technique for mapping instruction code onto available memory in which the code would not otherwise fit. In designing an overlay mapping, available memory is partitioned into one or more fixed regions to which one or more code segments may be assigned. Every segment assigned to a given region is mapped to the same address in the SPM. The size of a region in memory is precisely that of its largest mapped segment. Code may be stacked into segments so that the start address of the second code element follows the end address of the first element and so on.

Normally instructions may be assigned to segments at object file, function, or basic block resolution, and any number of elements may be assigned to one segment. When the code overlay scheme is implemented and code is executed, each memory region always contains exactly one code segment at any given time during execution. When instructions are requested which are not currently present in the SPM, the appropriate segment is loaded into its assigned region overwriting the current segment. We refer to this event as an overlay miss. Each miss has an associated overhead which is defined as the amount of time expended to load the missing segment. This overhead is proportional to the size of the code segment. The cumulative overhead in terms of time associated with execution of a program using a specific overlay mapping is the mapping’s total cost.

4. OVERLAY MISS MODEL

By analyzing source code, we can identify key characteristics which will be important for producing an accurate model of program behavior to effectively reduce overlay misses during execution. In this work, we create overlays using function or object file resolution, where an object file may contain one or more functions. We must identify any functions in the code which might be assigned to the same memory address, or overlayed, and determine the relationships between them which we expect to result in overlay misses. Currently we are limited to this level of resolution by the GNU linker, ld, which takes a script describing the overlay mapping where code segments must be defined using object files.

4.1 Graphical Representation of Code

In order to create an abstract graph representation of the input program, we need to address four important control flow relationships between functions as shown in Table 1. We build this information into an enhanced Control Flow Graph (CFG) referred to as a Global Call Control Flow Graph (GCCFG) as presented by Pabalkar et al. [8]. The GCCFG is similar to the CFG, but with control flow relationships explicitly represented as nodes. Figure 1 illustrates an example GCCFG which might be generated by identifying jumps or function calls, loops, and conditional statements in the source code. In our case, analysis is performed on cfg output generated by gcc using the -fdump-tree-cfg switch [6]. We parse the cfg dump to control collection data at the granularity of basic blocks from which loops and function calls information is collected. We are currently limited to function level resolution by the existing overlay manager available for the CBE which we use in our experiments. We simplify the function level graph generated from the cfg data in four steps:

1. First we ignore recursive function calls as indicated in the graph at function b. Since we are concerned with interference between functions, the effect of a recursive call is that the code necessary to run the called function is already in memory, resulting in no overlay miss.

2. In the second step, we will treat functions inside conditional statements including function pointers which may represent multiple functions as if they interfere with one another in much the same way as if they were called outside of any conditional control structure. The impact of this transformation on the model’s accuracy is dependent on a large number of variables including the detailed behavior of the code on a given input. We observe that two conditionally executed functions called by the same parent can have an interfer-
ference values. The base interference, $I_b$, first. When we visit a child, program execution, every function will have an overlay overhead the assumption that no code is present in the SPM at the start of Starting from the root node we assign number of times the function must be in memory due to a call from this way, a base interference is assigned to each function node with descend through a loop node, the current value of by a loop factor. A factor of 10 is used in Figure 2. When returning in the graph.

Now we have produced the simplified GCCFG structure in Figure 2. In the following sections we begin characterizing interference relationships and costs.

### 4.2 Counting Overlay Misses

For each function, we will identify base, return, and total interference values. The base interference, $I_b$, represents the number of times we expect a function to be called during program execution. Starting from the root node we assign $I_b$ a value of one. Based on the assumption that no code is present in the SPM at the start of program execution, every function will have an overlay overhead of at least one miss on its first call.

Base interferences are assigned by traversing the graph depth first. When we visit a child, $I_b$ remains the same as the parent node unless we pass through a loop between them. Each time we descend through a loop node, the current value of $I_b$ is multiplied by a loop factor. A factor of 10 is used in Figure 2. When returning upward through a loop node, $I_b$ is divided by the loop factor. In this way, a base interference is assigned to each function node with geometrically increasing values as we descend into loops.

**Definition 1. Base interference.** $I_b(f)$ is the number of times a function is expected to be called during program execution, i.e. the number of times the function must be in memory due to a call from another function.

The return interference, $I_r$, for each function represents the number of times the function must be present in memory due to function call returns. If we consider the call graph in Figure 3, the order in which the functions must be present in memory is:

< a, b, a, c, a, ..., a, n, a >

Notice that the parent function, $a$, must be present in memory for all $n-1$ function call returns, where $n$ is the number of functions in the graph.

### 4.3 Interference Between Functions

In order to establish the quality of an overlay mapping, we must define the interference relationship between code segments. To simplify the analysis, we will restrict the contents of a code segment to a single function, and equate segment interference with function interference.
Definition 4. Interference between functions, \( \text{inter}(u, v) \) is the number of times two functions, \( u \) and \( v \) are expected to replace each other in memory if both are mapped to the same region.

We consider two types of interference relationships defined by the lowest common ancestor (LCA), between two functions, \( u \) and \( v \). LCA(u, v) is defined as the loop or function ancestor common to both \( u \) and \( v \) which has the greatest depth in the GCCFG. For example in Figure 2, LCA(c, g) = \( a \), LCA(g, h) = \( f \). The two types of interference are as follows:

1. Neither \( u \) nor \( v \) are the LCA. In this case, neither function is a descendant of the other and the base interference of the LCA node determines the interference between the two functions. In Figure 2, \( \text{inter}(b, d) = I_b(L1) = 10 \), and \( \text{inter}(g, b) = I_g(f) = 10 \).

2. Either \( u \) or \( v \) is the LCA. Here, the function which is not the LCA, say \( v \) is a descendant of the other function \( u \). The interference between the two functions is determined by the sum of the base interference of the LCA node and the base interference of the first child function of the LCA node on the path to the descendant function. In Figure 2, \( \text{inter}(a, d) = I_a(a) + I_a(d) = 1001 \), and \( \text{inter}(a, h) = I_a(a) + I_a(f) = 11 \).

The first function on the path from the LCA to the descendant function determines the interference relationship because from the ancestor’s perspective any function call initiated within this child node is indistinguishable from the memory presence of the child itself. Considering functions \( a \) and \( f \) again, if they are in the same region, their interference is illustrated with the trace

\[
\langle a, (f), a, (f), \ldots \rangle
\]

The two functions alternate in memory with each call to \( f \) and return to \( a \). If we consider placing the entire GCCFG branch headed by \( f \) into a single region, the functions will swap one another out in memory as given by the trace

\[
\langle a, (f, g, f, h, f, \ldots), a, (f, g, f, h, f, \ldots), \ldots \rangle
\]

Here the displacement between \( a \) and \( h \) is again once for every call to \( f \) and return to \( a \). We say that \( f \) is shielding \( a \) from the interference effects of its descendants.

Definition 5. A function, \( u \), shields its parent function, \( s \), from interference costs associated with all of its descendants, \( T \), such that when \( s \) and \( T' \subseteq T \) are mapped to the same region, \( \text{shield}(s, t) = u : \forall t' \in T' \) where \( u \) is the shielding function, and \( \text{inter}(s, T') := \text{inter}(s, u) \).

The function interference algorithm is presented in Figure 4. Lines 7-10 handle the case where one of the functions is the LCA, and line 12 handles the case where neither function is the LCA. The shielding property of interference relationships between functions proves important in making decisions about assigning functions to regions and in predicting the cost or quality of overlay regions and mappings as a whole which are discussed below in Section 5.1.

5. COG ALGORITHM FRAMEWORK
5.1 COG Overlay Cost Model

The cost of an overlay miss is defined as the actual associated DMA overhead in terms of time calculated from the size of the DMA. The cost associated with a given overlay mapping is the sum of the costs of its regions. We will first study the interferences between functions assigned to the same region in order to determine region costs. Functions assigned to separate regions do not interfere with one another, as they do not share the same address space in memory.

The calculation used to find interference between two functions has been described in Section 4.3. These interferences are the basis for calculating interference between functions and regions. The interference between a function, \( u \), and an overlay region, \( R \), is given by:

\[
\text{inter}_{R}(u, R) = \max_{v \in \text{shield}(u)} \left( \text{inter}(u, v) + \sum_{v \in \text{shield}(u)} \text{inter}(u, v) \right)
\]  

In Equation 1 we consider two sources of interference in the GCCFG structure between a function, \( u \), and all other functions assigned to the same region:

1. The first type of interference is associated with all functions in the region which are not descendants of the function under consideration, i.e. \( u \) is not the LCA. The first term on the right side of the equation gives the interference due to non-descendant functions in the region. Only the maximum interference is considered here since in general the number of times a function is overwritten between calls depends only on one common ancestor, \( s \), due to its shielding \( u \) from all other non-descendant functions in the region. From Definition 5 we know any other functions in the region are shielded from \( u \). If there were another function in the region on the path from \( s \) to \( u \), it would have an interference cost greater than or equal to \( \text{inter}(s, u) \), and it would be shielding \( u \) from \( s \).

2. The second term on the right side of Equation 3 gives the interference of \( u \) in \( R \) due to the descendants of \( u \) in \( R \), i.e. \( u \) is the LCA. Again, by Definition 5, for each of the immediate children, \( t \), of \( u \), the interference between \( u \) and any or all of the descendants of \( t \) is exactly \( \text{inter}(u, t) \). Consequently, the interference between \( u \) and all of its descendants is the sum of the interference costs between \( u \) and any of its immediate children with descendants in the region.

The expected cost of an overlay region is derived from the interference or number of overlay misses between its functions, and the DMA overhead associated with each overlay miss. DMA overhead is the actual cost of an overlay miss in terms of time spent retrieving the missing code segment. The time required to execute the DMA, which will refer to as simply \( cost \), is a function of the size of the missing function, \( u \):

\[
\text{cost}(u) = \text{time needed to DMA size}(u) \text{ Bytes}
\]  

The actual DMA overhead function giving cost in seconds as a function of DMA size is empirically determined by measuring DMA performance on the target architecture. This function as determined for the CBE is described as part of the simulation discussion in section 8.1.2.

The expected overhead cost of a function assigned to a region is given by:

\[
\text{cost}_{R}(u, R) = \text{cost}(u) \cdot \text{inter}_{R}(u, R)
\]  

The total expected overlay overhead cost associated with one region is the sum of the costs of all of its assigned functions given by:

\[
\text{cost}_{R}(R) = \sum_{u \in \text{shield}(R)} \text{cost}_{R}(u, R)
\]
Finally, the total expected overlay overhead associated with an overlay mapping, $OVL$, is the sum of the costs of all of its assigned regions:

$$\text{cost}_{\text{mapping}}(OVL) = \sum_{R \in OVL} \text{cost}_f(R)$$  \hspace{1cm} (5)$$

The value obtained from $\text{cost}_{\text{mapping}}(OVL)$ is used to classify overlay mappings according to expected performance, enabling classification of solutions in terms of expected performance.

### 5.2 COG Algorithm

The Code Overlay Generator (COG) Algorithm is designed to produce overlays which result in the smallest possible number of misses. Once interference costs have been calculated and we have a method for empirically analyzing the quality of function to region mappings, we can construct an algorithm to generate high performance overlays.

The algorithm works on a fixed number of regions and generates overlay mappings designed to minimize misses. It does not consider function sizes. As a result, we will not know the size of the generated overlay mapping until the algorithm is complete. To account for mapping size and find the best overlay mapping for a given memory size, we first calculate the lower bound on the number of regions which will fit in memory. This bound is determined by the second for loop (line 14). The complexity of the expansion algorithm can improve the overlay’s performance by creating new regions and moving functions into them. We call the simple extension to the COG algorithm given in Figure 6, COG Expansion (COG-E).

In line 8, the algorithm removes functions from previous regions beginning with the function having the greatest total cost as long as they fit into remaining memory and the source region contains more than one function. If a function does not fit into remaining memory we skip it and continue to the next function in descending order of total interference.

In order to further improve the solution, in line 15 we check for opportunities to reduce overlay cost by testing the functions again from greatest to least total interference to see if their cost in their currently assigned region is less than their cost in a new region as long as the move does not increase the size of the new region. This process can cause the overlay size to shrink since the largest functions from some regions may have been moved to larger regions. We can take advantage of this reduction in overlay size by running the algorithm again until remaining memory is too small to hold the smallest function. We avoid increasing the computational complexity of the algorithm by limiting the outer loop to some small number of iterations.

Increasing the number of regions is guaranteed to reduce overlay cost since removing a function from a region with multiple functions must reduce the region’s interference cost if multiple functions are called in that region, and adding a function to an empty region incurs no cost. The complexity of the expansion algorithm is determined by the second for loop (line 14). The complexity of the for loop and consequently, the COG-E algorithm is $O\left(n^3\right)$. 

---

**Figure 5: COG Algorithm.**

**Figure 6: COG-E Algorithm.**

**Figure 7: COG-C Algorithm.**
5.4 COG-Compression

As a second extension to the COG algorithm, we consider COG Compression (COG-C). When COG returns an overlay mapping which is too large to fit in the SPM, we attempt to compress the size of the overlay until it will fit. This is done by evaluating regions other than the largest one and systematically moving their largest assigned function into a larger region. Each such function move can reduce the size of the overlay if all remaining functions are smaller than the largest function. Once the overlay is small enough to fit into available memory, we stop and return to the COG Algorithm where the compressed overlay is retained if it has a lower cost than the best solution so far. The complexity of the algorithm is again limited by restricting the outer loop to a small constant size. The algorithm is given in Figure 7. The loop at line 5 runs for \( n \) iterations with at most \( n \) comparisons per iteration giving the COG-C algorithm a complexity of \( O(n^2) \).

5.5 COG, Unified Algorithm

In practice, the three COG algorithms presented can be run simultaneously to select the solution with best predicted overlay performance. Since COG is the first step of each algorithm for each number of regions, the first step in the unified algorithm is to run the COG algorithm. When the COG solution is found to be smaller than available memory, we will run COG-E to find an improved solution which better utilizes memory. In the event that the COG solution is found to be larger than available memory, we run COG-C in an effort to find a mapping with reduced size and improved performance over the last valid COG solution with fewer regions. As is the case in the previously described algorithms, we retain the best solution at each step, and return the overall best solution after exhausting the region search space.

The computational complexity of the combined algorithm is again \( O(n^2) \), since we find a solution for each number of regions in the search space only once, and execute at most one COG extension algorithm per iteration. Although this method will return the solution with the best predicted performance from among COG and its two extensions, it is not guaranteed to return the mapping which gives the best real world performance, as compile time evaluation of the solutions is limited by the accuracy of the performance model.

6. SDRM ANALYSIS

6.1 SDRM Cost Model

In the SDRM algorithm [8], interference costs are calculated similar to our method, but only base costs are considered. The incorporation of return interference is an important new contribution in our model. In SDRM the interference used is the minimum of the two base costs for \textit{callee-callee} relationships where neither node is the common ancestor, and the base cost of the descendant is used for \textit{caller-callee} relationships. The cost is multiplied by the sum of the two function sizes since function size is understood to correlate with performance/energy overhead in the event of an overlay miss. In addition, the SDRM model calculates region cost based on the sum of interferences between all functions assigned to the region without accounting for the effects of shielding on function interference. In order to make valid comparisons between the COG and SDRM models, we use the DMA cost model as described in Equation 2 when calculating SDRM region costs instead of using function sizes directly.

6.2 Deadlock

SDRM uses the interference costs described in Section 6.1 to construct an overlay mapping. The interference costs are annotated on edges between functions in an interference graph. These edges are sorted from most expensive to least expensive. The most expensive edge is selected and each of its associated functions are added to newly created overlay regions, then the next edge is selected. If there is not enough unused memory to create a new region, the function is added to one of the existing regions such that the cost of adding the function is minimized. In the worst case, the SDRM algorithm makes \( n \) comparisons to select a region by traversing \( n^2 \) edges, for a computational complexity of \( O(n^3) \).

The scheme can run into trouble as illustrated in the example GCCFG in Figure 8(a). The interference graph shows that the most expensive edge occurs between functions \( a \) and \( b \). Figure 8(b) illustrates the behavior of the SDRM algorithm for the given SPM size. The algorithm first selects the edge \((a, b)\) adding functions \( a \) and \( b \) to new regions in the SPM. When \( c \) is selected, we find it is larger than the size of either region plus the remaining SPM memory. The consequence is that the algorithm hangs without producing a solution.

6.3 Performance Case Study

Figure 3 illustrates a \textit{flat} call graph structure commonly found in computer programs. After inspecting the call graphs for each of our benchmarks, we find that on average 67% of the functions are direct members of such structures even if we count only instances of the structure with at least four nodes. Given the data structure in the figure and assuming all functions are the same size, the interference graph generated by SDRM has the same weight on every edge. The algorithm first selects the edge \((a, b)\) and each of its associated functions are added to new regions in the SPM. When \( c \) is selected, we find it is larger than the size of either region plus the remaining SPM memory. The consequence is that the algorithm hangs without producing a solution.
and $R_2 : < b, ..., n >$. For COG the traces are $R_1 : < a >$ and $R_2 : < b, c, ..., n >$. We can see that the return costs of $a$ inflate the number of misses in the SDRM mapping. The cost of each mapping can be given in terms of misses for the SDRM and COG solutions with $n$ functions and $r$ regions as follows:

$$\text{misses}(SDRM) = n + \frac{r}{2} - 1 : r, n \in \mathbb{N} ; r > 1$$ (6)

$$\text{misses}(SDRM) = n + \frac{r}{2} - 1 = \frac{3r}{2} - 1 \approx \frac{3}{4}n : r = 2$$ (7)

$$\lim_{r \to n} \left( \text{misses}(SDRM) = n + \frac{r}{2} - 1 \right) = n$$ (8)

$$\text{misses}(COG) = 1 + r : n \in \mathbb{N}$$ (9)

Equations 6 and 9 describe the number of overlay misses experienced by the SDRM and COG solutions respectively. Equations 7 and 8 indicate that in the case of the structure in Figure 3, the SDRM solution is at best equal to the COG solution described in Equation 9 as the number of regions approaches the number of functions (i.e. as available memory increases), and at worst 50% more expensive than COG in the case of a two region solution given in Equation 7. Increasing the number of regions improves the SDRM solution, but COG is optimal even when there are only two regions, as $n$ is the smallest possible number of misses.

7. SPU-GCC ANALYSIS

The spu-gcc automatic overlay algorithm is described in [10]. It has a lower complexity, working on at most $n^2$ edges, of $O(n^2)$. The algorithm clusters highly interfering neighboring functions into the same segment as long as the segment fits in the memory region. Interference costs in the spu-gcc compiler are considered only between parent-child nodes, and are equivalent to the base cost of the child node in our scheme [10].

A key distinction between this algorithm and COG/SDRM is that it relies exclusively on code segments rather than regions to group functions into an overlay mapping. The compiler algorithm always returns a mapping with exactly one region. Each segment may contain one or more functions, unlike SDRM and COG, both of which ignore the potential benefit of placing multiple functions into one segment.

Since COG and SDRM use a separate segment for each function, performance for any solution generated with a single region gives worst-case performance because every function call results in a call and return miss. For this reason we expect the spu-gcc solution to outperform COG and SDRM when the solution consists of just one region. However, large segments quickly become a handicap in the spu-gcc algorithm as the size of available memory is increased. Performance is hampered with larger segment sizes because any misses during program execution result in memory access overhead which is a function of segment size. As the region size grows further, performance begins to improve again, approaching the optimal with every function always in the SPM and mapped to one segment containing the entire program. These effects will be highlighted in the discussion of experimental results.

In our experiments, spu-gcc overlays have been generated for our benchmarks using the –auto-overlay option described in the IBM Cell Programmer’s guide [2]. We modified the benchmark source code by adding a large data buffer which consumes sufficient LS space to force the compiler to generate multiple code segments in order to fit the instructions in memory. Larger buffer sizes result in smaller overlay mapping sizes. By examining the compiler output for a range of buffer sizes, we are able to determine the configurations needed to generate spu-gcc overlay solutions that sweep the program size. The size of the automatically generated overlay mappings is found by parsing the generated linker script to find the largest segment in the compiler’s one region solution.

8. EXPERIMENTAL RESULTS

8.1 Experimental Setup

8.1.1 Benchmarks and Memory Setup

The benchmarks used for experimentation are presented in Tables 2 and 3. We use a distinct set of benchmarks for simulation based analysis (Table 2) and execution on the CBE (Table 3). For each benchmark we first select memory sizes needed to test performance across a meaningful range of available SPM instruction memories. Benchmarks with smaller code sizes have been swept from the smallest possible overlay mapping size, defined by the size of the largest function in the program, up to the total size of the program, or the sum of sizes of all functions. This sweep is done in 15 steps for small benchmarks. For larger benchmarks the sweep is done in 10 steps and ranges from the smallest possible overlay mapping to one half of the full program size. The gsm benchmark is also swept up to the full program size. These ranges can be found in Tables 2 and 3 along with the number of functions present in each benchmark.

We consider benchmarks with less than 10 functions small. When selecting benchmarks for testing, every effort has been made to utilize available benchmarks with a large number of functions. Previous work, including SDRM [8], have only given results for benchmarks with very few functions. We assert that results achieved using larger benchmarks are more interesting for two reasons. Benchmarks with 10 or fewer functions can actually be solved optimally with a heuristic strategy. This sweep is done in 15 steps for small benchmarks. For larger benchmarks the sweep is done in 10 steps and ranges from the smallest possible overlay mapping to one half of the full program size. The gsm benchmark is also swept up to the full program size. These ranges can be found in Tables 2 and 3 along with the number of functions present in each benchmark.

We consider benchmarks with less than 10 functions small. When selecting benchmarks for testing, every effort has been made to utilize available benchmarks with a large number of functions. Previous work, including SDRM [8], have only given results for benchmarks with very few functions. We assert that results achieved using larger benchmarks are more interesting for two reasons. Benchmarks with 10 or fewer functions can actually be solved optimally with a heuristic strategy. This sweep is done in 15 steps for small benchmarks. For larger benchmarks the sweep is done in 10 steps and ranges from the smallest possible overlay mapping to one half of the full program size. The gsm benchmark is also swept up to the full program size. These ranges can be found in Tables 2 and 3 along with the number of functions present in each benchmark.

We consider benchmarks with less than 10 functions small. When selecting benchmarks for testing, every effort has been made to utilize available benchmarks with a large number of functions. Previous work, including SDRM [8], have only given results for benchmarks with very few functions. We assert that results achieved using larger benchmarks are more interesting for two reasons. Benchmarks with 10 or fewer functions can actually be solved optimally with a heuristic strategy. This sweep is done in 15 steps for small benchmarks. For larger benchmarks the sweep is done in 10 steps and ranges from the smallest possible overlay mapping to one half of the full program size. The gsm benchmark is also swept up to the full program size. These ranges can be found in Tables 2 and 3 along with the number of functions present in each benchmark.

8.1.2 Simulation

Overlay mapping performance for benchmarks in Table 2 has been validated through simulation of memory access overhead in the CBE between main memory and the SPU Local Store due to code overlay misses. Our simulator has been developed to validate the overlay source code and scheme before moving ahead with laborious effort needed to modify large benchmarks to actually run on the CBE using code overlays.

Inputs to the simulator are execution trace data, function sizes, and the overlay mapping to be evaluated. Benchmark execution traces were generated by instrumenting the source code with print statements. The trace consists of function names in the order in which they must to be present in memory during program execu-
Each function in the program has been modified to emit its name into the output trace upon entry and immediately following any function calls present in the function body. The resulting trace accounts for function calls and returns as described in Section 4.2, enabling accurate accounting of interference between functions.

The simulator maintains a table representing the state of the overlay map in memory as the trace is consumed. Miss overhead is recorded whenever a function appears in the trace and does not reside in the memory table, and the overlay map state table is updated to indicate the appropriate code segment is present. It is important to note that rather than an approximation, the number of misses recorded for each function in the simulator is identical to the number which would have been observed during actual execution. DMA overhead cost of a benchmark function, \( f \), in the simulation is calculated as a function of segment size according to real DMA overhead measurements taken on the CBE which we observe can be modeled to within an average error of 0.4% using the following equation:

\[
DMA_{cost}(f) = \begin{cases} 
3.9E-5 \cdot \text{size}(f) + 0.17\mu s & \text{size}(f) \leq 2\text{KB} \\
7.3E-5 \cdot \text{size}(f) + 0.1\mu s & \text{size}(f) > 2\text{KB}
\end{cases}
\]

DMA overhead costs are accumulated as the simulator executes, and the total overhead returned at the end of the input trace is the data point we plot in performance evaluation as seen in Figures 12(b) and 15.

8.1.3 IBM Cell Implementation

The benchmarks in Table 3 have been modified to run using user defined overlay mappings with function level resolution on a single CBE SPU. The SPU compiler, spu-gcc, is capable of taking a user defined linker script as an input in order to map code into LS memory according to the user’s overlay scheme [2]. When specifying the mapping of functions to segments and segments to regions, code objects must be specified in the script as individual object files. Consequently, in order to implement and evaluate arbitrary code overlay mappings at function level resolution, each function must be placed in a separate source file.

In order to obtain performance results on the CBE, linker scripts describing overlay solutions for each algorithm at each tested memory size are generated. Next, each benchmark is compiled once for each linker script, resulting in a separate executable for each data point. The benchmarks are also instrumented to print total execution time information after each run. The executables are then run multiple times and the resulting timing data is averaged to obtain the final data point as presented in Figures 13 and 14.

8.2 Results

In each of these plots, we notice that the COG extension algorithms, COG-E and COG-C, both produce better performing overlays than the unextended COG algorithm and SDRM, particularly when memory is severely restricted. The limited performance of the unextended algorithm is due to the fact that it can produce solutions which do not fully utilize available memory as corrected by
the COG-E algorithm, and the extension algorithms tend to produce solutions with more regions. The relationship between an overlay mapping’s performance and number of regions is further evaluated in Section 8.2.1.

The overall performance results comparing SDRM and the COG extension algorithms are given in Figures 10 and 11. The size of each benchmark is also indicated on the graph in terms of number of functions. There are several instances where the SDRM algorithm fails to generate a solution due to deadlock during overlay mapping generation. In particular, for the problem space we have defined on idecod and ispell in Figure 10, the SDRM algorithm does not return a solution for any data point. For other benchmarks, as seen in the SDRM trace in Figure 12(b), SDRM generates solutions for some memory sizes, typically on the larger end of the sweep. On average, the COG algorithms out perform SDRM by 38% in the simulated benchmarks, and by 16% in benchmarks executed on the CBE.

As mentioned above, we find that performance analysis is more meaningful when the number of functions in the program is larger. For the smallest benchmarks, rijndael, dijkstra, sha, and patricia, quantum effects due to so few functions amplifies the impact of individual mapping decisions. This effect is clear in the overlay performance results as illustrated in Figure 15. The SDRM solution in the figure has been impacted in particular by a compile time decision which has a substantial negative impact on the performance of its solutions above 6kB of instruction memory, where we would normally expect performance to improve.

8.2.1 Number of Regions and Performance

The solution generated for the smallest possible overlay mapping is always limited to one region, where the solution for the largest possible overlay mapping normally contains a region for each function. Figure 12(a) illustrates the number of regions in solutions generated by each algorithm for gsm. The three COG algorithms generate the same solution when available memory is restricted to the size of the largest function as indicated by the convergence of the three traces on the left. The mapping algorithms again produce identical solutions once memory size matches the total program size at the upper right in the graph since each function can be assigned to its own region. The SDRM solution is shown to have fewer regions at this point in the graph because in the SDRM model, functions whose common ancestor is main are assigned no interference cost in the interference graph and as a result may end up sharing a region although space is available in memory to create new regions and separate them.

We expect mapping solutions with more regions to give lower overhead costs, as only functions mapped to the same region can interfere with one another. We observe from Figure 12(a) that solutions with larger region counts generally do perform better as evident by the corresponding performance chart in Figure 12(b). In particular, the spa-gcc solution which is limited to one region, experiences a substantial performance disadvantage as available memory enables solutions with more regions from COG and SDRM.

8.2.2 Performance Model Accuracy

Our assessment of performance model accuracy is based on the ability of each model to predict, given two overlay mappings, which mapping will actually give better performance. We have plotted the performance predictions from the COG and SDRM models for each simulation benchmark and compared them against the simulated performance results.

For the gsm benchmark we see predictions made by the COG model in Figure 16(b) and predictions using the SDRM model in Figure 16(c). The actual results are shown for comparison in Figure 16(a). By inspection we find that, as expected, the SDRM cost model tends to give overly optimistic predictions for solutions generated by the SDRM algorithm. This is evident when comparing the SDRM trace in Figure 16(b) with the SDRM trace in Figure 16(c). We empirically test the ability of each model to predict performance by counting the number of times each model correctly predicted which overlay solutions actually performed better independently for every pair of algorithms in COG, COG-E, COG-C, and SDRM.
The results of these comparisons are presented in Figure 17. We find that on average, across all benchmarks, COG correctly predicts overlay performance 6.5% more often than the SDRM performance model.

9. CONCLUSION

We have presented a code overlay generator designed to map instructions onto limited Scratchpad Memories for improving performance in embedded systems. We describe an overlay mapping cost model for identifying good solutions at compile time without the benefit of profiling information. Our algorithm performs better than the previously published heuristic, while eliminating the deadlocking problem experienced in the previous work. We also show that our algorithm is able to perform substantially better than the scheme provided with the IBM Cell Broadband Engine compiler spu-gcc. Algorithms for gathering functions into segments for improved performance at the most restricted memory sizes, and improved computational complexity are targeted in future work. Lower complexity algorithms based on our model will also enable runtime profiling and evolutionary methods for finding high quality solutions, particularly in cases where program behavior is highly dependent on input data. The accuracy of our model and algorithm performance are also expected to improve if loop sizes are known at compile time through code analysis or profiling and included in the GCCFG directly.

10. ACKNOWLEDGMENTS

Michael A. Baker is sponsored by the U.S. Army’s Research Development and Engineering Command. The contents of this work reflect the views of the writers, who are responsible for the accuracy of the data presented herein. The contents do not necessarily reflect the official views or policies of the U.S. Army. This research has been supported in part by grants from Science Foundation Arizona (SFAz), Stardust Foundation, and the National Science Foundation (CCF - 0903513).

11. REFERENCES