Automatic Code Overlay Generation and Partially Redundant Code Fetch Elimination

CHOONKI JANG, Samsung Electronics
JAEJIN LEE and BERNHARD EGGER, Seoul National University
SOOJUNG RYU, Samsung Electronics

There is an increasing interest in explicitly managed memory hierarchies, where a hierarchy of distinct memories is exposed to the programmer and managed explicitly in software. These hierarchies can be found in typical embedded systems and an emerging class of multicore architectures. To run an application that requires more code memory than the available higher-level memory, typically an overlay structure is needed. The overlay structure is generated manually by the programmer or automatically by a specialized linker. Manual code overlaying requires the programmer to deeply understand the program structure for maximum memory savings as well as minimum performance degradation. Although the linker can automatically generate the code overlay structure, its memory savings are limited and it even brings significant performance degradation because traditional techniques do not consider the program context. In this article, we propose an automatic code overlay generation technique that overcomes the limitations of traditional automatic code overlaying techniques. We are dealing with a system context that imposes two distinct constraints: (1) no hardware support for address translation and (2) a spatially and temporally coarse grained faulting mechanism at the function level. Our approach addresses those two constraints as efficiently as possible. Our technique statically computes the Worst-Case Number of Conflict misses (WCNC) between two different code segments using path expressions. Then, it constructs a static temporal relationship graph with the WCNCs and emits an overlay structure for a given higher-level memory size. We also propose an inter-procedural partial redundancy elimination technique that minimizes redundant code copying caused by the generated overlay structure. Experimental results show that our approach is promising.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Code generation, Compilers, Optimization*

General Terms: Algorithms, Design, Experimentation, Measurement, Performance

Additional Key Words and Phrases: Code overlays, Path expressions, Static temporal relationship graphs, Partial redundancy elimination

ACM Reference Format:
Jang, C., Lee, J., Egger, B., and Ryu, S. 2012. Automatic code overlay generation and partially redundant code fetch elimination. ACM Trans. Architec. Code Optim. 9, 2, Article 10 (June 2012), 32 pages.
DOI = 10.1145/2207222.2207226 http://doi.acm.org/10.1145/2207222.2207226

10

This work was supported in part by grant 2009-0081569 (Creative Research Initiatives: Center for Manycore Programming) from the National Research Foundation of Korea. It was also supported in part by the Ministry of Education, Science and Technology of Korea under the BK21 Project. ICT at Seoul National University provided research facilities for this study. This work was done when C. Jang was a Ph.D. student at Seoul National University.
Author's addresses: C. Jang, Samsung Electronics; email: choonki.jang@samsung.com; J. Lee and B. Egger, School of Computer Science and Engineering, Seoul National University; S. Ryu, Samsung Electronics.

DOI 10.1145/2207222.2207226 http://doi.acm.org/10.1145/2207222.2207226

1. INTRODUCTION

Explicitly managed memory hierarchies, where the programmer or software is responsible for explicitly moving code and data between different levels of the hierarchy, can be found in many embedded processor architectures, such as ARM embedded and application processors [ARM Ltd. 2009], TI digital signal processors [Texas Instruments Inc. 2009], and an emerging class of multicore architectures, such as IBM Cell BE [IBM et al. 2007] and Intel Terascale [Mattson et al. 2008].

The explicitly managed memory hierarchy typically consists of two levels of different memories. The higher-level memory is typically a faster but smaller storage close to the processor core, such as scratchpad memory in embedded processors and local stores in the Cell BE or Terascale processor. In contrast, the lower-level memory is a slower but larger storage, such as external main memory.

To run an application that requires more code memory than the available higher-level memory, an overlay structure is needed. The overlay structure is typically generated manually by the programmer or automatically by a specialized linker [Cytron and Loewner 1986; IBM 2007; Levine 2000]. Manual code overlaying requires the programmer to deeply understand the program structure to achieve maximum memory savings as well as minimum performance degradation.

While automatic overlay generation techniques can significantly mitigate the programmer's burden to build overlaid executable, traditional techniques have some limitations. First, traditional algorithms [Cytron and Loewner 1986; Levine 2000] cannot generate an overlay structure for an arbitrary memory size because they do not allow different code segments to be mapped to the same memory region when they reside in the same call path. Consequently, memory savings are limited by the maximum total size of all code segments in the same call path. A *code segment* in this article is a code fragment that can be separately loaded into the higher-level memory.

Second, traditional algorithms do not consider the degree of potential conflict misses between two different code segments when they are mapped to the same higher-level memory region. For example, consider the call sites of two different code segments `A` and `B` that are located in the same loop. Assume that `A` and `B` are mapped to the same higher-level memory region in the overlay structure. In each iteration of the loop, `A` and `B` need to be copied to the memory because they overwrite each other whenever they are called. This results in generating an inefficient overlay structure. In turn, this introduces a significant overhead associated with memory copying.

Third, traditional techniques may have many redundant memory copying operations due to the difference in the size of code segments. When two code segments in different sizes are mapped to the same memory region, one does not entirely overwrite the other. Thus, we need to copy only the overwritten part to the higher-level memory when the other is called.

Finally, traditional techniques do not deal with functions called through function pointers. These functions are always mapped to a non-overlaid memory region, resulting in an inefficient overlay structure.

This article proposes an automatic code overlay generation technique that overcomes the limitations of traditional techniques. Especially, we are dealing with a system context that has explicitly managed small code memory and imposes two distinct constraints: (1) no hardware support for address translation and (2) a spatially and temporally coarse grained faulting mechanism (i.e., at the function level) that incurs heavy overhead and requires fault minimization. This article addresses those two constraints as efficiently as possible.

Our technique computes the Worst-Case Number of Conflict misses (WCNC) between two different code segments and the Worst-Case Number of Invocations (WCNI) of a code segment using Tarjan's path expressions [Tarjan 1981a, 1981b]. It constructs a

Static Temporal Relationship Graph (STRG) with the WCNCs and emits the overlay structure in a greedy manner based on the STRG and the WCNIs. The STRG summarizes the degree of potential conflict misses between two different code segments when they are mapped to the same higher-level memory region. After generating the overlay structure, it minimizes redundant code fetches by using a partial redundancy elimination technique. Our technique generates an overlay structure for an arbitrary memory size as long as the memory size is bigger than the maximum code segment size in the program.

We build a linker that performs a whole-program analysis. The linker implements both our algorithms and previous automatic code overlaying algorithms. Our experimental results show that our approach outperforms the previous work both in memory savings and execution time. Our technique is general enough to be applicable to any explicitly managed memory hierarchies that can be found in contemporary multicore architectures.

The major contributions of this article are the following.

—We propose a systematic method to compute the degree of potential conflict misses (i.e., WCNC) between two different code segments using path expressions and to construct an STRG based on the WCNCs.
—We propose a greedy algorithm that generates an efficient overlay structure using the STRG.
—We propose both intraprocedural and interprocedural partial redundancy elimination algorithms that eliminate redundant code fetches.
—We show the effectiveness of our approach by implementing a linker and comparing its memory savings and execution time with those of the previous approaches.

The rest of this article is organized as follows. Section 2 discusses related work. We briefly describe a typical automatic overlay generation algorithm in Section 3. The overlay generation algorithm is described in Section 4. Section 5 presents the method to insert and optimize the runtime code for overlaying. Section 7 presents the evaluation result. The article concludes in Section 8.

2. RELATED WORK

Cytron and Loewner [1986] propose a technique that automatically and statically generates an overlay structure for a given program. This is a typical traditional code overlaying technique. Pabalkar et al. [2008] propose an automatic overlay generation algorithm with a static analysis that abstracts the behavior of the program with the global call control flow graph (GCCFG). The overlay generation algorithm proposed by Jung et al. [2010] exploits the GCCFG whose edges are weighted with the profiling information. We elaborate on these techniques in Section 3. He et al. [2007] propose an on-demand code overlaying technique for the kernel of an operating system. It keeps infrequently executed kernel code segments in the secondary storage and loads them on-demand. Their technique is based on profiling information.

Managing code scratchpad memory (SPM) has been studied a lot in embedded systems to reduce power consumption and/or improve performance [Angiolini et al. 2004; Banakar et al. 2002; Egger et al. 2006a, 2006b, 2008; Steinke et al. 2002; Verma et al. 2004a, 2004b]. Unlike ours, all of them use profiling information to efficiently manage the SPM. Moreover, they assume that a code segment can be directly executable in the lower-level memory without copying it into the higher-level memory. Such techniques cannot be applicable to the local store management in the Cell BE multicore processors, where the code segment must be copied to the local store to be executed.

When we treat the higher-level memory as a cache that consists of cache lines with a variable size, the code overlaying problem is similar to the code placement problem to minimize cache misses. However, as mentioned in the previous section, we are dealing with a system context that imposes two distinct constraints: (1) no hardware support for address translation and (2) a spatially and temporally coarse grained faulting mechanism. Our approach addresses those two constraints as efficiently as possible. Pettis and Hansen [1990] propose a code positioning technique to improve instruction cache performance. It is based on profiling information. Hashemi et al. [1997] propose an algorithm that uses a weighted call graph to minimize conflict misses in instruction caches. Kalamatianos and Kaeli [1998] improve Hashemi et al.'s approach by using a conflict miss graph (CMG) to model the temporal conflict relationship between functions. Gloy et al. [1997] propose a mechanism that minimizes conflict misses in conventional direct-mapped instruction caches. They exploit the temporal relationship graph (TRG) of a given program. It connects any pair of functions if they are interleaved in the call and return trace of program execution. The TRG is based on profiling information, the trace of function references induced by calls and returns during the execution of the program. Based on the TRG, they position functions to minimize conflict misses. Guillon et al. [2004] prove that finding an optimal allocation that minimizes cache conflicts using the TRG is NP-complete. Our algorithm is based on static TRGs and path expressions that do not use any trace or profiling information.

Knoop et al. [1992] propose a partial redundancy elimination algorithm that is computationally optimal. Agrawal et al. [1995] propose an interprocedural partial redundancy elimination algorithm and applies it to communication optimizations for distributed memory multiprocessors. Our techniques of redundant memory copying operations are based on these techniques.

Larus [1999] proposes whole program paths (WPP) to measure a program's control flow that captures the program's dynamic behavior. The whole program paths are represented in a DAG representation of a context-free grammar that generates the program's acyclic path trace. His technique works interprocedurally and is based on profiling information. Unlike WPPs, our technique is static, and our path expression captures the alternating behavior of functions that are contained in a program.

3. TRADITIONAL TECHNIQUES

Code overlaying techniques have been used to execute applications that require more code memory than the available physical code memory can hold. The user or an automatic overlay generator [Cytron and Loewner 1986; IBM 2007; Levine 2000] builds an overlaid executable by specifying the overlay structure in a linker script. Such an overlay description consists of physical memory region specifications. Each memory region specification contains code segment names that are overlaid to the physical memory region. The unit of overlaying is a code segment. A code segment contains one or more functions in the program.

Cytron and Loewner [1986] propose an automatic code overlay generation algorithm for a linker. It automatically generates the linker script without any runtime information. After constructing the static call graph (SCG) for a given program, the algorithm collapses each strongly connected component (SCC) into a single node to create a Directed Acyclic Graph (DAG). Code segments that call each other need to be resident in the code memory at the same time; hence the code segments in each SCC cannot be overlaid in the same memory region. Consider the SCG in Figure 1(a) and the corresponding DAG in Figure 1(b). The DAG is converted into a tree by moving nodes with two or more incoming edges into their immediate dominators. A node with several incoming edges appears in different call paths and thus it must be resident in the memory for all call paths. By absorbing such nodes in their immediate dominator,

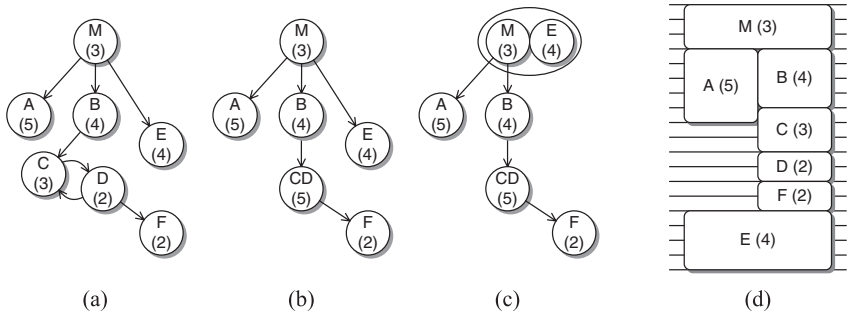


Fig. 1. An example of code overlays.

the DAG forms a tree (Figure 1(c)). The subtrees of these nodes cannot be mapped to the same memory region because the absorbed node appears in the effective call tree of several subtrees. For example, in Figure 1(c), the subtree of `E` (i.e., `E` itself) and the subtree of `M` (`M`, `A`, `B`, `CD`, and `F`) cannot occupy the same memory region. From the tree-shaped DAG, the final overlaid memory layout is computed. This traditional overlay generation algorithm proposed by Cytron and Loewner [1986] takes $O(N)$ time for a program with N functions. Figure 1(d) shows the final overlay structure computed from the tree-shaped DAG in Figure 1(c).

Based on the generated overlay structure, the linker relocates the code and inserts glue code in front of each call [Cytron and Loewner 1986; IBM 2007; Levine 2000]. The inserted glue code hooks into the overlay manager that ensures the target code segment is loaded in memory. If the target segment already resides in memory, the overlay manager simply forwards control to the target segment. Otherwise, it forwards control to the target segment after loading the target segment into memory. Consequently, different functions in the same call path are guaranteed not to be mapped to the same memory region by the generation algorithm; hence the caller always resides in memory when the callee returns.

Pabalkar et al. [2008] propose a technique that automatically generates an overlay structure with a static analysis. It summarizes the program into the global call control flow graph (GCCFG). The technique is called simultaneous determination of regions and function-to-region mapping (SDRM). The GCCFG captures function call sequences as well as control flow information such as loops and conditionals. A node in a GCCFG represents a function call (F-node), a loop (L-node), or a conditional (I-node). The weight of a node in a GCCFG estimates its execution count. The weight of each F-node is assigned by traversing the GCCFG in a top-down fashion. All F-nodes are initially weighted by one. Each child node of an I-node is assumed to be executed at the same rate. When an L-node is encountered, all descendents of the L-node are multiplied by a constant loop factor, say $Q = 10$. The weight of an F-node that contributes a cycle in the GCCFG due to a recursive function call or SCC in the call graph is multiplied by a constant recursive factor, say $R = 2$. Then, all back edges (e.g., dotted line in Figure 3(a)) are removed from the GCCFG. For example, the GCCFG for the CFGs in Figure 2 is shown in Figure 3(a).

When two functions are placed in the same memory region and called one after another, they are said to interfere with each other in their approach. To capture the interference relationship, a GCCFG is transformed into an interference graph (I-Graph). A node in the I-Graph represents a function in the program. Each edge (f, g) in the I-Graph is weighted by the interference cost between the two functions f and g. The interference cost between two functions estimates the amount of memory copying for

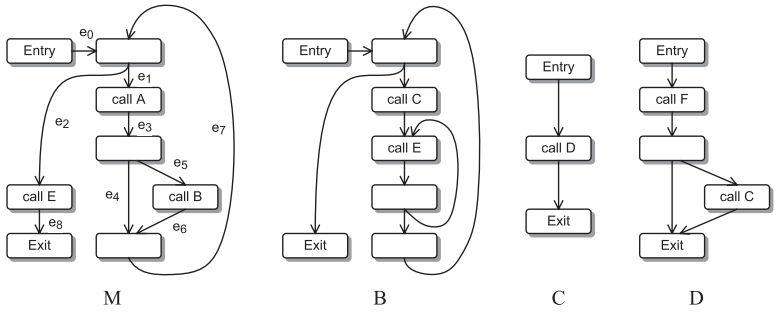


Fig. 2. The CFGs for M, B, C, and D.

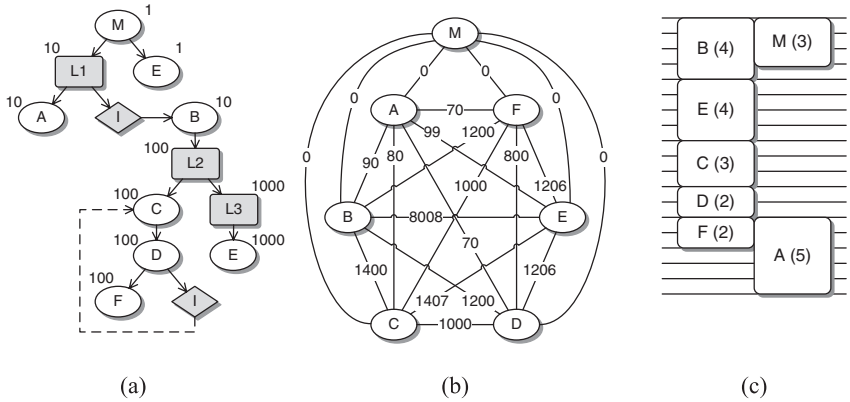


Fig. 3. (a) The GCCFG. (b) The I-Graph. (c) The overlay structure by SDRM with memory size 18.

two functions during the execution of the program when they are placed in the same memory region. For each pair of F-nodes n_x and n_y $(x \neq y)$ in the GCCFG, the weight of an edge (x, y) in the I-Graph is incremented by the following rules:

$$W(n_x, n_y) = \begin{cases} 0 & \text{if } n_{LCA} = Entry \\ (s_x + s_y) \times W(n_y) & \text{else if } n_{LCA} = n_x \\ (s_x + s_y) \times W(n_x) & \text{else if } n_{LCA} = n_y \\ (s_x + s_y) \times MIN(W(n_x), W(n_y)) & \text{otherwise,} \end{cases}$$

where s_x and s_y are the sizes of functions x and y, respectively. In addition, n_{LCA} is the lowest common ancestor (LCA) of n_x and n_y in the GCCFG, and $Entry$ is the entry function of the program. The resulting I-Graph is shown in Figure 3(b). Each edge in the I-Graph is selected in descending order of its weight. When an edge is selected, each function of the edge is placed in a memory region with the least interference cost if the function has not been placed. SDRM takes $O(V_f \cdot E)$ time for a program where V_f is the number of F-nodes in the GCCFG and E is the number of edges in the I-Graph [Pabalkar et al. 2008]. The final overlay structure is shown in Figure 3(c).

Jung et al. [2010] exploits the GCCFG where each node is assigned a weight based on analysis of profiling information. They propose two placement algorithms. One is function mapping by updating and merging (FMUM) and the other is function mapping by updating and partitioning (FMUP). FMUM begins with an overlay structure in which each function is placed in a different region. It repeatedly selects and merges a pair of memory regions with the minimal interference cost among all pairs of memory

regions until the overlay structure fits in the given memory size. On the other hand, FMUP begins with an overlay structure in which all functions are placed in the same memory region. It repeatedly selects the function with the maximal interference cost and places it in another region with minimal interference cost until the size of the overlay structure is less than the given memory size. FMUP and FMUM take $O(N^4)$ time to generate an overlay structure for a given program with N functions [Jung et al. 2010]. The number of interferences is estimated in the same way as in SDRM.

We compare our approach with SDRM, FMUM, and FMUP in Section 7.

4. OVERLAY STRUCTURE GENERATION

In this section, we present an algorithm that generates efficient code overlays for a given size of the higher-level memory containing the code. In theory, we can overlay every function in the program in the higher-level memory if the size of the higher-level memory is larger than the biggest function of the program. However, this approach may introduce a significant memory copying overhead.

We say that two code segments *conflict* when their assigned memory regions overlap in the higher-level memory. A *conflict miss* occurs when a code segment is being referenced by the program and currently does not reside in the higher-level memory because it has been replaced by another code segment in the higher-level memory. The more conflict misses occur, the bigger becomes the memory copying overhead. The goal is thus to minimize this copying overhead by reducing the number of conflict misses in the higher-level memory. To minimize the conflict misses between two code segments, x and y, which consist of different functions and are called interchangeably, their memory regions are ideally mapped to non-overlapping memory regions in the higher-level memory.

The first step in our overlay structure generation algorithm is building a *static temporal relationship graph* (STRG) for a given program. Each node in the STRG represents a function in the program. There exists an edge between two functions x and y if there exists a reference to y between two consecutive references to x in the execution. Every edge is annotated by a weight $W(x, y)$, which represents the number of alternations between x and y in the execution. The weight of an edge estimates the number of potential conflict misses between x and y during program execution assuming that they are mapped to the overlapping memory regions. When constructing the STRG, we statically estimate $W(x, y)$ between two different functions.

4.1. Static Temporal Relationship Graphs

Path expressions. To capture the alternating behavior between two functions and statically approximate the program flow in a context-sensitive manner, we construct path expressions for every function. A path expression is a regular expression representing all paths between two nodes in a directed graph. Tarjan [1981a, 1981b] proposes an efficient algorithm that constructs path expressions from a directed graph G.

A regular expression R over a given finite alphabet Σ represents a set $\sigma(R)$ of strings over Σ, where ϵ denotes the empty string and $\cdot$, $|$ and * denote concatenation, union, and Kleene closure operations, respectively. A path expression $P(v, w)$ on a directed graph $G(V, E)$ is a regular expression over E such that every string in $\sigma(P)$ is a path from v to w. We denote by P_f the path expression $P(Entry, Exit)$ of a function f, where $Entry$ and $Exit$ are the unique Entry and Exit nodes in the CFG of f. Thus, P_f represents the set of all paths from $Entry$ to $Exit$ in f.

We are interested in the alternating behavior between function references; however, paths are represented by edges in path expressions. We therefore replace each node that contains a function call in the CFG of a function f with a new edge that represents the function call. Each incoming edge to the node containing the function call becomes

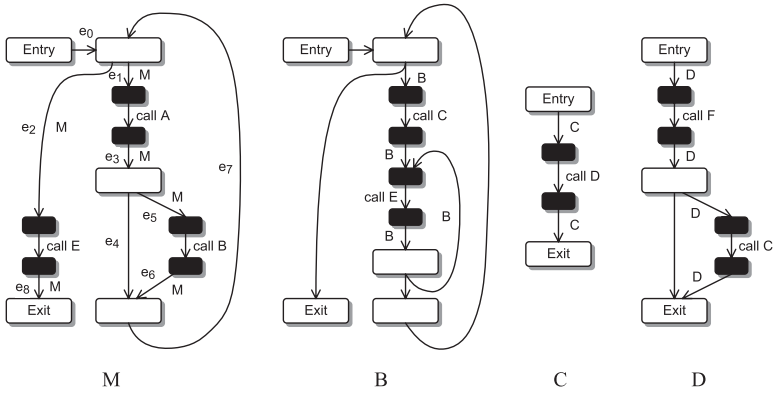


Fig. 4. The CFGs for the path expression.

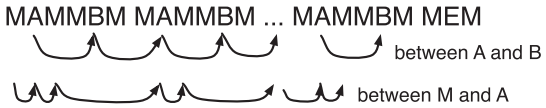


Fig. 5. The worst-case alternating behavior between A and B in M.

an incoming edge to the source of the new edge and each outgoing edge becomes an outgoing edge from the sink of the new edge. The path expression obtained by the resulting CFG summarizes the alternating behavior between functions called by f. In addition, to detect the alternating behavior between the functions called by f and f itself, we annotate each incoming and outgoing edge of a function call node with f.

Figures 2 and 4 shows the original and the transformed CFGs for the four functions, M, B, C, and D. The original CFG for function M in Figure 2 is converted to M in Figure 4. The set of all paths from *Entry* to *Exit* can be summarized by the path expression:

$$e_0 \cdot (\mathsf{M} \cdot \mathsf{A} \cdot \mathsf{M} \cdot ((\mathsf{M} \cdot \mathsf{B} \cdot \mathsf{M})|e_4) \cdot e_7)^* \cdot \mathsf{M} \cdot \mathsf{E} \cdot \mathsf{M}$$

We replace for our purposes irrelevant edges in the regular expression with ϵ and simplify the expression. For M, we obtain the following regular expression which summarizes the alternating behavior between the functions M, A, B, and E:

$$P_{\mathsf{M}} = (\mathsf{M} \cdot \mathsf{A} \cdot \mathsf{M} \cdot ((\mathsf{M} \cdot \mathsf{B} \cdot \mathsf{M})|\epsilon))^* \cdot \mathsf{M} \cdot \mathsf{E} \cdot \mathsf{M}$$

Similarly, path expressions for the CFGs of B, C, and D are given by,

$$\begin{aligned} P_{\mathsf{B}} &= (\mathsf{B} \cdot \mathsf{C} \cdot \mathsf{B} \cdot \mathsf{E} \cdot \mathsf{B} \cdot (\mathsf{B} \cdot \mathsf{E} \cdot \mathsf{B})^*)^*|\epsilon \\ P_{\mathsf{C}} &= \mathsf{C} \cdot \mathsf{D} \cdot \mathsf{C} \\ P_{\mathsf{D}} &= \mathsf{D} \cdot \mathsf{F} \cdot \mathsf{D} \cdot ((\mathsf{D} \cdot \mathsf{C} \cdot \mathsf{D})|\epsilon) \end{aligned}$$

Using the path expression computed from each function's CFG, we can estimate the number of potential conflict misses between two functions. For example, the number of potential conflict misses between A and B can be counted using only P_{M} because M is the root of and also the only caller of both A and B in the static call graph (Figure 1). From the CFG of M in Figure 2, we conclude that the worst case occurs when e_5 is always taken. A and B cause potential conflict misses in every iteration of the loop that encloses them in M. The corresponding path expression for this worst-case path in P_{M} is $(\mathsf{M} \cdot \mathsf{A} \cdot \mathsf{M} \cdot \mathsf{M} \cdot \mathsf{B} \cdot \mathsf{M})^* \cdot \mathsf{M} \cdot \mathsf{E} \cdot \mathsf{M}$, and the number of potential conflict misses between A and B is equivalent to the number of alternations between A and B in the path represented by the path expression (Figure 5), namely twice the number of the

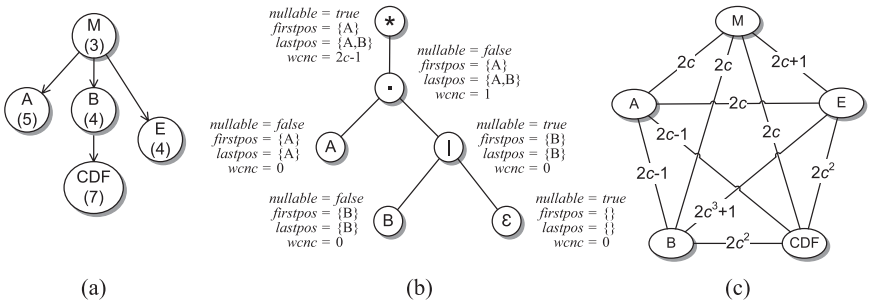


Fig. 6. (a) The call graph forms a DAG after collapsing SCCs. (b) The computation of $WCNC_{\mathsf{A,B}}$. (c) The STRG.

loop iterations minus 1. Similarly, we can compute the worst-case number of potential conflict misses between M and A using P_{M}. It is twice the number of loop iterations.

In general, we need a path expression that represents all paths from the program entry point to the program exit point to estimate the number of potential conflict misses between any two functions x and y. To construct the path expression for the whole program, we start from the path expression of the root function in the static call graph (SCG). We replace calls to ancestors of x and y with the path expressions of their CFGs in a topological sort order or reverse postorder of the SCG. We denote this path expression with $P_{x,y}$. As an example, to compute the number of potential conflict misses between the functions A and E, we replace all calls to B in P_{M} with P_{B} because B is an ancestor of E:

$$P_{\mathsf{A,E}} = (\mathsf{M}\cdot\mathsf{A}\cdot\mathsf{M}\cdot((\mathsf{M}\cdot((\mathsf{B}\cdot\mathsf{C}\cdot\mathsf{B}\cdot\mathsf{E}\cdot\mathsf{B}\cdot(\mathsf{B}\cdot\mathsf{E}\cdot\mathsf{B})^*)^*|\epsilon)\cdot\mathsf{M})|\epsilon))^*\cdot\mathsf{M}\cdot\mathsf{E}\cdot\mathsf{M}.$$

Using this path expression, we can now estimate the number of potential conflict misses between A and E.

These replacement technique does not terminate if there exist a strongly-connected component (SCC) in the SCG (i.e., the nodes C and D in Figure 1(a)). Note that a recursive function also introduces an SCC in the SCG. This replacement procedure is similar to the method of inlining callees into their caller and then computing the path expression from the resulting CFG. However, it is in general impossible to inline all functions in an SCC. Therefore, similar to the traditional code overlaying techniques, we collapse the SCC and its descendants into a single node which will then form a leaf node in the SCG. When two SCCs share the same descendant node, we combine the two SCCs and their descendants together into a single node. After combining the SCCs and their descendants, the SCG becomes a DAG and the collapsed nodes become leaf nodes. The SCG in Figure 1(a) is transformed into the DAG in Figure 6(a). The functions C, D, and F are combined into a single node CDF and form a leaf node in the SCG.

Computing Worst-Case Number of Conflict misses. When constructing the STRG, each edge (x, y) is annotated with the worst-case number of conflict misses between x and y. We denote the worst-case number of conflict misses between x and y during program execution by $WCNC(x, y)$.

To compute $WCNC(x, y)$, we need to consider alternations between x and y only. For this reason, functions other than x and y in $P_{x,y}$ are replaced by ϵ and then the expression is simplified. The resulting path expression is denoted by $P'_{x,y}$. For example, when computing $WCNC(\mathsf{A}, \mathsf{B})$, $P_{\mathsf{A,B}}$ and $P'_{\mathsf{A,B}}$ are given by

$$\begin{aligned} P_{\mathsf{A,B}} &= (\mathsf{M}\cdot\mathsf{A}\cdot\mathsf{M}\cdot((\mathsf{M}\cdot\mathsf{B}\cdot\mathsf{M})|\epsilon))^*\cdot\mathsf{M}\cdot\mathsf{E}\cdot\mathsf{M} \\ P'_{\mathsf{A,B}} &= (\epsilon\cdot\mathsf{A}\cdot\epsilon\cdot((\epsilon\cdot\mathsf{B}\cdot\epsilon)|\epsilon))^*\cdot\epsilon\cdot\epsilon\cdot\epsilon \\ &= (\mathsf{A}\cdot(\mathsf{B}|\epsilon))^*. \end{aligned}$$

Table I. Rules for Computing $WCNC(x, y)$

n	Operations
Leaf	$nullable(n) = \begin{cases} true & \text{if } n = \epsilon \\ false & \text{otherwise} \end{cases}$
	$firstpos(n) = \begin{cases} \{x\} & \text{if } n = x \\ \{y\} & \text{if } n = y \\ \emptyset & \text{otherwise} \end{cases}$
	$lastpos(n) = \begin{cases} \{x\} & \text{if } n = x \\ \{y\} & \text{if } n = y \\ \emptyset & \text{otherwise} \end{cases}$
	$wcnc(n) = 0$
$n_l \mid n_r$	$nullable(n) = nullable(n_l) \vee nullable(n_r)$
	$firstpos(n) = firstpos(n_l) \cup firstpos(n_r)$
	$lastpos(n) = lastpos(n_l) \cup lastpos(n_r)$
	$wcnc(n) = MAX(wcnc(n_l), wcnc(n_r))$
$n_l \cdot n_r$	$nullable(n) = nullable(n_l) \wedge nullable(n_r)$
	$firstpos(n) = \begin{cases} firstpos(n_l) \cup firstpos(n_r) & \text{if } nullable(n_l) \\ firstpos(n_l) & \text{otherwise} \end{cases}$
	$lastpos(n) = \begin{cases} lastpos(n_r) \cup lastpos(n_l) & \text{if } nullable(n_r) \\ lastpos(n_r) & \text{otherwise} \end{cases}$
	$wcnc(n) = \begin{cases} wcnc(n_l) + wcnc(n_r) + 1 & \text{if } (x \in lastpos(n_l) \wedge y \in firstpos(n_r)) \\ & \quad \vee (y \in lastpos(n_l) \wedge x \in firstpos(n_r)) \\ wcnc(n_l) + wcnc(n_r) & \text{otherwise} \end{cases}$
n_l^*	$nullable(n) = true$
	$firstpos(n) = firstpos(n_l)$
	$lastpos(n) = lastpos(n_l)$
	$wcnc(n) = \begin{cases} wcnc(n_l) \times c + c - 1 & \text{if } (x \in lastpos(n_l) \wedge y \in firstpos(n_l)) \\ & \quad \vee (y \in lastpos(n_l) \wedge x \in firstpos(n_l)) \\ wcnc(n_l) \times c & \text{otherwise} \end{cases}$

Next, a syntax tree [Aho et al. 2006] for $P'_{x,y}$ is constructed and for each node four functions are computed. The four functions are labeled *nullable*, $firstpos$, *lastpos*, and *wcnc*. We denote the subexpression of $P'_{x,y}$ that is rooted at a node n in the syntax tree by $sub(n)$. The definitions of the function are given as,

(1) $nullable(n)$ is true for a syntax tree node n if and only if $\epsilon \in \sigma(sub(n))$, that is, there exists a path in $\sigma(sub(n))$ that does neither invoke x nor y.
(2) $firstpos(n)$ is the set of functions that can appear at the beginning of a path in $\sigma(sub(n))$. $firstpos(n) \subseteq \{x, y\}$.
(3) $lastpos(n)$ is the set of functions that can appear at the end of a path in $\sigma(sub(n))$. $lastpos(n) \subseteq \{x, y\}$.
(4) $wcnc(n)$ is the number of potential conflict misses between x and y for the worst-case path in $\sigma(sub(n))$.

The syntax tree of $P'_{x,y}$ is traversed in postorder and for each node the operations defined in Table I are applied to compute $WCNC(x, y)$ for $P'_{x,y}$. In Table I, n_l and n_r denote the left child and the right child of n, respectively. Without profiling information, no information on the number of loop iterations is available, hence we assume that each loop in the program has the same constant number of iterations, c (≥ 1). $WCNC(x, y)$ for $P'_{x,y}$ is equal to $wcnc(r)$ where r denotes the root of the syntax tree.

THEOREM 4.1. *The number of potential conflict misses between two different functions x and y for each path in $\sigma(sub(n))$ is less than or equal to $wcnc(n)$ under the assumption that each loop in the program has the same constant number of iterations, c (≥ 1).*

PROOF. We denote the number of potential conflict misses in a path p with $nc(p)$. We show that $nc(p) \leq wcnc(n)$ for any path $p \in \sigma(sub(n))$. We prove it using structural induction.

To complete the basis step, we show that $nc(p)$ is zero when a syntax tree that consists of a single node n. Since n is an empty string or a function, it is trivial that there is no potential conflict miss between two functions in any path $p \in \sigma(sub(n))$. Thus, $nc(p) = 0$.

For the inductive hypothesis, we assume that $nc(p_l) \leq wcnc(n_l)$ and $nc(p_r) \leq wcnc(n_r)$ for any path $p_l \in \sigma(sub(n_l))$ and $p_r \in \sigma(sub(n_r))$, respectively.

—For a syntax tree that is rooted at a node $n = n_l|n_r$, $p \in \sigma(sub(n_l|n_r))$ implies $p \in \sigma(sub(n_l))$ or $p \in \sigma(sub(n_r))$. We have $wcnc(n_l) \leq wcnc(n_l|n_r) \wedge wcnc(n_r) \leq wcnc(n_l|n_r)$ by the $wcnc(n_l|n_r)$ rule in Table I. Using the inductive hypothesis, we obtain $nc(p) \leq wcnc(n_l|n_r)$.

—For a syntax tree that is rooted at a node $n = n_l \cdot n_r$, $p \in \sigma(sub(n_l \cdot n_r))$ implies $p = p_l \cdot p_r$, where $p_l \in \sigma(sub(n_l))$ and $p_r \in \sigma(sub(n_r))$.
When $x \in lastpos(n_l) \wedge y \in firstpos(n_r)$ or $y \in lastpos(n_l) \wedge x \in firstpos(n_r)$, $nc(p) = nc(p_l \cdot p_r) \leq nc(p_l)+nc(p_r)+1$. In addition, we have $wcnc(n_l \cdot n_r) = wcnc(n_l)+wcnc(n_r)+1$ by the $wcnc(n_l \cdot n_r)$ rule in Table I. Using the inductive hypothesis, we obtain $nc(p) \leq wcnc(n_l \cdot n_r)$.
Otherwise, $nc(p) = nc(p_l \cdot p_r) = nc(p_l) + nc(p_r)$. In addition, we have $wcnc(n_l \cdot n_r) = wcnc(n_l)+wcnc(n_r)$ by the $wcnc(n_l \cdot n_r)$ rule in Table I. Using the inductive hypothesis, we obtain $nc(p) \leq wcnc(n_l \cdot n_r)$.

—For a syntax tree that is rooted at a node $n = n_l^*$, $p \in \sigma(sub(n_l^*))$ is $p_1 \cdot p_2 \cdot \cdots \cdot p_c$ where $p_i \in \sigma(sub(n_l))$ and $1 \leq i \leq c$ because we assume that each loop in the program iterates $c \geq 1$ times. By the inductive hypothesis, we have $nc(p_i) \leq wcnc(n_l)$ where $1 \leq i \leq c$.
When $x \in lastpos(n_l) \wedge y \in firstpos(n_l)$ or $y \in lastpos(n_l) \wedge x \in firstpos(n_l)$, there are at most $c-1$ potential conflict misses in p. Thus, we have $nc(p_1 \cdot p_2 \cdot \cdots \cdot p_c) \leq nc(p_1) + nc(p_2) + \cdots + nc(p_c) + c - 1$. In addition, $wcnc(n_l^*) = wcnc(n_l) \times c + c - 1$ by the $wcnc(n_l^*)$ rule in Table I. Thus,

$$\begin{aligned} nc(p_1 \cdot p_2 \cdot \cdots \cdot p_c) &\leq nc(p_1) + nc(p_2) + \cdots + nc(p_c) + c - 1 \\ &\leq wcnc(n_l) \times c + c - 1 \\ &= wcnc(n_l^*) \end{aligned}$$

Otherwise, $nc(p_1 \cdot p_2 \cdot \cdots \cdot p_c) = nc(p_1) + nc(p_2) + \cdots + nc(p_c)$. In addition, we have $wcnc(n_l^*) = wcnc(n_l) \times c$ by the $wcnc(n_l^*)$ rule in Table I. Thus,

$$\begin{aligned} nc(p_1 \cdot p_2 \cdot \cdots \cdot p_c) &= nc(p_1) + nc(p_2) + \cdots + nc(p_c) \\ &\leq wcnc(n_l) \times c \\ &= wcnc(n_l^*) \end{aligned}$$

□

Figure 6(b) shows an example that computes $WCNC(\mathsf{A}, \mathsf{B})$ for $P'_{\mathsf{A},\mathsf{B}} = (\mathsf{A} \cdot (\mathsf{B}|\epsilon))^*$. After computing $WCNC(x, y)$ for each pair (x, y) of nodes in the static call graph, we construct the static TRG by annotating each edge (x, y) with $WCNC(x, y)$. Figure 6(c) shows the corresponding annotated STRG.

4.2. Assigning Memory Regions

Based on the STRG, the code overlay structure for a given higher-level memory size is computed. A greedy algorithm repeatedly selects the heaviest edge in the STRG and merges the two connected nodes into one until the STRG has no more remaining edges. Whenever two nodes are merged, all functions in the merged node that have not yet been placed in the higher-level memory are assigned a memory region.

Table II. Rules for Computing $WCNI(n)$

n	Operations
Leaf	$wcni(n) = \begin{cases} 1 & \text{if } n = x \\ 0 & \text{otherwise} \end{cases}$
$(n_l \mid n_r)$	$wcni(n) = MAX(wcni(n_l), wcni(n_r))$
$(n_l \cdot n_r)$	$wcni(n) = wcni(n_l) + wcni(n_r)$
(n_l^*)	$wcni(n) = wcni(n_l) \times c$

The node merging algorithm considers not only the number of potential conflict misses between two functions but also the number of runtime checks introduced by the overlay manager. However, the overlay manager has to check whether the target function resides in the higher-level memory at every invocation of the function if that function shares a memory region with any other function. The number of runtime checks is proportional to the number of invocations of the function.

To estimate the number of invocations of a function x, we propose an algorithm that computes the worst-case number of invocations (WCNI) of a function using path expressions. The algorithm is similar to one that computes the WCNC. Starting from the path expression of the root function in the static call graph, we replace all calls to ancestors of x with the path expressions of their CFGs in a topological sort order or reverse postorder of the SCG. We denote such a path expression with P_x. After replacing all functions other than x in P_x with ϵ and simplifying the expression, we obtain P'_x. For example, P_{E} and P'_{E} are given by

$$\begin{aligned} P_{\mathsf{E}} &= (\mathsf{M}\cdot\mathsf{A}\cdot\mathsf{M}\cdot((\mathsf{M}\cdot((\mathsf{B}\cdot\mathsf{CDF}\cdot\mathsf{B}\cdot\mathsf{E}\cdot\mathsf{B}\cdot(\mathsf{B}\cdot\mathsf{E}\cdot\mathsf{B})^*)^*|\epsilon)\cdot\mathsf{M})|\epsilon))^*\cdot\mathsf{M}\cdot\mathsf{E}\cdot\mathsf{M} \\ P'_{\mathsf{E}} &= (\epsilon\cdot\epsilon\cdot\epsilon\cdot((\epsilon\cdot((\epsilon\cdot\epsilon\cdot\epsilon\cdot\mathsf{E}\cdot\epsilon\cdot(\epsilon\cdot\mathsf{E}\cdot\epsilon)^*)^*|\epsilon)\cdot\epsilon)|\epsilon))^*\cdot\epsilon\cdot\mathsf{E}\cdot\epsilon \\ &= (((\mathsf{E}\cdot\mathsf{E}^*)^*|\epsilon)|\epsilon)^*\cdot\mathsf{E} \end{aligned}$$

Similarly, P'_{A}, P'_{B}, and P'_{CDF} are given by

$$\begin{aligned} P'_{\mathsf{A}} &= (\mathsf{M}\cdot\mathsf{A}\cdot\mathsf{M}\cdot((\mathsf{M}\cdot\mathsf{B}\cdot\mathsf{M})|\epsilon))^*\cdot\mathsf{M}\cdot\mathsf{E}\cdot\mathsf{M} \\ &= (\epsilon\cdot\mathsf{A}\cdot\epsilon\cdot((\epsilon\cdot\epsilon\cdot\epsilon)|\epsilon))^*\cdot\epsilon\cdot\epsilon\cdot\epsilon \\ &= (A\cdot(\epsilon|\epsilon))^* \\ P'_{\mathsf{B}} &= (\mathsf{M}\cdot\mathsf{A}\cdot\mathsf{M}\cdot((\mathsf{M}\cdot\mathsf{B}\cdot\mathsf{M})|\epsilon))^*\cdot\mathsf{M}\cdot\mathsf{E}\cdot\mathsf{M} \\ &= (\epsilon\cdot\epsilon\cdot\epsilon\cdot((\epsilon\cdot\mathsf{B}\cdot\epsilon)|\epsilon))^*\cdot\epsilon\cdot\epsilon\cdot\epsilon \\ &= (\mathsf{B}|\epsilon)^* \\ P'_{\mathsf{CDF}} &= (\mathsf{M}\cdot\mathsf{A}\cdot\mathsf{M}\cdot((\mathsf{M}\cdot((\mathsf{B}\cdot\mathsf{CDF}\cdot\mathsf{B}\cdot\mathsf{E}\cdot\mathsf{B}\cdot(\mathsf{B}\cdot\mathsf{E}\cdot\mathsf{B})^*)^*|\epsilon)\cdot\mathsf{M})|\epsilon))^*\cdot\mathsf{M}\cdot\mathsf{E}\cdot\mathsf{M} \\ &= (\epsilon\cdot\epsilon\cdot\epsilon\cdot((\epsilon\cdot((\epsilon\cdot\mathsf{CDF}\cdot\epsilon\cdot\epsilon\cdot\epsilon\cdot(\epsilon\cdot\epsilon\cdot\epsilon)^*)^*|\epsilon)\cdot\epsilon)|\epsilon))^*\cdot\epsilon\cdot\epsilon\cdot\epsilon \\ &= ((\mathsf{CDF}^*|\epsilon)|\epsilon)^* \end{aligned}$$

We denote the WCNI of a function x by $WCNI(x)$. The function $wcni(n)$ is defined by the number of invocations of a function x for the worst-case path in $\sigma(sub(n))$. We compute $wcni(n)$ for each node n in the syntax tree of P'_x by traversing the syntax tree in postorder and performing the operations defined in Table II. $WCNI(x)$ for P'_x is equal to $wcni(r)$, where r is the root of the syntax tree. The $WCNI$ for each function in Figure 6(a) is computed as $WCNI(\mathsf{M}) = 1$, $WCNI(\mathsf{A}) = WCNI(\mathsf{B}) = c$, $WCNI(\mathsf{CDF}) = c^2$, and $WCNI(\mathsf{E}) = c^3 + c^2 + 1$

We describe the placement algorithm with an example. Figure 7 illustrates each step of the algorithm for a given memory size of 18, which is the size of the generated overlay structure by the traditional algorithm in Figure 1(d). The heaviest edge in Figure 6(c) is (B, E), hence nodes B and E are merged first and allocated to a region in the higher-level memory. Figure 7(a) shows the STRG and the overlay structure after merging B and E. The edges (x, B) and (x, E) with $x \cap \{B, E\} = \phi$ are combined together into a single edge with the weight equal to the sum of the weights of the single edges. The edge (B, E) is removed from the STRG. In the STRG in Figure 7(a), the heaviest

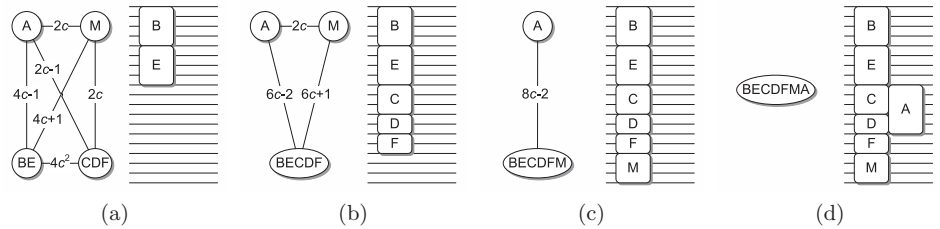


Fig. 7. Placing each function in the higher-level memory using the STRG.

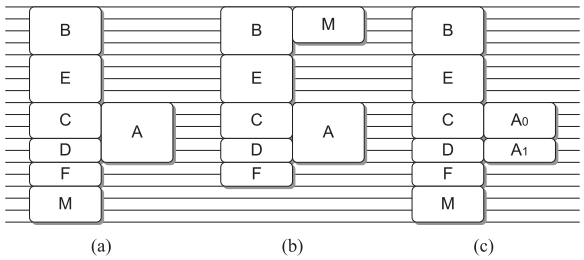


Fig. 8. (a) The final overlay structure. (b) The final overlay structure by our algorithm with a memory size of 15. (c) The slices of overlay structure (a).

edge is now (BE, CDF). Thus, BE and CDF are merged into BECDF, then C, D, and F are placed in the next available consecutive regions in the higher-level memory. The result of this round is shown in Figure 7(b). Next, M and BECDF are merged, and M is placed in memory (Figure 7(c)).

When the algorithm merges BECDFM and A in Figure 7(c), the remaining available memory is too small to contain A. In that case, the algorithm tries to place A starting at offset 0 to S_{mem} - size of A by incrementing the offset by S_{inc} at a time, where S_{mem} is the size of available higher-level memory. S_{inc} should follow the address alignment requirement.

—S_{inc} must be a multiple of the minimum instruction address alignment requirement of the target architecture.
—When overlaid functions are loaded into the higher-level memory by exploiting a block transfer functionality (e.g., DMA), S_{inc} must be a multiple of the address alignment requirement of the block transfer functionality.

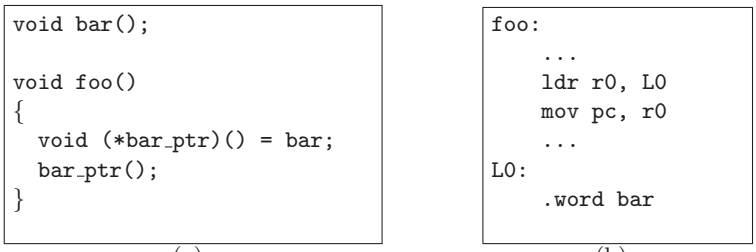
The offset with the minimum total cost for overlaying A is chosen as A's location (as follows). We obtain an overlay structure as shown in Figure 8(a).

When placing a function x in higher-level memory, the total cost considered is composed of two parts: the cost of loading code segments, $C_{loading}$, and the cost of the runtime checks, C_{checks}.

$$C_{loading} = \sum_{y \in F} S_{xy} \times WCNC(x, y)$$

$$C_{checks} = \begin{cases} 0 & \text{if } F = \emptyset \\ WCNI(x) + \sum_{y \in G} WCNI(y) & \text{otherwise.} \end{cases}$$

F denotes the set of functions that share the memory region with x, and $x \notin F$. $G \subseteq F$ denotes the set of functions that share the memory region only with x. S_{xy} is the size of the overlap between x and $y \in F$. Note that a code segment y, such that $y \in F$ and



```
void bar();

void foo()
{
  void (*bar_ptr)() = bar;
  bar_ptr();
}
```

(a)

```
foo:
    ...
    ldr r0, L0
    mov pc, r0
    ...
L0:
    .word bar
```

(b)

Fig. 9. Identifying exposed functions. (a) A sample source code that invokes a function `bar` with a function pointer. (b) an assembly code of the source shown in (a).

$y \notin G$, has an overlap with another code segment z ($\neq x$) in F. Thus, y requires the runtime check cost regardless of x. On the other hand, the runtime check cost for a function $y \in G$ is introduced by placing x because y does not require the runtime check unless x is placed in the same location. The total cost C_{total} is given by

$$C_{total} = \alpha \cdot C_{loading} + \beta \cdot C_{checks.}$$

The weights α and β can be determined empirically.

The overlay structure generated by our algorithm for the memory size of 15 is shown in Figure 8(b). Unlike traditional techniques, our algorithm can generate a legal overlay structure for any given memory size as long as its size is bigger than the maximum code segment size (including the SCCs) in the program.

As discussed in Section 3, to reduce the redundant memory copying overhead each code segment in the overlay structure is divided into function slices. A function slice does not have a fixed size. We say that an address a is *contained* in a code segment x if a is greater than the start address of x and a is less than the end address of x. A code segment x is divided into slices at every address a contained in x if a is the start or end address of other code segment $y \in F$, where F is the set of code segments that share the memory region with x. A slice x_i represents the i-th slice of x. Figure 8(c) shows the result of slicing the overlay structure in Figure 8(a).

4.3. Indirect Function Calls

If a function is invoked through a pointer, the input object files to the linker must contain relocation information that refers to the absolute address of the function. In other words, the absolute address of the function is determined after linking. Figure 9(a) shows a sample source code that invokes a function `bar` with a function pointer (`bar_ptr`). The assembly code generated by a compiler for the ARM architecture is shown in Figure 9(b). The symbol `L0` points to the memory location whose value cannot be determined at compile time (i.e., in the object generated by the compiler), and the symbol `bar` has the relocation information of an absolute address. The relocation information indicates that it will have an absolute address after linking. The linker determines the value of `bar` after resolving all symbols in the program. By analyzing the relocation information of all input objects including libraries, we can conservatively identify such function symbols that have the absolute-address relocation information. We call a function whose absolute address need to be exposed by the linker as an *exposed function*. An exposed function *may* be invoked through a pointer.

Conservatively, an indirectly called function can be any exposed function. Thus, inserting conservative edges in the STRG for every exposed function will likely results in too many edges in the STRG. For this reason, we do not consider indirect function

calls in the STRG construction. Instead, the glue code and overlay manager handles the indirect function calls. This is elaborated in the next section.

5. GLUE CODE INSERTION

In this section, we present how to insert glue code into the program for the overlay structure at link time and several optimization techniques for the glue code. Unlike traditional code overlaying techniques, our technique allows the caller and callee to share the same memory space. Moreover, it can also place functions invoked through function pointers in the overlaid memory region. Therefore, our technique translates each function call and return to glue code that forwards control to the overlay manager.

The overlay manager is a small code fragment that is resident in the higher-level memory and consists of two routines: `Load()` and `Load&Branch()`. `Load()` checks whether the target code segment resides in the higher-level memory. If not, it loads the segment into the higher-level memory according to the overlay structure. Note that the unit of loading is a *function slice* that is mentioned in the previous section. `Load&Branch()` is similar to `Load()`, but it forwards control to the target after loading.

5.1. Glue Code and the Overlay Manager

Depending on the relationship between the caller and the callee, we insert four different cases of glue code.

—The callee is not overlaid (i.e., it does not share its memory space with other code segments). In this case no glue code is inserted, the function call is left as-is.
—The callee is called through a function pointer. The glue code invokes `Load&Branch()` with the target address. The overlay manager searches the set of exposed functions for the target address and checks if the target segment resides in the higher-level memory.
—The caller and callee do not overlap in the overlay structure. The glue code invokes `Load()`. After returning from `Load()`, the remaining part of the glue code forwards control to the target. Since the caller and callee do not overlap, we can place the glue code that invokes `Load()` as early as possible to hide target loading time and to eliminate redundant calls to `Load()`.
—The caller and callee overlap in the overlay structure. The glue code invokes `Load&Branch()`. Since the callee overwrites the caller, the overlay manager must forward the control to the target after loading. An example is the case when M calls B under the overlay structure in Figure 8(b).

In addition, the glue code modifies the return address appropriately depending on the type of the callee and passes it to the callee. The return address will be used later by the glue code inserted at the return site of the callee.

When a function returns, it must be guaranteed that the target of the return resides in the higher-level memory. We consider two different cases in the glue code at the return site depending on the type of the returning function. We assume that if a function contains a function call through a pointer, its descendants in the SCG include all the exposed functions in the program.

—When any of the returning function's predecessors and any of the returning function's descendants in the SCG do not overlap, we do not insert glue code at the return site. In this case, the caller (i.e., one of the returning function's predecessors) is guaranteed to be located in the higher-level memory.
—Otherwise, at least one of the returning function's predecessors and one of the returning function's descendants in the SCG are overlapping in the overlay structure.

In this case, glue code is inserted at the return site of returning function. Depending on the type of the caller, the glue code may not invoke any overlay manager routine:

—When the caller (i.e., one of the returning function's predecessors) and any of the returning function's descendants in the SCG do not overlap in the overlay structure, the caller passes an unmodified return address to the callee. The caller is guaranteed to reside in the higher-level memory when the callee returns.

—When the caller and one of the returning function's descendants in the SCG do overlap, the caller passes a modified return address to the callee.

The glue code at the return site checks if the return address is modified or not. If it is modified, the glue code invokes `Load&Branch()`. Otherwise, the glue code just forwards control to the target in the higher-level memory. To distinguish a modified return address from an unmodified one, the least significant bit of the modified address is set to 1. For example, consider the overlay structure in Figure 8(b). When `M` calls `B`, `M` (the caller) modifies the return address and passes it to `B` (the callee) because `M` and `B` overlap.

5.2. Intraprocedural Redundancy Elimination

In this section, we present a technique to eliminate redundant calls to `Load()` and schedule those calls as early as possible to hide load latencies. This method is especially effective when we use DMA operations to load a code segment. Our technique is based on the partial redundancy elimination technique [Knoop et al. 1992].

The unit of loading in our overlaying technique is a function slice. A function slice has the property that it either entirely overlaps or does not overlap at all with another function slice in the overlay structure. This enables us to denote a slice as a bit in a bit vector. We denote with S_f the set of all slices that belong to a function f. Two different slices *interfere* with each other if and only if they overlap in the overlay structure. A function f and a function slice s interfere with each other if and only if S_f contains a slice that interferes with s.

A function is represented by its CFG. The sets of all predecessors and successors of a node n in the CFG are denoted by $\mathsf{pred}[n]$ and $\mathsf{succ}[n]$, respectively. Without loss of generality, we assume that every block in the CFG lies on some path from *Entry* to *Exit*. We are interested in those program points that are either the entry or the exit of a basic block.

Availability and anticipatability. We define the *availability* and *anticipatability* of a slice in the higher-level memory. The definitions are similar to the availability and anticipatability of expressions [Knoop et al. 1992].

A basic block B in a function f *uses* a slice s of a function g if $f = g$ or B contains a call to g. A *use* of a slice is the basic block that uses the slice. A basic block *kills* a slice s if it uses a slice that interferes with s or it calls a function f that reaches another function g (possibly $f = g$) that interferes with s in the SCG (when the basic block calls a function through a function pointer, we consider functions that interfere with s and are reachable from any exposed function). A slice s is *partially/fully available* in the higher-level memory at a program point p if along *some/each* path reaching p from `Entry`, there is a use of s and there is no basic block that kills s. A slice s is *partially/fully anticipatable* in the higher-level memory at a program point p if along *some/each* path leading from p to exit, there is a use of s and there is no basic block that kills s from p to that use.

Using these definitions of availability and anticipatability of a slice, we define the redundancy of fetches (i.e., `Load()`). A fetch for a slice s at program point p is *partially/fully redundant* if s is *partially/fully available* at p. For example, the fetch for s_A at the entry of block B_5 in Figure 10(a) is partially redundant. As shown in Figure 10(b), it can be moved up to the entry of block B_3 (i.e., as early as possible) to

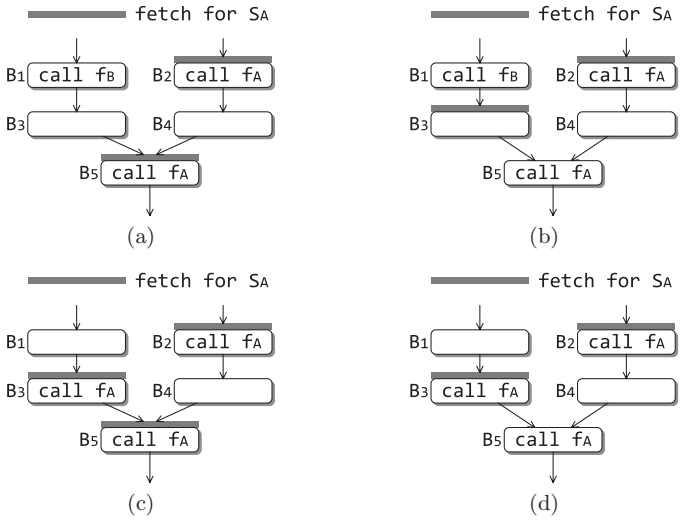


Fig. 10. Assume that s_B overlaps with s_A in the overlay structure. (a) and (b) are before and after eliminating partial redundancy of fetches. (c) and (d) are before and after eliminating full redundancy of fetches.

minimize the number of fetches for s_A. It can not be moved further upward because s_B interferes with s_A in the higher-level memory. In the example shown in Figure 10(c), the fetch at the entry of B_5 is fully redundant, and we can remove it (Figure 10(d)).

We insert a new synthetic basic block between the source and the destination of an edge if either the destination has more than one predecessor or the source has more than one successor. This edge splitting transformation eliminates all *critical edges* in the CFG. A *critical edge* is an edge leading from a node with more than one successor to a node with more than one predecessor. In addition, it simplifies our data-flow analysis [Knoop et al. 1992].

Fetch placement. Anticipation of a slice in the higher-level memory limits how early a slice can be fetched into the higher-level memory. When a fetch for slice s is placed, it can not be scheduled at a program point where s is not anticipated. To solve the anticipated-slices problem with a data-flow analysis, we abstract the semantics of each block B for a slice s with two Boolean properties:

$$\mathsf{USE}[B] = \begin{cases} \mathsf{true} & \text{when } B \text{ uses } s \\ \mathsf{false} & \text{otherwise} \end{cases}$$

$$\mathsf{KILL}[B] = \begin{cases} \mathsf{true} & \text{when } B \text{ kills } s \\ \mathsf{false} & \text{otherwise.} \end{cases}$$

We use $\mathsf{ANTin}[B]$ and $\mathsf{ANTout}[B]$ to denote the anticipatability of s at the entry and the exit of B, respectively. A slice s is anticipatable on entry to B if it is used in B or if it is anticipatable on exit from B and is not killed in B. A slice s is anticipatable on exit from B if it is anticipatable on entry to each successor of B:

$$\mathsf{ANTin}[B] = \mathsf{USE}[B] \vee (\mathsf{ANTout}[B] \wedge \neg \mathsf{KILL}[B])$$

$$\mathsf{ANTout}[B] = \begin{cases} false & \text{if } B = Exit \\ \bigwedge_{s \in \mathsf{succ}[B]} \mathsf{ANTin}[s] & \text{otherwise.} \end{cases}$$

To compute places for the earliest fetches, we define the notion of *will-be-available* for a slice. When a slice s is anticipatable but not available at program point p, it could be

made available after p if we chose to insert a fetch for s at p. We use WBAVAILin[B] and WBAVAILout[B] to denote the availability of s at the entry and the exit of B, respectively. A slice s is will-be-available on exit from block B if it is anticipatable or will-be-available on entry to B and is not killed in B. A slice s is will-be-available on entry to B if it is will-be-available on exit from each predecessor of B:

$$\mathsf{WBAVAILout}[B] = (\mathsf{ANTin}[B] \vee \mathsf{WBAVAILin}[B]) \wedge \neg\mathsf{KILL}[B]$$

$$\mathsf{WBAVAILin}[B] = \begin{cases} false & \text{if } B = Entry \\ \bigwedge_{P \in \mathsf{pred}[B]} \mathsf{WBAVAILout}[P] & \text{otherwise.} \end{cases}$$

We place a fetch for a slice s where s is anticipated earliest along some path. The earlier the fetch is placed, the more redundant fetches for s can be removed. A fetch for a slice s is earliest on entry to block B if s is anticipated on entry to B but not will-be-available:

$$\mathsf{EARLIESTin}[B] = \mathsf{ANTin}[B] \wedge \neg\mathsf{WBAVAILin}[B].$$

We insert a fetch for a slice s on entry to B if EARLIESTin[B] = true.

Wait&check placement. When we use a DMA operation to fetch a slice, before transferring control to the fetched code, we have to wait and check (i.e., busy waiting) if the DMA operation has been completed. It is important to schedule the wait&check as late as possible to avoid performance degradation due to the checking overhead.

To find the latest wait&check point for each slice, we define the *postponability* of a slice in the higher-level memory, which is similar to the postponability of expressions [Knoop et al. 1992]. A slice s is *postponable* at a program point p if there is a fetch for s along every path from the entry to p and there is no subsequent use of s after the last fetch for s. Postponability of a slice s limits how late a fetch for s can be scheduled. A wait&check for a slice s should be scheduled at a program point where s is earliest or postponable.

We use POSTin[B] and POSTout[B] to denote the postponability of s at the entry and the exit of B, respectively. A slice s is postponable on exit from block B if it is earliest or postponable on entry to B but not used in B. A slice s is postponable on entry to B if it is postponable on exit from each predecessor of B:

$$\mathsf{POSTout}[B] = (\mathsf{EARLIESTin}[B] \vee \mathsf{POSTin}[B]) \wedge \neg\mathsf{USE}[B]$$

$$\mathsf{POSTin}[B] = \begin{cases} false & \text{if } B = Entry \\ \bigwedge_{P \in \mathsf{pred}[B]} \mathsf{POSTout}[P] & \text{otherwise.} \end{cases}$$

A wait&check for a slice s is latest on entry to block B if it is earliest or postponable on entry to B and it is used in B or not earliest and not postponable on entry to some successor of B:

$$\begin{aligned} \mathsf{LATESTin}[B] = \; & (\mathsf{EARLIESTin}[B] \vee \mathsf{POSTin}[B]) \wedge \\ & (\mathsf{USE}[B] \vee \neg(\textstyle\bigwedge_{S \in \mathsf{succ}[B]} (\mathsf{EARLIESTin}[S] \vee \mathsf{POSTin}[S]))). \end{aligned}$$

We insert a wait&check for a slice s on entry to B if LATESTin[B] = true.

Loop optimization. The original partial redundancy elimination algorithm [Knoop et al. 1992] does not move loop invariants in a `while`-like loop to the outside of the loop. Moving them outside of the `while`-like loop is unsafe for expressions because it may slowdown the program when the loop is not executed and creates unexpected exceptions[Bodík et al. 1998]. However, since we are dealing with fetching of code segments, there is no need to worry about the exceptions generated by expressions. We analyze a set of *loop-invariant* slices for each natural loop in the CFG. If a slice s is used in a block in the loop but not killed in the loop, we say that the slice s is *loop-invariant* for the loop. If a slice s is loop-invariant, we move the fetch for s to the loop preheader [Aho et al. 2006].

5.3. Interprocedural Redundancy Elimination

In this section, we present an interprocedural data-flow analysis algorithm for eliminating redundant fetches. It uses the result of intraprocedural redundancy elimination. When a call site calls a function through a function pointer, we conservatively assume that it calls all the exposed functions.

Let C_f be the set of all basic blocks that contain a function call to f. When a block B contains a function call to f, a slice s is anticipatable on entry to B if it is anticipatable on entry to f. A slice s is anticipatable on exit from f if it is anticipatable on exit from all blocks that contain a call to f:

$$\mathsf{ANTup}[B] = \begin{cases} \mathsf{ANTin}[Entry_f] & \text{if } B \text{ calls } f \\ false & \text{otherwise} \end{cases}$$

$$\mathsf{ANTdown}[B] = \begin{cases} \bigwedge_{X \in C_f} \mathsf{ANTout}[X] & \text{if } B = Exit \text{ of } f \\ false & \text{otherwise} \end{cases}$$

$$\mathsf{ANTin}[B] = \mathsf{USE}[B] \vee \mathsf{ANTup}[B] \vee (\mathsf{ANTout}[B] \wedge \neg(\mathsf{KILL}[B]))$$

$$\mathsf{ANTout}[B] = \begin{cases} \mathsf{ANTdown}[B] & \text{if } B = Exit \\ \bigwedge_{s \in \mathsf{succ}[B]} \mathsf{ANTin}[s] & \text{otherwise.} \end{cases}$$

Our interprocedural algorithm performs the intraprocedural data-flow analysis for each function in the SCG to compute ANTin and ANTout, and then computes ANTup and ANTdown until there is no change in ANTup and ANTdown.

Similarly, when a block B contains a function call to f, a slice s is will-be-available on exit from B if it is will-be-available on exit from f. A slice s is will-be-available on entry to f if it is will-be-available on entry to all blocks that contain a call to f:

$$\mathsf{WBAVAILup}[B] = \begin{cases} \mathsf{WBAVAILout}[Exit_f] & \text{if } B \text{ calls } f \\ false & \text{otherwise} \end{cases}$$

$$\mathsf{WBAVAILdown}[B] = \begin{cases} \bigwedge_{X \in C_f} \mathsf{WBAVAILin}[X] & \text{if } B = Entry \text{ of } f \\ false & \text{otherwise} \end{cases}$$

$$\mathsf{WBAVAILout}[B] = \mathsf{WBAVAILup}[B] \vee ((\mathsf{ANTin}[B] \vee \mathsf{WBAVAILin}[B]) \wedge \neg(\mathsf{KILL}[B]))$$

$$\mathsf{WBAVAILin}[B] = \begin{cases} \mathsf{WBAVAILdown}[B] & \text{if } B = Entry \\ \bigwedge_{P \in \mathsf{pred}[B]} \mathsf{WBAVAILout}[P] & \text{otherwise.} \end{cases}$$

Again, our interprocedural algorithm performs the intraprocedural data-flow analysis for each function in the SCG to compute WBAVAILin and WBAVAILout, and then computes WBAVAILup and WBAVAILdown until there is no change.

When a block B contains a function call to f, a slice s is postponable on exit from B if it is postponable on exit from f. A slice s is postponable on entry to f if it is earliest or postponable on entry to but not used in all blocks that contain a call to f:

$$\mathsf{POSTup}[B] = \begin{cases} \mathsf{POSTout}[Exit_f] & \text{if } B \text{ calls } f \\ false & \text{otherwise} \end{cases}$$

$$\mathsf{POSTdown}[B] = \begin{cases} \bigwedge_{X \in C_f} ((\mathsf{EARLIESTin}[X] \vee \mathsf{POSTin}[X]) \wedge \neg\mathsf{USE}[X]) \\ \qquad \text{if } B = Entry \text{ of } f \\ false \qquad \text{otherwise} \end{cases}$$

ALGORITHM 1: The STRG construction algorithm

Input: SCG : Static call graph, `m` : the entry function.
Output: $WCNI$: the worst-case number of invocations, $WCNC$: the worst-case number of potential conflict misses

```
begin
    order := topological sort order of SCG;
    visit := ∅;
    P := m;
    while order ≠ ∅ do
        x := pop_front(order) ;
        WCNI[x] := compute_wcni(P, x) ;                          /* compute WCNI(x) */
        foreach y ∈ visit do
            WCNC[x][y] := compute_wcnc(P, x, y) ;                /* compute WCNC(x, y) */
        end
        if SUCC[x] ≠ ∅ then
            P_x := compute_path_exp(x) ;          /* compute the path expression P_x of x */
            replace(P, x, P_x) ;                        /* replace every x in P with P_x */
        end
        visit := visit ∪ {x};
    end
end
```

$$\mathsf{POSTout}[B] = \mathsf{POSTup}[B] \vee ((\mathsf{EARLIESTin}[B] \vee \mathsf{POSTin}[B]) \wedge \neg \mathsf{USE}[B])$$

$$\mathsf{POSTin}[B] = \begin{cases} \mathsf{POSTdown}[B] & \text{if } B = \mathit{Entry} \\ \bigwedge_{P \in \mathsf{pred}[B]} \mathsf{POSTout}[P] & \text{otherwise.} \end{cases}$$

Our algorithm performs the intraprocedural data-flow analysis for each function in the SCG to compute POSTin and POSTout, and then computes POSTup and POSTdown until there is no change.

6. DETAILED ALGORITHMS AND COMPLEXITY ANALYSES

In this section, we present the detailed algorithm and time complexity of each step in our approach. Our approach consists of the following three steps:

—STGR construction,
—code segment placement,
—redundant code fetches elimination.

6.1. STRG Construction

Algorithm 1 shows the detailed algorithm of STRG construction phase. Since the SCG is transformed into the DAG in the way as described in Section 4, a topological sort order of the DAG can be obtained. A path expression is represented as a syntax tree as shown in Figure 6(b). The path expression P of the whole program is initially assigned to a path expression `m` that represents a function call to the entry function (i.e., `main()`) of the program. A set of visited functions `visit` is initialized as an empty set. $SUCC[x]$ denotes the set of successors of a function x in the DAG.

The outermost loop visits each function x in the DAG in the topological sort order. P_x, the path expression for a function x, is computed by the function `compute_path_exp(`x`)` using Tarjan's algorithm [1981a, 1981b]. Every x in P is replaced with P_x using the function `replace(`P, x, P_x`)` after computing $WCNI(x)$ and $WCNC(x, y)$ for each $y \in$ `visit`.

If the CFG, G, of a function is a reducible flow graph, Tarjan's algorithm takes $O(m \cdot \alpha(m, n))$ time to obtain the path expression of the function, where m and n are the number of edges and nodes in G, respectively. α is an inverse of Ackermann's function. We replace irrelevant edges in P_x with ϵ and simplify P_x in the way that has been discussed in Section 4. This implies that the number of leaf nodes in the syntax tree of P_x is bounded by the number of function calls in x.

Since we follow a topological sort order, all ancestors of a function x have been visited when x is visited. This implies that every ancestor y of x has been replaced with P_y in P, and every call site of x is exposed in P. Consequently, we can compute $WCNI(x)$ and $WCNC(x, y)$ for each $y \in$ `visit` when x is visited.

The function `compute_wcnc(`P, x, y`)` computes $WCNC(x, y)$ using the rules in Table I and the function `compute_wcni(`P, x`)` computes $WCNI(x)$ using the rules in Table II. They traverse every node in the syntax tree of the current path expression P of the whole program.

For each node in the syntax tree, the rules in Table I and Table II take constant time. Since we replace every x in P with P_x for a non-sink node x in the DAG, the same subexpression P_x may appear in P many times. Intuitively, the results of rules in Table I and Table II are exactly the same for the same path expressions. With this property, each leaf node in the syntax tree of P_x is visited at most once by reusing the results of P_x for the others in P. Therefore, the number of leaf nodes in the syntax tree of P is bounded by the number of call sites of the program. The syntax tree is a binary tree with M leaf nodes, where M is the number of all call sites in the program. Thus, the maximum number of nodes in the syntax tree is $2 \cdot M$. The time complexity of both $WCNI(x)$ and $WCNC(x, y)$ for functions x and y is, therefore, given by $O(2 \cdot M)$. Since we visit every function in the DAG, we compute $WCNI(x)$ for N times and $WCNC(x, y)$ for N^2 times, where N is the number of nodes in the DAG. Consequently, the STRG construction phase takes $O(2 \cdot M \cdot N^2 + N \cdot m_{max} \cdot \alpha(m_{max}, n_{max}))$ time, where m_{max} and n_{max} are the maximum number of edges and nodes in the CFG of a function in the program.

6.2. Code Segment Placement

Algorithm 2 shows the detailed algorithm of our code segment placement algorithm. The function `place` is invoked to find out an offset of the minimum cost for each function. After placing all functions, the overlay structure is kept in `MAP`. `MAP` maps a function x to (`offset`, `alone`). The `alone` flag indicates if x overlaps with any other function. The function `overlap_size(offset,` x`,` y`)` calculates the size of overlapping memory regions between functions x and y when x is placed at `offset`. The innermost loop iterates over the number of functions that have been placed. The number of candidate locations for a given memory size S_{mem} is $\lfloor S_{mem}/S_{inc} \rfloor$. Thus, our placement algorithm takes $O(\lfloor S_{mem}/S_{inc} \rfloor \cdot N^2)$ time to generate a final overlay structure for a given program with N functions.

6.3. Redundancy Elimination

With the overlay structure, the linker finds optimal positions where the glue code is inserted according to the algorithm described in Section 5. The number of passes taken by the iterative data flow algorithm is generally known as $d + 2$ [Aho et al. 2006], where d is the maximum number of back edges on any acyclic path in the CFG. Since we are dealing with the CFG of a whole program, the time complexity of the data flow algorithms described in Section 5 is given by $O(D \cdot B)$, where D is the maximum number of back edges on any acyclic path in the CFG of the whole program and B is the number of all basic blocks in the CFG.

ALGORITHM 2: The detailed algorithm of placing a function x

Input: x : a function to place, S_{mem} : the size of available higher-level memory.
Output: MAP : an overlay structure.
Data: $WCNI$: the worst-case number of invocations, $WCNC$: the worst-case number of potential conflict misses.

```
function place(x, S_mem)
begin
    S_x := the size of the given function x ;
    min_cost := INFINITE ;                                   /* the minimum cost */
    offset_min_cost := 0 ;                                   /* the offset with the minimum cost */
    F_min_cost := ∅, G_min_cost := ∅ ;                       /* F and G when x is placed at offset_min_cost */
    offset := 0 ;
    while offset ≤ (S_mem − S_x) do
        /* compute F, G, C_loading and C_checks when x is placed at offset */
        F := ∅ , G := ∅ ;
        C_loading := 0 ;                                     /* the cost of loading code segments */
        C_checks := 0 ;                                      /* the cost of runtime checks */
        foreach y ∈ MAP do
            S_xy := overlap_size(offset,x,y) ;
            if S_xy > 0 then
                F := F ∪ {y} ;
                C_loading += WCNC[x][y] × S_xy ;
                if MAP[y].alone = true then
                    G := G ∪ {y} ;
                    C_checks += WCNI[y] ;
                end
            end
        end
        if F ≠ ∅ then
            C_checks += WCNI[x] ;
        end
        C_total := α × C_loading + β × C_checks ;
        if C_total < min_cost then
            min_cost := C_total ;
            offset_min_cost := offset ;
            F_min_cost := F , G_min_cost := G ;
        end
        offset += S_inc ;
    end
    MAP[x].offset := offset_min_cost ;                       /* place x at offset_min_cost */
    if F_min_cost ≠ ∅ then
        MAP[x].alone := false ;                              /* x overlaps with y ∈ F_min_cost */
    else
        MAP[x].alone := true ;                               /* x does not overlap with any other function */
    end
    foreach y ∈ G_min_cost do
        MAP[y].alone := false ;                              /* x overlaps with y ∈ G_min_cost */
    end
end
```

Table III. PXA320 System Configuration

Clock frequency	Core	806Mhz
	Internal SRAM	312Mhz
	External DDR SDRAM	156Mhz
Memory size	Internal SRAM	768KB
	External DDR SDRAM	256MB
Data cache	32KB, 32-way set associative write-back for the external DDR SDRAM	
Instruction cache	Disabled	

7. EVALUATION

In this section, we describe evaluation methodology and results of our code overlaying technique.

7.1. Methodology

Since the base case for our evaluation is the case when the entire code resides in the higher-level memory, we choose a system that has a large higher-level memory block: Marvell PXA320 development board [Marvell 2009; Intel Corp. 2004]. Table III shows the system configuration of our PXA320 system. It has an explicitly managed memory hierarchy that consists of internal SRAM and external DDR SDRAM. Execution time is measured with the performance monitors available in the PXA320 processor. We enable the L1 data cache to exclude the effect of data accesses as much as possible. The internal SRAM address space is configured uncachable to avoid coherence problems that can be caused by the DMA transfer.

Benchmark applications. We use a set of 21 benchmark applications to evaluate our approach. The set includes four applications from MiBench [Guthaus et al. 2001], two from MediaBench [Lee et al. 1997], six from SPEC2000 [SPEC 2000], six from SPEC2006 [SPEC 2006], an MPEG-4 Xvid encoder and decoder [Xvid 2005], and the reference implementation of a MP3 decoder [ISO/IEC 1996]. Table IV shows the characteristics of each application including libraries. All but mp3 have some exposed functions that might be called through a pointer. We select the benchmark applications whose code size is less than the internal SRAM size (768KB) of PXA320 processor because we cannot obtain the base case performance for those applications if their code size is greater than the internal SRAM size. In addition, we exclude those applications whose code size is too small ($<$ 30KB) because the overlay technique will not be useful for them.

The linker. We have built a linker that implements the technique described in Section 4 and Section 5. To compare our approach to the previous techniques described, the linker also supports the previous code overlaying techniques described in Section 3. It takes application and library objects in the ARM ELF format as its inputs. To enable relocation, it disassembles the object binary into basic blocks and replaces hard-coded offsets of branches and load/store instructions by relocation information. After code transformations are performed, it generates an executable image in the ARM ELF format by resolving all relocations.

The linker initially generates an overlay structure using the algorithm described in Section 4. With this overlay structure, it finds optimal positions where the glue code is inserted according to the algorithm described in Section 5. Note that the size of each function may be slightly increased after the glue code insertion step. This implies that the initial overlay structure is not correct any more. For this reason, the linker measures the size of each function when the glue code is inserted. If there exists at least one function whose size is increased, the linker generates a new overlay structure with the function size measured in the previous overlay structure and finds optimal

Table IV. Characteristics of Benchmark Applications

	Source	A		B		C	
		Size [KB]	#	Size [KB]	#	Size [KB]	#
ammp	SPEC2000	217	364	46	39	117	175
art	SPEC2000	62	175	1	4	48	104
bzip2	SPEC2006	100	235	1	4	82	166
cjpeg	MiBench	141	521	62	181	61	244
crafty	SPEC2000	216	298	1	6	184	182
djpeg	MiBench	144	475	56	175	65	228
epic	MediaBench	40	152	6	8	28	102
hmmer	SPEC2006	250	753	9	29	168	489
lbm	SPEC2006	46	179	1	4	30	105
mcf	SPEC2006	36	173	1	6	26	118
milc	SPEC2006	185	418	2	11	134	255
mp3	MP3 ref.	36	110	0	0	31	84
mp4d	Xvid dec.	330	557	100	131	163	317
mp4e	Xvid enc.	333	573	101	136	127	224
parser	SPEC2000	149	489	1	8	68	386
pgp	MediaBench	179	517	1	7	84	230
sjeng	SPEC2006	138	299	2	13	24	164
toast	MiBench	34	166	2	13	146	116
twolf	SPEC2000	199	351	1	6	20	247
untoast	MiBench	34	166	2	13	16	100
vpr	SPEC2000	180	458	2	10	139	336

A: The total code size and the number of all functions. B: The total code size and the number of all exposed functions. C: The total code size and the number of all executed functions for the given input.

positions of the glue code again. It repeats this process until there is no function whose size is increased in the generated overlay structure.

Note that each function call is a basic block, and our algorithm inserts the glue code only at the beginning of a basic block. Since we know the set of function slices to be copied into the higher-level memory at the beginning of each basic block, we can generate each glue code as a function call either to `Load()` or `Load&Branch()` by passing the set of function slices as an argument. Thus, the size of glue code never varies. The maximum size increase of a function due to glue code insertion is bounded by the size when the glue code is inserted at the beginning of every basic block. Therefore, the aforementioned repetition due to the function size increase always halts. The last column in Table V shows the average number of iterations for each application. The number of iterations is typically 5 and at most 10.

When the `Load()` or `Load&Branch()` routine loads a slice to the higher level memory, it selectively uses load and store instructions or DMA operations depending on the slice size. When the size is big, DMA operations are used. If not, simple load and store instructions are inserted to copy the slice to the higher-level memory. When DMA operations are used, computation and data transfer overlapping can be exploited to boost performance.

Compile time. Table V shows the average compile time spent for generating the overlaid executable of each application by traditional techniques and our approach. C&L and SDRM show the compile time of the previous approaches proposed by Cytron and Loewner [1986] and Pabalkar et al. [2008], respectively. FMUM and FMUP show the compile time of each approach proposed by Jung et al. [2010]. STRG shows the compile time of our approach. C&L takes the least compile time because it neither analyzes the control flow of the program nor estimates the cost of placing a function in the overlay structure. FMUM and FMUP show very poor compile time because they traverse all nodes in the GCCFG whenever they measure the cost for placing a function

Table V. Comparison of Compile Time to the Traditional Techniques

	C&L (sec)	SDRM (sec)	FMUM (sec)	FMUP (sec)	STRG (sec)	# of iterations
ammp	0.0002	20.60	526.35	1380.75	2.59	3.25
art	0.0001	0.47	22.27	70.96	1.25	4.63
bzip2	0.0001	0.87	86.69	238.07	3.05	5.43
cjpeg	0.0002	0.69	187.15	598.93	2.76	5.75
crafty	0.0002	72.98	317.53	1094.63	6.56	4.00
djpeg	0.0002	0.48	122.79	338.91	2.24	4.63
epic	0.0001	0.19	13.02	30.34	0.52	5.25
hmmer	0.0002	9.19	507.23	1633.75	20.31	5.25
lbm	0.0001	1.54	30.80	91.41	0.91	5.20
mcf	0.0001	0.22	25.86	78.41	1.01	5.88
milc	0.0002	15.60	640.20	2065.64	9.33	6.13
mp3	0.0001	0.09	4.46	10.70	0.45	3.14
mp4d	0.0001	4.70	73.03	190.04	2.13	3.57
mp4e	0.0001	1.65	78.91	241.90	2.45	4.88
parser	0.0006	12.20	3535.80	12851.60	11.63	6.50
pgp	0.0002	13.09	522.28	1652.19	25.77	6.38
sjeng	0.0002	4.18	145.84	384.56	2.83	5.50
toast	0.0001	0.15	22.28	61.32	0.78	4.88
twolf	0.0002	24.46	1213.03	3993.00	21.18	5.38
untoast	0.0001	0.11	12.17	30.80	0.49	5.00
vpr	0.0003	28.54	3487.00	12582.33	29.56	8.63
avg.	0.0002	10.09	551.18	1886.18	7.04	5.20

at a memory location. Our approach takes less time than SDRM on average and takes much more compile time than C&L. However, the compile time of our approach does not exceed 30 seconds for all applications.

7.2. Results

Figure 11 shows our evaluation result for each benchmark application. The speedup is obtained by comparing the execution time to that of the case when all code segments are placed in the internal SRAM. C&L shows the speedup of the traditional overlaying technique that is proposed by Cytron and Loewner [1986]. SDRM shows the speedup of the static overlaying technique proposed by Pabalkar et al. [2008]. NOSLICE shows the speedup of our approach without the slicing technique and any optimizations. STRG shows the speedup of our approach with the slicing technique but without any optimizations. STRG_PRE is the speedup when the redundant fetch elimination is applied. STRG_PRE_LOOP is the speedup when the loop optimization is applied in addition to the redundant fetch elimination.

Comparison on overlay structure size. We see that STRG and SDRM can generate an overlay structure whose size is much smaller than that of the overlay structure generated by C&L. Since C&L generates a minimal overlay structure for a given program, the size of the internal SRAM must be equal to or greater than that of the minimal overlay structure. However, unlike C&L, SDRM and our algorithms can generate an overlay structure that fits in an arbitrary memory size. Thus, we vary the size of available internal SRAM from 20% to 90% of the original code size in each application. For ammp, crafty, parser, and sjeng, they contain many SCCs that are merged together by the STRG algorithm, that is, 62.4%, 74.9%, 35.0%, and 50.5% of the original code size, respectively. For this reason, STRG cannot generate an overlay structure whose size is less than 70%, 80%, 40%, and 60% of the original code size for them.

Comparison on performance. The execution time of STRG at the same SRAM size as that of C&L is comparable to or better than that of C&L for all applications. C&L shows significant performance degradation for mcf and vpr. This is because C&L does

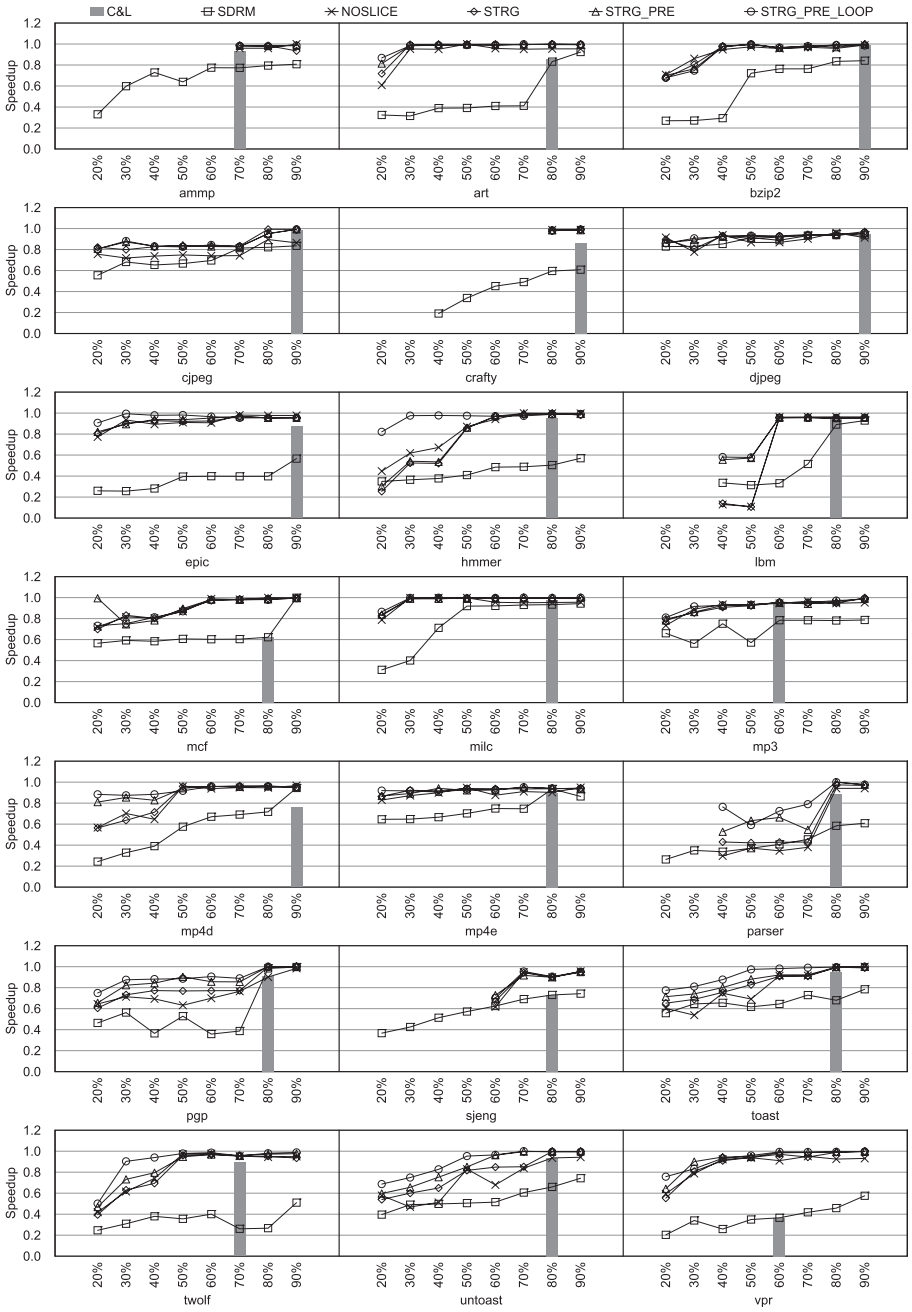


Fig. 11. Comparison with the traditional techniques.

not consider the temporal relationship between functions when it generates the overlay structure, resulting in many switches between functions in the same memory region.

Except djpeg, STRG shows much better performance than SDRM for all applications and for all SRAM sizes. The execution time of SDRM at the same SRAM size as that of C&L is even worse than that of C&L for ammp, crafty, epic, hmmer, parser, toast, twolf,

Table VI. Effect of Redundancy Elimination (The execution time is normalized to STRG)

SRAM size	STRG_PRE	STRG_PRE_LOOP
20%	0.90	0.81
30%	0.95	0.88
40%	0.88	0.81
50%	0.88	0.86
60%	0.96	0.95
70%	0.97	0.95
80%	1.00	1.00
90%	1.00	1.00

and untoast. This is because the I-Graph construction algorithm described in Section 3 wrongly estimates interference costs between functions and generates an inefficient overlay structure. To show an example, let us assume that M in Figure 3(a) is not the entry function of the program and M is invoked once in the program. Then, the weight of an edge (M, C) in the I-Graph is 1200 by the second rule of I-Graph construction because the LCA of M and C is M in Figure 3(a). However, we see that the interference cost between M and C depends on the number of iterations of the loop L1 (i.e., 10 in Figure 3(a)), not the number of invocations of C (i.e., 200 in Figure 3(a)).

The effect of SRAM size. As the SRAM size decreases, STRG's performance declines gradually for most applications. The degree of degradation becomes significant for hmmer, lbm, mp4d, twolf, and vpr. There are some exceptional cases: djpeg at 30%, mcf at 40% and parser at 50%, 60%, and 70%. The execution time of these cases is significantly worse than that at a smaller SRAM size. This is because our assumption that the number of iterations is the same for all loops does not reflect the real program behavior well and the cost metric used in the overlay structure generation (Section 4) suboptimally places some functions in the overlay structure.

The execution time of STRG does not significantly change at the SRAM size that is bigger than 60% for most applications. With more SRAM space, the generated overlay structure has fewer conflict misses between functions as well as fewer runtime checks caused by the glue code.

The effect of slicing. When the function slicing described in Section 4 is applied to the overlay structure, the overlay manager can reduce redundant memory copying at the additional cost of bookkeeping function slices that are loaded in the higher-level memory. When the slice management cost is bigger than the gain of reducing redundant memory copying, NOSLICE will show better performance than STRG. NOSLICE at 20%, 30%, and 40% of hmmer show this effect. Except hmmer, we see that STRG shows better performance than NOSLICE for all applications, especially for parser, pgp and untoast. More important benefit of function slicing is that the property of a function slice, that is, a function slice is either entirely overlap or does not overlap at all with another function slice, makes our redundancy elimination optimizations possible as described in Section 5. We see that our redundancy elimination algorithms (STRG_PRE and STRG_PRE_LOOP) amortize the slice management cost by eliminating redundant code fetches.

The effect of redundancy elimination. STRG_PRE_LOOP is especially effective for hmmer, lbm, mp4d, parser, twolf and untoast. STRG_PRE is worse than STRG for some applications. This is because STRG_PRE and STRG have different overlay structures. As we discussed in Section 7.1, the size of a function may changed after finding optimal program points for glue code insertion. Thus, the linker re-generates the overlay structure. The same is true between STRG_PRE_LOOP and STRG_PRE.

Table VI summarizes the effect of our redundancy elimination optimizations. It shows the average (geo. mean) execution time of STRG_PRE and STRG_PRE_LOOP over all

Table VII. Effect of Redundancy Elimination (number of fetches and amount of memory copying)

	SRAM size	A		B	
		C	D	C	D
ammp	90%	1.00	1.00	1.00	1.00
art	20%	0.96	0.06	0.02	0.05
bzip2	30%	1.87	1.07	1.82	1.05
	40%	0.94	0.59	0.60	0.58
cjpeg	20%	1.00	0.93	1.00	0.95
	30%	1.00	0.93	1.00	0.94
	40%	1.00	0.93	1.00	0.93
crafty	80%	0.31	0.71	0.31	0.71
	90%	0.09	0.11	0.08	0.11
djpeg	20%	0.99	0.71	0.99	0.28
	30%	0.99	0.11	0.99	0.06
epic	20%	0.86	0.24	0.18	0.18
	30%	0.99	0.91	0.02	0.15
	40%	0.99	0.05	0.02	0.05
	50%	0.99	0.01	0.02	0.01
hmmer	20%	0.48	0.85	0.01	0.83
	30%	0.92	0.51	0.01	0.50
	40%	1.00	0.83	0.01	0.80
	50%	1.00	0.64	0.04	0.84
	60%	0.83	0.85	0.77	0.87
lbm	40%	0.07	0.29	0.07	0.29
	50%	0.08	0.07	0.07	0.07
	60%	1.38	0.44	0.96	0.44
mcf	20%	0.99	0.65	0.78	0.79
	30%	0.97	1.57	0.98	1.65
	40%	1.00	0.59	0.95	0.60
milc	20%	0.66	0.87	0.09	0.80
	30%	0.93	0.29	0.09	0.66
mp3	20%	0.94	0.84	0.83	0.89
	30%	0.80	0.92	0.45	0.96
	40%	0.66	0.98	0.55	0.98
	50%	0.86	0.93	0.90	0.98
mp4d	20%	0.19	0.55	0.14	0.05
	30%	0.26	0.99	0.21	0.86
	40%	0.42	0.77	0.39	0.00
mp4e	20%	0.74	0.95	0.55	0.31
	30%	1.17	0.32	0.89	0.60
	40%	0.87	0.29	0.86	0.09
parser	40%	0.33	0.59	0.14	0.63
	50%	0.30	0.25	0.34	0.97
	60%	0.31	0.89	0.14	0.73
	70%	0.31	0.53	0.14	0.56
	80%	0.58	0.61	0.47	0.73
pgp	20%	0.74	0.89	0.27	1.00
	30%	0.34	0.06	0.24	0.09
	40%	0.32	0.04	0.23	0.05
	50%	0.31	0.86	0.22	0.91
	60%	0.31	0.60	0.22	0.94
	70%	0.31	0.81	0.22	0.73
sjeng	60%	0.62	0.14	0.70	0.14
	70%	0.95	0.30	0.95	0.68
	80%	0.97	0.75	0.97	0.75
toast	20%	0.51	0.89	0.29	0.95
	30%	0.51	0.98	0.27	0.97
	40%	0.66	0.88	0.35	0.88
	50%	0.50	0.93	0.02	0.94
	60%	0.97	0.55	0.03	0.56
	70%	0.99	0.99	0.03	0.99
	80%	0.86	0.93	0.86	0.93
twolf	20%	0.54	0.49	0.48	0.54
	30%	0.43	0.73	0.17	1.00
	40%	0.43	0.41	0.17	0.96
	50%	0.84	0.33	0.80	0.54
untoast	20%	0.51	0.94	0.27	0.81
	30%	0.50	0.69	0.26	0.70
	40%	0.65	0.47	0.34	0.46
	50%	0.96	0.34	0.04	0.34
	60%	0.03	0.12	0.03	0.12
	70%	0.01	0.83	0.01	0.83
vpr	20%	0.69	0.76	0.46	0.67
	30%	0.37	0.14	0.93	0.27
	40%	0.99	0.43	0.86	0.38
	50%	0.93	0.24	0.70	0.28

A: STRG_PRE, B: STRG_PRE_LOOP, C: the number of fetches, D: amount of memory copying.

applications at each SRAM size. The execution time is normalized to STRG. The effect of the redundant fetch elimination (STRG_PRE) manifests itself as the SRAM size decreases. As the SRAM size decreases, the number of redundant fetches increases. Thus, we have more opportunities for optimization at a smaller SRAM size. For the same reason, STRG_PRE_LOOP is better than STRG_PRE at a smaller SRAM size. STRG_PRE is on average up to 13.8% faster than STRG and STRG_PRE_LOOP is on average up to 23.9% faster than STRG.

Table VII shows the comparison between STRG, STRG_PRE, and STRG_PRE_LOOP for the number of fetches and the amount of memory copying. The numbers are normalized to those of STRG. We see that the optimizations show the less number of fetches or the less size of fetches, or both. Sometimes STRG_PRE has more fetches than STRG (lbm 60% and mp4e 30%). This is because STRG_PRE generates a different overlay structure from that of STRG for those cases. The amount of memory

copying in STRG_PRE_LOOP is often bigger than that in STRG_PRE. This is because STRG_PRE_LOOP moves the fetches for loop invariant slices to the loop preheader. This may cause extra code fetches when the loop is not executed. However, the number of fetches in STRG_PRE_LOOP is less than or equal to that in STRG_PRE for all applications and SRAM sizes. Overall, our partial redundancy elimination technique is quite effective.

In summary, our approach (STRG_PRE_LOOP) is, on average, 13% faster than C&L with the SRAM size that is comparable to that of C&L. With the execution time that is comparable to that of C&L, our approach can reduce the SRAM size to 41.7% of the original code size, on average. If we allow 5% slowdown compared to the base case, on average, our approach can reduce the SRAM size to 55.3% of the original code size.

Comparison with a profiling-based approach. Figure 12 shows our evaluation result that compares our static approach (STRG_PRE _LOOP) to the profiling-based approaches. The speedup is obtained by comparing the execution time to that of the case when all code segments are placed in the internal SRAM. After profiling an application with the input described in Table IV and obtaining a call and return trace, we obtain the count of alternations between all the functions that appear in the trace. Then, we construct a dynamic TRG (DTRG) for DTRG_PRE_LOOP in Figure 12.

WCG shows the speedup of the approach based on the weighted call graph (WCG) [Hashemi et al. 1997; Kalamatianos and Kaeli 1998; Pettis and Hansen 1990] with profiling information. FMUM+FMUP shows the speedup of the better execution time between FMUM and FMUP [Jung et al. 2010] at the same SRAM size. We see that STRG_PRE_LOOP shows much better performance than WCG and FMUM+FMUP. The reason is that WCG cannot represent the temporal relationship between two functions that do not call each other even with profiling information [Gloy et al. 1997]. FMUM+FMUP shows better performance than WCG but much worse performance than DTRG_PRE_LOOP. The same is true for even STRG_PRE_LOOP. The performance improvement of STRG_PRE_LOOP over FMUM+FMUP becomes larger as the SRAM size decreases.

STRG_PRE_LOOP is comparable to DTRG_PRE_LOOP for most of the applications. However, STRG_PRE_LOOP is worse than DTRG_PRE_LOOP for some larger SRAM sizes in cjpeg, parser, and sjeng. Note that DTRG_PRE_LOOP can generate an overlaid executable for ammp, crafty, parser, and sjeng for the SRAM sizes with which STRG_PRE_LOOP cannot generate any overlay structure due to SCCs. The profiling-based approach may cause a problem if the unexecuted functions are invoked for an input that is different from the profiling input. In this case, the unexecuted functions can be processed with STRG_PRE_LOOP. Consequently, a combination of profiling-based approach and the static approach will give a better result.

8. CONCLUSION

We introduce an automatic overlay generation algorithm and a method to eliminate redundant memory copying operations for explicitly managed memory hierarchies. Our overlay generation algorithm analyzes the whole program including libraries and generates an overlay structure for an arbitrary memory size as long as the memory size is larger than the size of the biggest function in the given program. It exploits the static temporal relationship between two different functions when generating the overlay structure. The relationship is estimated by the Worst-Case Number of Conflict misses (WCNC) between them and it is computed with path expressions. With a given overlay structure, our interprocedural data-flow analysis algorithm can find the optimal program points to copy each function into the higher-level memory. This results in eliminating the redundant memory copying overhead. The experimental results indicate that our approach overcomes the limitations of the traditional code overlaying

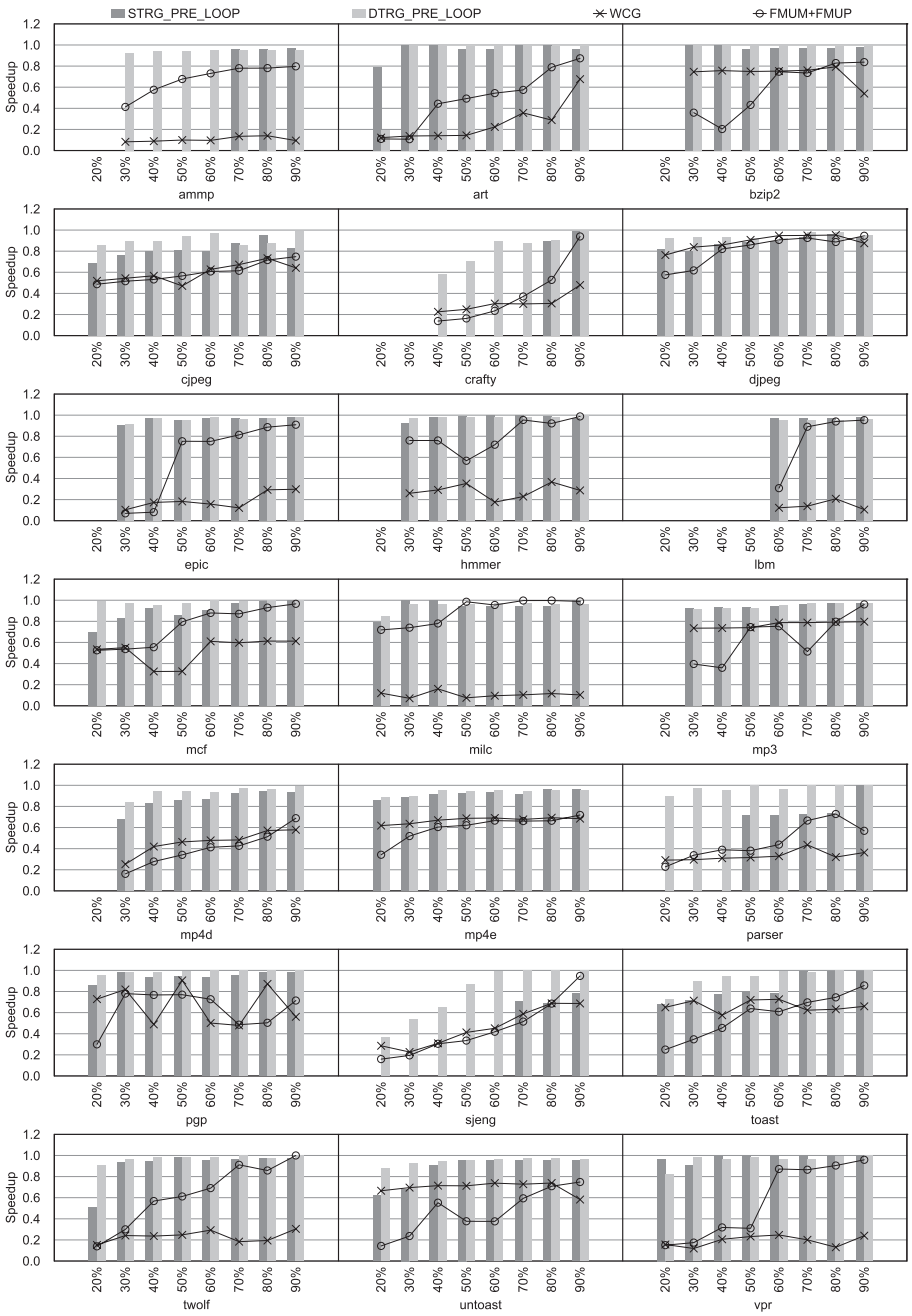


Fig. 12. Comparison with a profiling-based approach.

techniques. Our algorithm outperforms the previous work not only with respect to execution time but also memory savings. In combination with a profiling-based approach, our algorithm will result in a better memory savings and performance. With our code overlay technique, the developer can determine the size of code memory as small as possible at the cost of tolerable performance degradation.

REFERENCES

AGRAWAL, G., SALTZ, J., AND DAS, R. 1995. Interprocedural partial redundancy elimination and its application to distributed memory compilation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'95)*. ACM, New York, NY, 258–269.

AHO, A. V., LAM, M. S., SETHI, R., AND ULLMAN, J. D. 2006. *Compilers: Principles, Techniques, And Tools* 2nd Ed. Addison-Wesley Longman Publishing Co., Inc., Boston, MA.

ANGIOLINI, F., MENICHELLI, F., FERRERO, A., BENINI, L., AND OLIVIERI, M. 2004. A post-compiler approach to scratchpad mapping of code. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'04)*. ACM, New York, NY, 259–267.

ARM LTD. 2009. http://www.arm.com.

BANAKAR, R., STEINKE, S., LEE, B.-S., BALAKRISHNAN, M., AND MARWEDEL, P. 2002. Scratchpad memory: Design alternative for cache on-chip memory in embedded systems. In *Proceedings of the 10th International Symposium on Hardware/Software Codesign (CODES'02)*. ACM, New York, NY, 73–78.

BODÍK, R., GUPTA, R., AND SOFFA, M. L. 1998. Complete removal of redundant expressions. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98)*. ACM, New York, NY, 1–14.

CYTRON, R. AND LOEWNER, P. G. 1986. An automatic overlay generator. *IBM J. Resear. Dev. 30*, 6, 603–608.

EGGER, B., KIM, C., JANG, C., NAM, Y., LEE, J., AND MIN, S. L. 2006a. A dynamic code placement technique for scratchpad memory using postpass optimization. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'06)*. ACM, New York, NY, 223–233.

EGGER, B., LEE, J., AND SHIN, H. 2006b. Scratchpad memory management for portable systems with a memory management unit. In *Proceedings of the 6th ACM/IEEE International Conference on Embedded Software (EMSOFT'06)*. ACM, New York, NY, 321–330.

EGGER, B., LEE, J., AND SHIN, H. 2008. Scratchpad memory management in a multitasking environment. In *Proceedings of the 8th ACM/IEEE International Conference on Embedded Software (EMSOFT'08)*. ACM, New York, NY, 265–274.

GLOY, N., BLACKWELL, T., SMITH, M. D., AND CALDER, B. 1997. Procedure placement using temporal ordering information. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture.* IEEE Computer Society, Los Alamitos, CA, 303–313.

GUILLON, C., RASTELLO, F., BIDAULT, T., AND BOUCHEZ, F. 2004. Procedure placement using temporal-ordering information: dealing with code size expansion. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'04)*. ACM, New York, NY, 268–279.

GUTHAUS, M. R., RINGENBERG, J. S., ERNST, D., AUSTIN, T. M., MUDGE, T., AND BROWN, R. B. 2001. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the IEEE International Workshop onWorkload Characterization (WWC'01)*. IEEE Computer Society, Los Alamitos, CA, 3–14.

HASHEMI, A. H., KAELI, D. R., AND CALDER, B. 1997. Efficient procedure mapping using cache line coloring. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'97)*. ACM, New York, NY, 171–182.

HE, H., DEBRAY, S. K., AND ANDREWS, G. R. 2007. The revenge of the overlay: automatic compaction of OS kernel code via on-demand code loading. In *Proceedings of the 7th ACM/IEEE International Conference on Embedded Software (EMSOFT'07)*. ACM, New York, NY, 75–83.

IBM. 2007. *Software Development Kit for Multicore Acceleration* version 3.0*, Programmer's Guide*. IBM.

IBM, SONY, AND TOSHIBA. 2007. Cell broadband engine architecture. IBM. http://www.ibm.com/developerworks/power/cell/.

INTEL CORP. 2004. *Intel XScale Core Developer's Manual*. http://www.intel.com.

ISO/IEC. 1996. IS 13818-3 information technology - generic coding of moving pictures and associated audio: Audio. MP3.

JUNG, S. C., SHRIVASTAVA, A., AND BAI, K. 2010. Dynamic code mapping for limited local memory systems. In *Proceedings of the 21st IEEE International Conference on Application-Specific Systems Architectures and Processors (ASAP)*. 13–20.

KALAMATIANOS, J. AND KAELI, D. 1998. Temporal-based procedure reordering for improved instruction cache performance. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture (HPCA'98)*. IEEE Computer Society, Los Alamitos, CA, 244.

KNOOP, J., RÜTHING, O., AND STEFFEN, B. 1992. Lazy code motion. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'92)*. ACM, New York, NY, 224–234.

LARUS, J. R. 1999. Whole program paths. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'99)*. ACM, New York, NY, 259–269.

LEE, C., POTKONJAK, M., AND MANGIONE-SMITH, W. H. 1997. Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*. IEEE Computer Society, Los Alamitos, CA, 330–335.

LEVINE, J. R. 2000. *Linkers and Loaders.* Morgan Kaufmann.

MARVELL. 2009. Marvell PXA3xx processor family. http://www.marvell.com.

MATTSON, T. G., VAN DER WIJNGAART, R., AND FRUMKIN, M. 2008. Programming the Intel 80-core network-on-a-chip terascale processor. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC'08)*. IEEE Press, 1–11.

PABALKAR, A., SHRIVASTAVA, A., KANNAN, A., AND LEE, J. 2008. Sdrm: simultaneous determination of regions and function-to-region mapping for scratchpad memories. *In Proceedings of the 15th International Conference on High Performance Computing (HiPC'08)*. Springer, 569–582.

PETTIS, K. AND HANSEN, R. C. 1990. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'90)*. ACM, New York, NY, 16–27.

SPEC. 2000. SPEC. http://www.spec.org/benchmarks.html.

SPEC. 2006. SPEC. http://www.spec.org/benchmarks.html.

STEINKE, S., WEHMEYER, L., LEE, B., AND MARWEDEL, P. 2002. Assigning program and data objects to scratchpad for energy reduction. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'02)*. IEEE Computer Society, Los Alamitos, CA, 409.

TARJAN, R. E. 1981a. Fast algorithms for solving path problems. *J. ACM* 28, 3, 594–614.

TARJAN, R. E. 1981b. A unified approach to path problems. *J. ACM 28*, 3, 577–593.

TEXAS INSTRUMENTS INC. 2009. http://www.ti.com.

VERMA, M., WEHMEYER, L., AND MARWEDEL, P. 2004a. Cache-aware scratchpad allocation algorithm. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'04)*. IEEE Computer Society, Los Alamitos, CA, 21264.

VERMA, M., WEHMEYER, L., AND MARWEDEL, P. 2004b. Dynamic overlay of scratchpad memory for energy minimization. In *Proceedings of the 2nd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'04)*. ACM, New York, NY, 104–109.

XVID. 2005. Xvid MPEG-4 video codec. http://www.xvid.org.

Received April 2010; revised April 2011, July 2011; accepted September 2011