Construction of GCCFG for Inter-procedural Optimizations in Software Managed Manycore (SMM) Architectures

Bryce Holton[1], Ke Bai[1], Aviral Shrivastava[1], and Harini Ramaprasad[2]

[1]Arizona State University

[2]Southern Illinois University Carbondale

{Bryce.Holton, Ke.Bai, Aviral.Shrivastava}@asu.edu, harinir@siu.edu

ABSTRACT

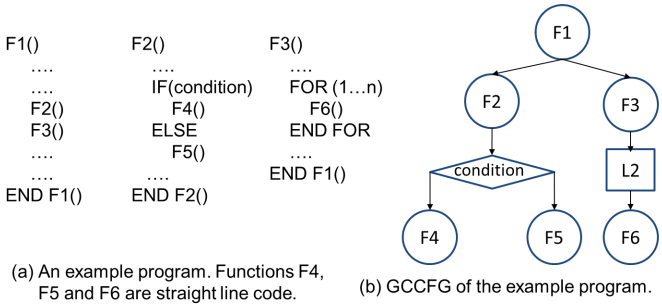
Software Managed Manycore (SMM) architectures – in which each core has only a scratch pad memory (instead of caches), – are a promising solution for scaling memory hierarchy to hundreds of cores. However, in these architectures, the code and data of the tasks mapped to the cores must be explicitly managed in the software by the compiler. State-of-the-art compiler techniques for SMM architectures require inter-procedural information and analysis. A call graph of the program does not have enough information, and Global CFG, i.e., combining all the control flow graphs of the program has too much information, and becomes too big. As a result, most new techniques have informally defined and used GCCFG (Global Call Control Flow Graph) – a whole program representation which captures the control-flow as well as function call information in a succinct way – to perform inter-procedural analysis. However, how to construct it has not been shown yet. We find that for several simple call and control flow graphs, constructing GCCFG is relatively straightforward, but there are several cases in common applications where unique graph transformation is needed in order to formally and correctly construct the GCCFG. This paper fills this gap, and develops graph transformations to allow the construction of GCCFG in (almost) all cases. Our experiments show that by using succinct representation (GCCFG) rather than elaborate representation (GlobalCFG), the compilation time of state-of-the-art code management technique [4] can be improved by an average of 5X, and that of stack management [20] can be improved by an average of 4X.

1. INTRODUCTION

Scaling the memory architecture is a major challenge as we transition from a few cores to many core processors. Experts believe that coherent cache architectures will not scale to hundreds and thousands of cores [11, 12, 16, 25], not only because the hardware overheads of providing coherency increases rapidly with core count, but also because caches consume a lot of power. One promising option for a more power-efficient and scalable memory hierarchy is to use raw, "uncached" memory (commonly known as Scratch Pad Memory or SPM) in the cores. Since SPM does not have the hardware for address lookup and translation, they occupy 30% less area, and consume 30% less power than a direct mapped cache of the same effective capacity [10]. In addition, the coherence has to be provided in the software, so hardware is more power-efficient and scalable. A multicore/manycore architecture in which each core has an SPM instead of hardware caches is called a Software Managed Multicore (SMM) architecture [4,20]. The Cell processor [14] (used in PlayStation 3) is a good example of an SMM architecture. Thanks to the SMM architecture, the peak power-efficiency of the Cell processor is 5 GFlops per Watt [14]. Contrast this to the Intel i7 4-core Bloomfield 965 XE with power- efficiency of 0.5 GFlops per Watt [1, 2], both fabricated in the 65nm technology node.

The main challenge in the SMM architecture is that several tasks like data management (movement of data between SPMs of the cores and the main memory) and inter-core communication (movement of data between the SPMs of cores), which were originally done by the hardware (more specifically, the cache hierarchy) now have to be explicitly done in the software, and that may cause overheads. Recent research results have been quite encouraging. Techniques have been proposed to manage all kinds of data: code [4,19], stack [8, 20], and heap [3, 5–7] efficiently on the SPMs of the core. In fact, [4] and [20] show that the overhead of code and stack management on SPMs is lower than on a cache based architecture. Thus SMMs are coming up as a strong contender for processor architecture in the manycore era. All the state-of-the-art data management techniques that have been developed for SMM architectures are inter-procedural code transformations and require extensive inter-procedural analysis. One of the fundamental challenges in this research has been in finding out the right representation of the whole program, such that it captures the required information, and yet is not too big and cumbersome. For example, the call graph of a program captures which functions calls which other function, but it does not contain information about the loops and if-then-elses present in the program. Also it does not contain information about the order in which the functions are called. All this information is vital for

ESWEEK'14, New Delhi, India.



(a) An example program. Functions F4, F5 and F6 are straight line code.

(b) GCCFG of the example program.

Figure 1: *A code snippet and a corresponding GCCFG representation of the program. Assume that functions F4, F5 and F6 are straight line code.*

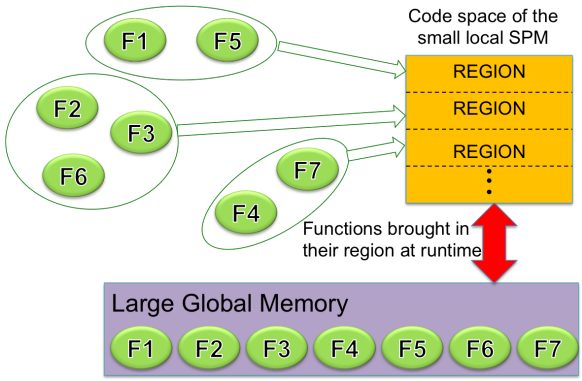


Figure 2: *There are two parts to the problem of optimal code mapping i) Divide the code space on SPM into regions, and ii) Map functions into regions. Functions mapped to the same region replace each other when called. Therefore, for a given code space, we have to find a division of that space into regions and mapping of functions into regions so that there is minimum data transfer between SPM and the main global memory.*

the code transformations required for SMM architectures. Control Flow Graph or CFG contains detailed information of all the control flow, but it is only for a single function. Researchers have tried to stitch together the CFGs of various functions by essentially pasting the CFG of the called function at the site of its call – named Global CFG [22–24]. Global CFG is a detailed representation of the whole program, but it grows too big, and discovering the information SMM transformations need from this graph is very time consuming and cumbersome at the least. What is needed is a succinct representation of the whole program that contains functional call information, as well as important control flow structures of the program, e.g., loops and if-then-elses.

Recent, previous research on developing code transformations for SMM architecture have used Global Call Control Flow Graph or GCCFG [4, 7, 9, 19, 20]. GCCFG is a whole program representation, and is a hybrid between a call graph and control flow graph. GCCFG is hierarchical representation of the program, abstracts away straight line code, and captures the function call and control flow information of the program. GCCFG is ordered hierarchical representation of the program, and consists of three kinds of nodes, function nodes (shown as circles in Figure 1), loop nodes (shown as squares in Figure 1), and condition nodes (shown as diamonds in Figure 1). The GCCFG of a simple program is shown in Figure 1. In the program function $F1$ calls functions $F2$ and $F3$ in order. That is why $F2$ is the left child and $F3$ is the right child of $F1$. Function $F2$ contains an if-then-else, in which $F4$ is called in the if-part, and $F5$ is called in the else part. Function $F3$ contains a loop in which $F6$ is called. Functions $F4$, $F5$, and $F6$ contain only straight line code. Note that the GCCFG abstracts away the straight line code, and only keeps the function call and control flow information in a succinct hierarchical form.

While previous researches have informally defined and used GCCFG, they have not shown how to construct it for any given program. We find that, while constructing GCCFG for simple programs is relatively straightforward, there are several very commonly found program structures where constructing GCCFG is complicated. For example, loops that have multiple exits (commonly caused by continue , break, goto statements, and loops that return values), and intertwined loops (caused by goto statements), and switch statements, and if-then-elses with exit statements, etc. In this paper we formally define GCCFG and show the construction of GCCFG in all these cases. We also show that as opposed to Global CFG, GCCFG is a more succinct representation, and results in more efficient implementations of the inter-procedural analysis required for code and stack data management on SMM architectures. Experiments conducted on MiBench benchmarks [15] demonstrate that, the compilation time of a state-of-the art code management technique can be improved by an average of 5X, and that of stack management can be improved by 4X through GCCFG as compared to Global CFG.

2. WHY GCCFG FOR CODE MAPPING ON SMM ARCHITECTURES?

2.1 The optimal code mapping problem in SMM architectures

In an SMM architecture, a task is mapped to a core, and the core has access to only it's local Scratch Pad Memory or SPM. All code and data that the task uses must come from SPM. In case, the code of the application is larger than the SPM size, then the whole code can reside in the large global memory, but can be brought in piecemeal approach. To facilitate this, SMM architecture allows users to divide the code part of the SPM into regions, and functions in the program can be mapped to the regions in the SPM. This mapping of functions to regions is specified in the linker file, and is used at runtime. At runtime, of all the functions mapped to a region, only one can be present in the region at runtime. When a function is called, it is brought into its region in the SPM, and the existing function is simply overwritten. While any region formation, and mapping of functions to regions is legitimate, but if we have only one region, and map all the functions to that region, then there will be a function code replacement at every function call – severely hurting the performance. Similarly, if we have a separate region for each function, then the performance will be excellent, since there will be no replacements, but it will require a lot of space which may not be available in the SPM. *Thus, the problem of optimal code management is to find the division of the code space in the SPM into regions,*

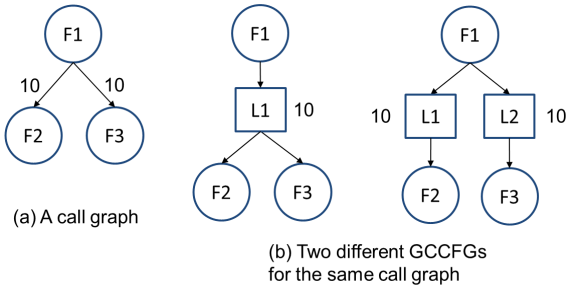


Figure 3: *For the same weighted call graph, there can be multiple GCCFGs, and they can have different sequence of function executions. Function execution sequence of first GCCFG is* $F1-(F2-F1-F3-F1)^{10}$*, while for the second GCCFG, the function execution sequence is* $F1-(F2-F1)^{10}-(F3-F1)^{10}$*.*

and find the mapping of functions to those regions so as to minimize the data transfers between the SPM and the main global memory.

2.2 What information do we need for optimal code mapping?

The fundamental information required to be able to solve this optimal code mapping problem is to estimate the amount of data transfers that will happen when two functions are mapped to the same region. For example, in the program shown in Figure 1, if $F3$ and $F6$ are mapped to the same region, then there will be a lot of replacements, since the function $F3$ calls $F6$ in a loop. Each time after the execution returns from $F6$, the execution comes back to the function $F3$, therefore it must be brought back into the SPM. On the other hand, it might be better to place the functions $F4$ and $F5$ in the same region, since only one of them will be called. Thus, in order to find out which functions can be mapped in the same region, we need information like which function calls another function, and whether a function call happens in a loop (and if so, how many levels of loops), an if-then-else, or just in a straight-line code – collectively called control flow information. GCCFG captures all this information in a succinct manner. In fact, the sequence in which the functions, are called in the program, can be derived relatively accurately from the GCCFG. For the given GCCFG, the approximate sequence of function executions is $F1-F2-(F4|F5)-F2-F1-(F3,F6)^*-F1$, where $(F4|F5)$ implies the execution of one of them. Note that after executing $F4$ or $F5$, the execution returns to $F2$ and then to $F1$, since they are the calling functions.

2.3 Why is call graph not enough?

The function execution sequence cannot be derived from a call graph. The first problem is that in the call graph, the order in which functions are called is absent, while GCCFG preserves the order. The left child of a node is called before the right child. Second, call graph looses all control flow information, so we do not know if a function is being called in a loop, or in an if-then-else, or just in straight-line code. While it is clear that each of these structures has a very significant impact on the sequence of the function executions, and therefore the number of times the function has to be brought into the SPM. In fact, even annotating the call graph with how many times a function is called is not enough. That is because, it still does not capture the context in which the functions are called. For example, Figure 3 shows two GCCFGs that can be drawn for the same call graph. The GCCFGs have very different function call sequence, which ultimately results in a different mapping being optimal for each case.

3. RELATED WORK

Data management optimizations require both the function call information and the control flow information. Since control flow information is so important, data management cannot be performed using just the information in the Call Graph. The Call Graph only has information about which functions call other functions, but it doesn't show an ordering to those calls, or control flow information. Data management techniques can use Global CFG instead [22–24]. Global CFG is a graph where all CFGs are inlined to make a single large graph. Indeed Global CFG contains all the information needed, however, the information is arranged in a manner, such that it is compute intensive to dig out the information we need.

Other program level level graphs have been defined and used in other contexts. System Dependence Graph (SDG) was designed by Horowitz et al [17]. In the SDG, nodes can be split at the sub basic block level to represent individual statements as well as procedure entry and exit points. The SDG also requires different types of edges between the nodes. There are edges that represent control dependence, as well as, flow and def-order dependence. In order to maintain context between procedure calls, a grammar is used to model the call structure of each procedure. While the SDG could be used as input for data management schemes, it is not succinct. The fact that it breaks basic blocks into smaller parts and introduces edges between them makes it quite large. This is opposite of what we want – we are attempting to abstract away the straight line code so that we can get a succinct representation of the whole program. Udayakumaran et al. [23] proposed Data-Program Relationship Graph (DPRG). They start with a call graph, and then append loop and condition nodes for control flow information. However, DPRG does not maintain ordering information, it must use a depth first search to get a general ordering. Also DPRG requires extra nodes for *then* and *else*, instead of just one node for IF-THEN-ELSE statements, making it less than a succinct representation of the program. Whitham et al. [24] proposes a graph called a Control Flow Tree (CFT). They derive this data structure from a control flow graph that has been converted into a tree, and then has been compressed into the final CFT. The graph proposed in their work maintains ordering by making sure that a parent node is always called before a child node. However, they must maintain a list of back edges to keep information about when the leaf of the tree needs to return to an earlier parent. The CFT is not a succinct representation of a program since it needs multiple data structures to represent control flow.

To facilitate data management optimizations on SMM architectures, Lee et al. [18] used regular expression, called a path expression, to represent the way control flows through a program where kleen(*) closure represents a loop, union(|) closure represents a condition, and concatenation(·) closure represents the next segment to be executed. This information reveals the alternating behavior between functions in a program, so that an efficient mapping of function code, to

memory can be made. The information present in the regular expression is also present in GCCFG, however, it is much easier to annotate GCCFG with more information, like the number of times a loop executes or branch probability, than to annotate a regular expression with more information.

The state of the art data management schemes [4, 9, 18, 20] for SMM architectures have used GCCFG or GCCFG-like data structures, but the construction of GCCFG has not been shown yet. This paper will formally define, and describe the algorithm to construct GCCFG.

Our approach to construct GCCFG is quite similar to the construction of Hierarchical Task Graph (HTG) [22], however, HTG is only for one function. HTG is a hierarchical representation of the program control flow, and is derived by separating the control flow graph into hierarchies at the loop level. We extend the HTG concepts to create an interprocedural graph, which we call GCCFG. However, GCCFG construction can become quite challenging when the program has ill-formed control flow, e.g., poorly formed loops, switch statements, and hard to find convergence point of conditions – and this paper proposes solutions to correctly construct GCCFG in these cases.

4. GLOBAL CALL CONTROL FLOW GRAPH

Global Call Control Flow Graph or GCCFG is a whole program view of structures within an application. Specifically, GCCFG identifies three different types of structures that are commonly found in a program: *function calls, loop blocks*, and *if-then-else condition blocks*.

DEFINITION 1. *(***GCCFG***) Let $G := (V, E)$ be a DAG, where $V = \{V_f \cup V_c \cup V_l\}$ is a set of vertices representing program structures and $\{e = (v_1, v_2) \in E : v_1 \in V \wedge v_2 \in V\}$ is a set of directed edges. Then G is a GCCFG of a program where the vertices identify three program structures, function calls, loops, and if-then-else conditions respectively, and the edges represent the change of control flow in a program structure; where the program code in v_2 is called from inside the program code corresponding to v_1.*

The three types of program structures represented by the vertex set in GCCFG are distinguished in the following ways: A vertex $v \in V_f$ represents a function call in a program, has only one set of outgoing edges, and is represented by a circle shape in the final GCCFG graph. The vertex $v \in V_l$ represents a loop in a program, also has only one set of outgoing edges, and is represented by a square in GCCFG. Finally, a vertex $v \in V_c$ represents an if-then-else condition in the program, where it has two sets of outgoing edges. One set of outgoing edges represent the path that control takes when the condition is true, and the other represent when the condition is false. A condition vertex is represented in GCCFG as a diamond shape. A set of outgoing edges from any vertex is an ordered set where vertices connected to edges on the left are executed before vertices to the right of them.

The GCCFG of the example program in Figure 1, has $V_f = \{F1, F2, F3, F4, F5, F6\}$, $V_l = \{L2\}$, and $V_c = \{condition\}$. The GCCFG represents a program that starts with function $F1$. Inside $F1$, two functions are called $F2$ and $F3$ in the order. Inside $F2$, there is an if-then-else, in which $F4$ is called in the true path, and $F5$ is called in the false path. Inside $F3$, the function $F6$ is called in a loop. Functions $F4$, $F5$, and $F6$ have only straight line code.

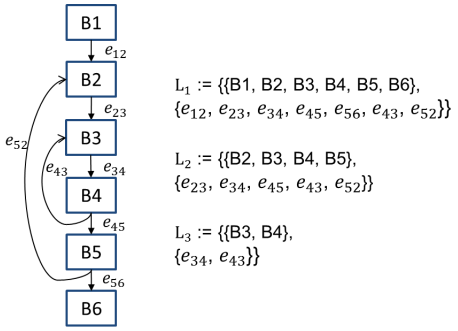


Figure 4: *A set of loops in a CFG. Each loop contains all of the basic blocks of itself and all loops nested inside of it.*

4.1 How to Construct GCCFG

GCCFG is constructed from the set of Control Flow Graphs (CFG) of a program. As each CFG represents a procedure in a program, we can view a procedure as a hierarchy of loops. Each hierarchy can be viewed as its own graph, and examined for necessary and unnecessary information. We can remove the unnecessary information from each hierarchy and glue the condensed graph to the other hierarchies in a procedure. Finally, we can glue all procedure graphs together at call sites to create a succinct whole program graph.

4.1.1 Step 1: Extracting Loop Information

A Program $p \supseteq$ CFG H, where $H = (B, E)$, represents a single function in a program. B is the set of basic blocks in the function, and E is the set of edges between basic blocks [13].

Given a set of CFGs we must extract the loop information from them. We find this information by looking for strongly connected components [13]. For now we assume each loop has a unique entry block and a unique exit block. If a loop doesn't have this property we will convert it so that they have a unique entry and exit block.

DEFINITION 2. *(***LOOP***) Each Loop $L \subset H$, where $L_i = (B_{L_i}, E_{L_i})$ and $H = (B, E)$ represents control flow graph of a program. $B_{L_i} \subset B$ are the blocks in a loop and the loops nested within it. $E_{L_i} \subset E$ are the edges between the blocks in a loop.*

Definition 2 explains that a loop is a set of blocks and a set of edges that are both a subset of the blocks and edges in a CFG. Figure 4 shows how the loop sets are extracted. In the example Loop 2 contains the information about Loop 3, while Loop 3 only has information about itself. In this work Loop 1 is a special case, which has all of the blocks and edges from the entire CFG. Namely, $L_1 = (B_{L_1}, E_{L_1}) = H$.

4.1.2 Step 2: Constructing Hierarchical Flow Graph

The next step after extracting the loop information is to separate the loops into hierarchical levels. All nested loops are a subset of the loop they are nested inside, so we identify which loop is one level of nesting below another loop. Therefore, $\forall L(LV(L) \rightarrow level)$ where the function LV finds the level of the loop.

In this work L_1 will always have the highest level hierarchy. So following the example in Figure 4: $LV(L_1) \rightarrow 1$, $LV(L_2) \rightarrow 2$, and $LV(L_3) \rightarrow 3$.

Algorithm 1: Build HFG

Input: A loop $L = (B_L, E_L)$
Output: An HFG $L' = (B'_L, E'_L)$

1 $B'_L \leftarrow B_L$
2 $E'_L \leftarrow E_L$
3 **forall the** $K \subset L : LV(K) = LV(L) + 1$ **do**
4 $B'_L \leftarrow B'_L - B_K$
5 $E'_L \leftarrow E'_L - E_K$
6 **forall the** $e \in E_L : e = (b_1, b_2)$ **do**
7 **if** $(b_1 \in \{B_L - B_K\}) \wedge (b_2 \in B_K)$ **then**
8 $B'_L \leftarrow B'_L + \{LPH\}$
9 $E'_L \leftarrow E'_L - \{e\} + \{e' : e' = (b_1, LPH)\}$
10 **if** $(b_1 \in B_K) \wedge (b_2 \in \{B_L - B_K\})$ **then**
11 $E'_L \leftarrow E'_L - \{e\} + \{e' : e' = (LPH, b_2)\}$

Algorithm 2: Build CCFG

Input: A HFG $L' = (B'_L, E'_L)$
Output: A CCFG $G' = (V', E')$

1 **forall the** $b \in B'_L$ **do**
2 **if** $OutDegree(b) := 2$ **then**
3 $V' \leftarrow V' + \{v \in V_c\}$
4 $m : b \mapsto v$
5 **if** $b := LPH$ **then**
6 $V' \leftarrow V' + \{v \in V_l\}$
7 $m : b \mapsto v$
8 **if** $(b := FPH) \vee (CodeContainsFunction(b))$ **then**
9 $V' \leftarrow V' + \{v \in V_f\}$
10 $m : b \mapsto v$
11 DepthFirstSearchHFG($Root(L')$,G')

By separating all of the loops in a CFG and identifying the hierarchy where they appear, we can use the loop information to build a new graph called Hierarchical Flow Graph (HFG). A HFG contains basic blocks and edges, where the blocks and edges hold the same meaning as a loop. A HFG also contains two new types of blocks: Loop Place Holder (LPH), and Function Place Holder (FPH). Both of these blocks represent an entry into a loop and a function respectively. Further the LPH and the FPH are used at the entry to a function or loop and at the call site for the corresponding loop or function. Each FPH and LPH is annotated with a label identifying which loop, or function it is a place holder for.

Definition 3. *(**Hierarchical Flow Graph**) An HFG $L' = (B'_L, E'_L)$ is a DAG, where B'_L represents all of the basic blocks in a loop plus one LPH for each highest level nested loop, and one FPH for each function call in the loop. E'_L is the set of edges between B'_L. An HFG has either an LPH or an FPH to denote if it is above the highest level loop, or one of the loops in a function.*

Algorithm 1 explains how to separate nested loops into different graphs. The algorithm starts by copying all blocks and edges to the sets B'_L and E'_L respectively in lines 1 and 2. It then cycles through all nested loops that are at the first level of nesting below the loop L. It finds a nested loop K, as K is a proper subset of L and its nesting level is one more than L, so in lines 4 and 5 it removes the blocks and edges that are in K and also in L'. In line 6 the algorithm examines each edge in the original loop L; if the head or tail is in K, then the edge is removed from L' and a new edge is added to L'. The new edge connects to a new node that is a LPH, where the one entry edge to the loop K now connects to LPH and the exit from K, also connects to LPH. The complexity of Algorithm 1 is dependent on the number of loops that are nested within one level of another loop. Therefore the time complexity of this algorithm would be $O(n * b)$, where n is the number of loops in the outer loop, and b is the number of blocks in the largest loop.

Figure 5 (a) shows how the example CFG in Figure 4 will become several graphs after applying algorithm 1. In the original CFG there are 3 nested loops at different levels, and each corresponding HFG gets a LPH to represent the blocks and edges that were there.

What is needed to move beyond this stage is a forest of DAGs, so that the HFG information can be used to build a more condensed graph. The first step is to remove any back edges, and to add a new root block. If the HFG is a loop, its root block becomes a LPH, and if it is the highest level HFG, its root block becomes a FPH. Figure 5 (b) shows the forest of DAGs after this final transformation.

4.1.3 Step 3: Building Call Control Flow Graph

We must traverse the HFGs in a Depth First Search (DFS) and build a graph, that condenses the information present in the HFG, into a graph called the Call Control Flow Graph (CCFG). A CCFG is a proper subset of GCCFG, it is constructed when a block of interest is found on a HFG, we then apply a set of rules to construct the proper vertexes and edges in the CCFG. A block of interest is a block with two outgoing edges, a LPH, a FPH, or a block with a function call in its code.

Definition 4. *(**Call Control Flow Graph**) A CCFG $G' = (V', E')$, where given a GCCFG $G \Rightarrow G' \subset G$. V' is a set of vertices representing program structures of a loop, function call, or if-then-else. E' is the set of edges connecting the program structures.*

Algorithm 2 and Algorithm 3 explain how to build a CCFG given the information present in an HFG. First, Algorithm 2 gives three of the rules, for building a CCFG, by showing the cases for building verticies. The first rule, is found at line 2, where there is a condition in the program, therefore a condition vertex is added to the set of vertices, in the CCFG G'. The second rule, is found at line 5 where a loop is found, then a loop vertex is added to the vertex set in G'. Finally, if the block contains a call to another function or is a FPH

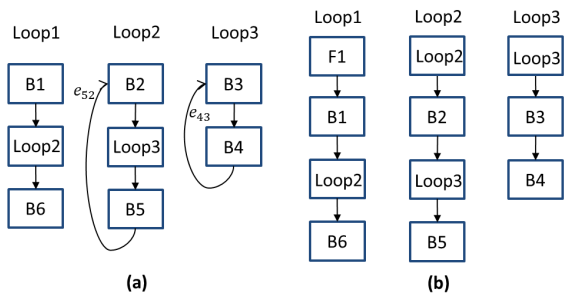


Figure 5: *(a) HFGs extracted from the CFG in Figure 4 after applying algorithm 1. (b) A forest of DAGs after all the HFGs have been transformed.*

Algorithm 3: Depth First Search HFG

Input: A Basic Block of an HFG $b_1 \in B'_L$, A CCFG G'
Output: A CCFG G'

```
1  if (OutDegree(b1) := 2) ∧ (m : b1 ∈ V′) then
2      forall the p ∈ P|p := {b1, ..., bn} do
3          ConvergeBlock = HighestCommonDescendant(P)
4          forall the b ∈ p|(b > b1) ∧ (m : b ∈ V′) do
5              if b ≤ ConvergeBlock then
6                  E′ ← E′ + {e = (m : b1, m : b)}
7                  if TrueCondition(b1) then
8                      AddTrueSet(m : b1,e)
9                  else
10                     AddFalseSet(m : b1,e)
11             else
12                 v = Parent(m : b1)
13                 if v ∉ Vc then
14                     E′ ← E′ + {e = (v, m : b)}
15 forall the b2 ∈ Children(b1) do
16     DepthFirstSearch(b2,G′)
```

at line 8 then a function vertex is added to the vertex set in G'. At lines 4, 7, and 10 a mapping between the block in L' and the vertex in G' is created, as shown in Figure 6 (this will be necessary later). The complexity of Algorithm 2 is based on the number of blocks in an HFG. Each block must be examined, and in the worst case, each line of code in the block must be examined to determine if there is a function call. Therefore the complexity is $O(b * l)$, where b is the number of blocks in an HFG and l is the number of lines in the largest block.

Algorithm 3, called from algorithm 2 is a recursive function, which describes the remaining rules for building CCFG. First at line 1 we locate a condition, that also is mapped to a vertex in V'. Then, we examine all true and all false paths through the graph, that appear after the condition diverges and before it converges. The fourth rule for creating a CCFG appears at line 6 and line 7, where if another block mapped to the CCFG is found then a true edge is added to the CCFG. The fifth rule appears at line 6 and line 9, which like the previous rule adds an edge, this time is a false edge in the CCFG. This case is illustrated in Figure 7, where false edges are created. The final rule, for building GCCFG, appears at line 13 where the block of interest appears after the condition converges. In this case an edge is created between the node and the parent of the condition. This case is illustrated by Figure 8 where the block of interest is after the convergence so the edge is created with the parent of the condition. The complexity for Algorithm 3 is $O(b * p)$, where b is the number of blocks in an HFG and p is the largest number of paths starting at a condition block in the HFG. The Highest Common Descendant has a complexity of $O(h)$, where h is the height of the graph. In our algorithm we store the traversal information from the highest common descendant algorithm to build all of the paths in the graph, therefore the complexity is affected by the largest path and not the height of the graph.

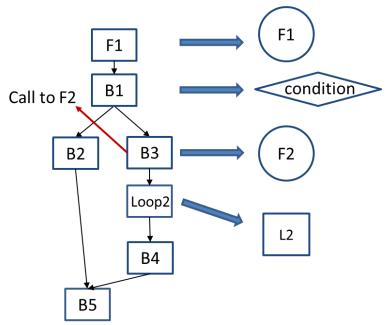


Figure 6: *Basic Blocks are mapped to vertices in the CCFG. A FPH is mapped to a circle vertex, a LPH is mapped to a square vertex, and a if-then-else condition is mapped to a diamond vertex.*

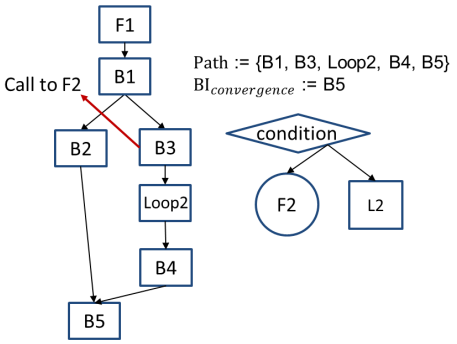


Figure 7: *Edges in a CCFG are constructed by finding the path from a condition block to a convergence block for that condition in the HFG. Along that path there are other FPH, LPH, and condition blocks.*

4.1.4 *Final step: Integrating Call Control Flow Graph*

After we have built a CCFG for every HFG in the program, we can glue the CCFG's together. Figure 10 illustrates how this is done. If there are loop vertices in two CCFGs with the same label then they become one vertex, gluing the two graphs together. This same gluing technique is applied if both are function vertices. Once all CCFGs have been glued together there will be one large graph, which is a GCCFG.

4.2 Challenges cases in GCCFG construction

Till now, we have explained the definition and construction of GCCFG in a typical setting. However, several times the input program graphs are ill-formed, and that makes the task of building GCCFG challenges. Challenges cases include a program that is constructed with poorly formed loops, a program which contains switch statements, finding the convergence point of some conditions, how to represent

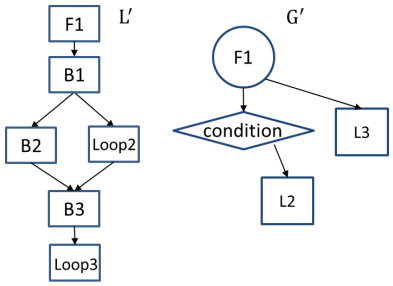


Figure 8: *In a HFG L' the block of interest may appear after a condition's convergence block. In the CCFG G' the vertex it is mapped to must become the child of the condition vertices parent.*

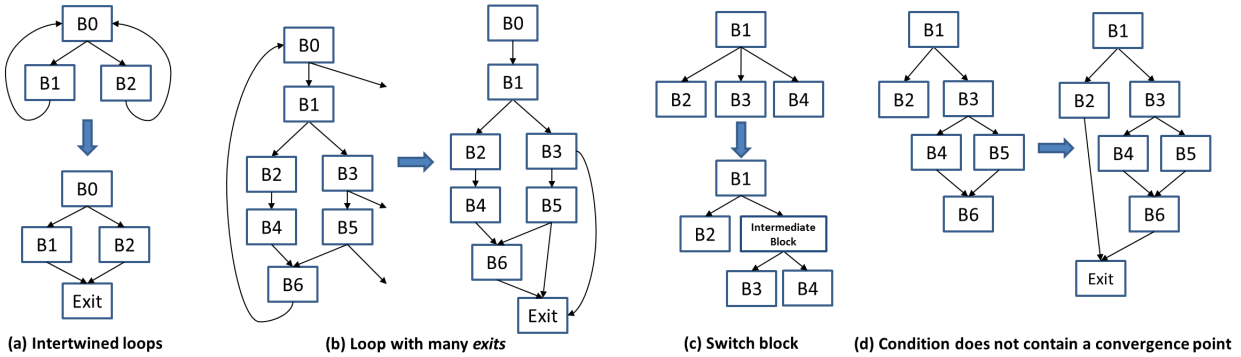


Figure 9: ***Challenges and Solutions****: (a) Two intertwined loops on the top graph, while the bottom graph shows both loops become one well defined loop. (b) A loop with many exits on the left graph, while the right graph shows a well defined loop with only one exit point. (c) A switch block on the left is transformed to the graph on the right, where each block has at most two children. (d) A condition does not have a convergence point in the graph on the left, so a new one is created as an exit block from the HFG in the graph on the right.*

recursive procedures, and how to represent function pointers. These problems must be addressed to be able to successfully build GCCFG.

4.2.1 Poorly formed loops

The first challenge to address is that of *poorly formed loops*. These include, loops that have multiple exits (commonly caused by continue , break, and goto statements), and intertwined loops (caused by goto statements). Both of these types of loop problems must be removed before transforming the basic blocks into a final HFG. Figure 9 (a), top graph, shows an example of an intertwined loop. This loop can return to its head block from either loop body, making which loop body executes non-deterministic. Figure 9 (b), left graph, shows a loop with many exits. It cannot be determined which basic block will execute immediately following the conclusion of the loop. The process for dealing with this challenge is the same for either case, we must create a unique entry and exit point for a loop. Figure 9 (a), bottom graph, shows how a unique exit block is added to the graph and all back edges are attached to it, effectively making two loops into one. Figure 9 (b), right graph, shows how adding a unique exit block and having all exit edges point to it, makes the loop exit deterministic, while maintaining the control structure of condition blocks. The transformation illustrated in Figure 9 (a) loses the information about all but one of the loops intertwined with others. However, none of the control flow information resulting from if-then-else statements is removed. Since one of the useful traits of GCCFG is determining which loops are nested within other loops, saving intertwined loops does not provide useful information. Therefore the transformation illustrated in Figure 9 (a) saves the relative information that the blocks that were part of the intertwined loops are still part of a loop, and conserves all other control flow information. The transformation in Figure 9 (b) does not lose any control flow or loop related information.

4.2.2 Switch statements

The second challenge to address is programs with *switch* statements. While switches are not poor programming practice the challenge is that you cannot break a single block's children down into true and false children. We apply a transformation in our technique to distinguish true and false children by adding an intermediate block to the graph. The top part in Figure 9 (c) shows what a sub graph of a CFG would look like where B_1 has all of the switch conditions, and B_2 through B_4 has each case for the switch. The graph on the bottom side of Figure 9 (c) shows how the graph looks after transformation. All children of the switch except the left-most one are removed and an intermediate block takes their place. The removed children become descendants of the intermediate block, and if there are more than two children left, this process is repeated until each block has at most 2 children. No information will be lost in the process of adding intermediate blocks as this is simply a place holder without any code in it. It also doesn't add or take away any control flow information as the number of edges leading to each condition of the switch hasn't changed.

4.2.3 Finding a convergence point

Another challenge to address is finding the convergence point of a condition with exit statements. These are mostly caused by error conditions that exit the program immediately. In the representing CFG the block with the exit has no descendants, so it is not clear what the corresponding convergence block would be. If there are nested conditions within this condition on the true or false paths it further confuses the issue as the convergence point of the nested condition may appear to occur after the convergence of parent condition. The solution to this challenge is similar to the loop problem in that each CFG must have a unique exit block. Figure 9 (d) shows a graph with an exit at B_2, most likely caused by an error check in the program, while the graph on the right shows how the block now has an edge pointing to a unique exit. This can cause a control flow problem in this program now as it appears that an error may in fact allow execution to continue, but we can annotate the edge and block so that if a block of interest occurs after the

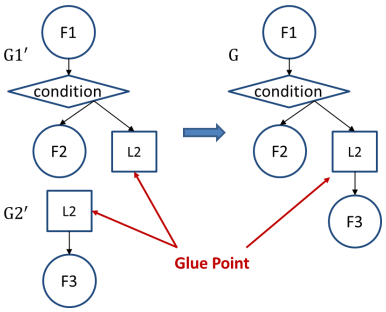


Figure 10: *Two CCFG's G'_1 and G'_2 are glued together at a common LPH to make a GCCFG G.*

error it will have less weight in the final GCCFG. Adding a new edge from a program exit to a new place holder exit doesn't change the program as the exit block is simply a place holder and doesn't have any useful information inside of it. There is no added control flow information as only one outgoing edge is added to the program exit block. This transformation just makes HFG graph traversal much easier when finding blocks of interest.

4.3 Recursion and Function Pointers

Up to this point call sites have represented an edge from one vertex to a function vertex. However, when a program contains recursion we need to be able to represent that control has been given back to the entry point of the recursive procedure. To continue with the trend in GCCFG of having a unique function vertex for control to move to, would not adequately represent the control flow contained within the recursive procedure, or would require that we duplicate all of this information in the graph. Therefore, we introduce a back edge in GCCFG. Any back edge in the graph represents a recursive procedure call, where the edge starts at the call site and ends at the recursive procedures function vertex. It is important to note, that there is no structure in Global CFG to handle recursive programs. The Global CFG requires that when a function call occurs a functions CFG is inserted in its place, and this is not possible in Global CFG.

Determining the set of functions that a pointer can point to requires program analysis, where the most conservative results, which run quickly, will give a much larger set than a more accurate analysis, which will take a longer time. The trade-off lies in choosing between accuracy and speed for pointer analysis. GCCFG needs to be generated quickly as a benefit to doing more data management analysis at compile time, and needs to be succinct for those analyses. Our technique uses the pointer analysis presented in [21], where a less accurate model gives enough information to determine which pointers will be equivalent at compile time and placing the corresponding functions and their pointers into an equivalence class. This relationship between pointers and the functions they may point to is used to generate an edge between the call site where the pointer exists and the functions in the equivalence class. In GCCFG these special edges will be represented by dotted lines.

5. EXPERIMENTS

5.1 Experimental Setup

We perform experiments to demonstrate the need, and usefulness, of GCCFG over Global CFG. The experiments are for code management, and stack data management optimizations in SMM architectures. To do that, we implement our technique to construct GCCFG in LLVM compiler. Since a pass can only be applied on a single source code file, we use the llvm-link utility to transfer several bitcode files into one bitcode file. We implement a FunctionPass in LLVM, which operates on each function of a program. The function pass extracts control flow and loop information from each function and stores it in a data structure. After all the passes have finished the extracted information is combined into a GCCFG. GCCFG nodes and edges are annotated with information necessary for code and stack data management. For comparison purpose, we also implemented the generation of Global CFG. The code and stack management implementations get information about the program from GCCFG (or Global CFG) through some functions, like $estimateInterferenceCost$ that can be computed using both GCCFG and GlobalCFG. We then run LLVM passes for code [4] and stack data management [20]. We run our compiler on benchmarks from the Mibench suite [15] to compare the compilation time.

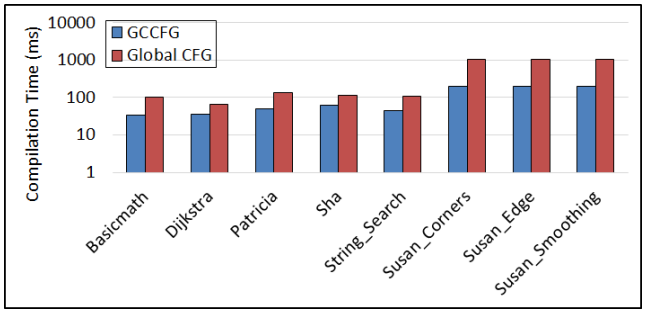


Figure 11: *The compilation time when applying Code Mapping to benchmarks using GCCFG vs. Global CFG as input.*

5.2 GCCFG makes code management 5X faster as compared to Global CFG

Figure 11 plots the time (in milliseconds) to perform code mapping on our benchmarks using GCCFG and Global CFG. The results show that across all the benchmarks we get a consistent speedup of around 5X. To dig deeper in where the benefits are coming from, we measure the average amount of time it takes to do a single step of code management [4] for each benchmark. Initially all the functions are mapped to their own sections. Then in each step the functions in two regions are merged to make one region. The two regions that will cause the least increase in the data transfers between the global main memory and the local SPMs must be selected for merging. This must be done until the code space constraint is met. In each step, the code mapping algorithm needs the estimate of data transfers between the global main memory and the local SPMs for a given mapping, called interference, and that is calculated using using GCCFG (by algorithm 4) and using Global CFG (by algorithm 5). We see that on average a single pass of code management Global CFG takes 8X longer vs. GCCFG. This is caused by the fact that on each step of Code Management the necessary information must be extracted from the graph before moving on to the next step. Since Global CFG is so much larger it takes much longer than GCCFG to extract this information.

5.3 GCCFG makes stack management 4X faster as compared to Global CFG

Figure 12 plots the runtimes of the stack management with GCCFG and Global CFG. In the case of stack mapping the algorithm runs on a reduced graph with only vertices representing functions, therefore Global CFG must be reduced at every iteration where the data structure must be traversed. Therefore, we see an average speedup of 4X using GCCFG over Global CFG. To dig deeper into this, we again compute the time for each step of stack management. For the sake of space we omit the details of the algorithms for performing Stack Management using GCCFG and Global CFG, and only present the results. However, in a single pass of stack management, stack frames are combined together into sets on a single path of execution. Then

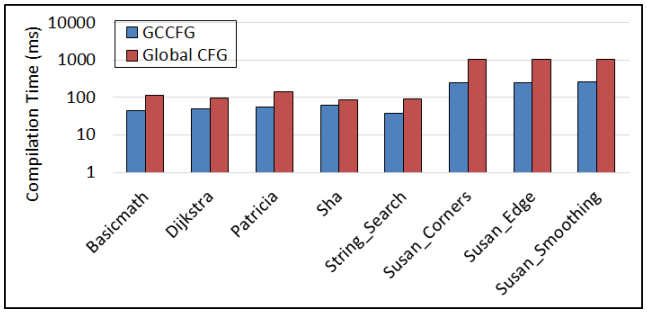


Figure 12: *The compilation time when applying Stack Management to benchmarks using GCCFG vs. Global CFG as input.*

the set is modified slightly to determine if a different set of stack frames will provide a more efficient execution time for the application. On average a single pass takes 8X longer with Global CFG than it does with GCCFG. The reason is again because Global CFG is much bigger than GCCFG, and Stack Management needs to extract the necessary information from Global CFG before it can move on to the next step.

5.4 GCCFG is succinct representation of the program

Figure 13 plots the number of nodes in GCCFG and Global CFG for the benchmarks. We can see that on average Global CFG is 9X larger than GCCFG, which is due to the fact that Global CFG has nodes that represent stops along paths that lead to code that is not a function. Global CFG also has nodes that represent sequential intermediate parts of a program, while GCCFG only has nodes that represent control flow information leading to function calls. However Global CFG is very quick to build – almost $200X$ faster. That is because to construct Global CFG we only need to connect an edge from a function call site to a functions first basic block. However, even GCCFGs for all the benchmarks are built in less than a second (while code management for example, can take tens to hundreds of seconds), so the build time is not so important.

6. COMPLEXITY ANALYSIS OF USING GCCFG AND GLOBAL CFG

In this section we look at the algorithms to generate the information required for code and stack data management, and compare the complexity of those algorithms. That will give us insight into how succint a representation GCCFG is.

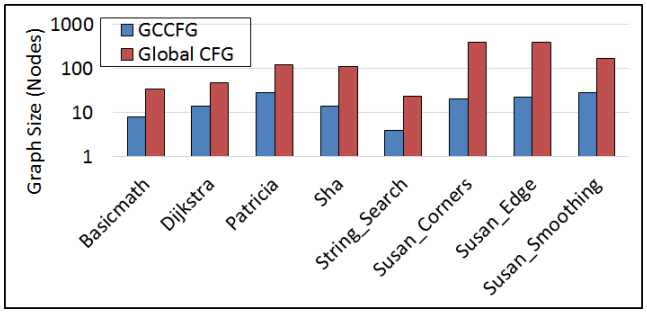


Figure 13: *The size (number of nodes) of GCCFG is about 9X less than Global CFG, for our benchmarks.*

Algorithm 4: Interference GCCFG

```
Input: A GCCFG G = (V, E), M
Output: TotalInterference
1 foreach i ∈ V_f do
2     foreach j ∈ V_f : i ≠ j ∧ M[i] = M[j] do
3         if i := LCA(i, j) then
4             TotalInterference+ = Cost(i) +
              Cost(FunctionsOnPath(i, j)_1)
5         else if j := LCA(i, j) then
6             TotalInterference+ = Cost(j) +
              Cost(FunctionsOnPath(j, i)_1)
7         else
8             TotalInterference+ = Cost(LCA(i, j))
```

6.1 Interference calculation using GCCFG

Algorithm 4 shows how code mapping determines the total interference of a program based on a mapping of functions to regions [9]. Interference is the amount of data transfers that will take place between the local SPM and the global main memory for a given mapping. The input M is a mapping of functions to regions. At lines 1 and 2 the algorithm iterates over all pairs of functions in the program that are mapped to the same region in memory. At lines 3 and 4 if function i is the Lowest Common Ancestor (LCA) of the two functions then the total interference will have the number of times the function is called during execution plus the number of times the first function on the path from i to j is called. Lines 5 and 6 do the same thing as 3 and 4 except j is the LCA of the two functions. Finally, at line 8 if neither i or j are the LCA of the other, the number of times the actual LCA of the two functions is executed is added to the total interference. The total running time of GCCFG would be $O(\frac{n*l}{2} * n^2) \Rightarrow O(n^3)$, where n is the number of function vertices, and l is the number of loop nodes in the GCCFG. Note that we will need to traverse the height of the graph twice to find the LCA of two given nodes.

6.2 Interference calculation using Global CFG

Algorithm 5 shows how to determine the interference cost if we used the Global CFG. To calculate the total interference between any two functions in a program there are two main loops at lines 1 and 3. First we must cycle through every basic block in the whole program until we find one i that is a entry to or exit from a function, as this is where a swap can occur in memory. Now we need to find another block j that is an entry into a function, mapped to the same region as i, and i and j are in separate functions. At line 5 we then need to do a depth first search to find the state of the memory (active functions in memory) and compare that to the function containing i. This means we have a conflict that will increase the cost of the interference. Line 7 determines if both blocks are inside a loop, because this will increase the total cost by the number of times a loop iterates during the execution of the program. Otherwise we only add the cost of the number of times blocks i and j are executed to the total interference cost. The total running time for Algorithm 5 is $O(b^2 * b + 2L)$, where b is the number of basic blocks in a program, and L is the maximum number of basic blocks in a loop. b can be approximated as $n * B$, where n is the number of functions in the whole program and B is the maximum number of basic blocks in a function. Further, in a Global CFG, if a function is called

Algorithm 5: Interference Global CFG

```
Input: A GlobalCFG P = (B, E) : H = (B', E') ⊆ P, M
Output: TotalInterference
1  foreach i ∈ B do
2      if Entry(i) ∨ Exit(i) then
3          foreach j ∈ B do
4              if Entry(j)
               ∧(M[H_1] = M[H_2] : i ∈ B'_1 ∧ j ∈ B'_2 ∧ H_i ≠ H_j)
               then
5                  MEM = {DFS(P, i)}
6                  if {H_1 : i ∈ B'_1} ∉ MEM then
7                      if LoopHead(i) := LoopHead(j) then
8                          TotalInterference+ =Cost(i) +
                           Cost(j) ∗LoopCost
9                      else
10                         TotalInterference+ =Cost(i)+
                           Cost(j)
```

multiple times then it is necessary to make a copy of the basic blocks for each call and inline those blocks into the graph. We can represent this inline factor as c which is a multiplication factor for how many times a function is inlined. Therefore the total running time for Algorithm 5 is actually $O([n * B * c]^3)$ compared to $O(n^3)$ for Algorithm 4.

7. SUMMARY

Since coherent caches architectures will not scale for long, researchers are on the lookout for a new memory architecture that can scale for hundreds and thousands of cores. Software Managed Manycore (SMM) architectures – in which each core has only a scratch pad memory is a promising solution, since hardware are simpler, scalable and more power-efficient. However, in SMM architectures the code and data of the tasks must be explicitly managed in the software by the compiler. State-of-the-art compiler techniques for SMM architectures require inter-procedural information and analysis, and they have used GCCFG (Global Call Control Flow Graph) for that. GCCFG is a whole program representation that captures the control-flow as well as function call information in a succinct way. However, how to construct GCCFG has not been shown yet. We find that there are several commonly occurring cases where constructing GCCFG is not so straightforward. This paper develops graph transformations that allow us to correctly construct GCCFG in (almost) all cases. Our experiments show that by using succinct representation (GCCFG) rather than elaborate representation (GlobalCFG), the compilation time of state-of-the-art code management technique [4] can be improved by an average of 5X, and that of stack management [20] can be improved by an average of 4X.

8. REFERENCES

[1] Raw Performance: SiSoftware Sandra 2010 Pro (GFLOPS).

[2] Intel Core i7 Processor Extreme Edition and Intel Core i7 Processor Datasheet, Volume 1. In *White paper*. Intel, 2010.

[3] K. Bai, D. Lu, and A. Shrivastava. Vector Class on Limited Local Memory (LLM) Multi-core Processors. In *Proc. of CASES*, pages 215–224, 2011.

[4] K. Bai, J. Lu, A. Shrivastava, and B. Holton. CMSM: An Efficient and Effective Code Management for Software Managed Multicores. In *Proc. of CODES+ISSS*, pages 1–9, 2013.

[5] K. Bai and A. Shrivastava. Heap Data Management for Limited Local Memory (LLM) Multi-core Processors. In *Proc. of CODES+ISSS*, pages 317–326, 2010.

[6] K. Bai and A. Shrivastava. A Software-Only Scheme for Managing Heap Data on Limited Local Memory (LLM) Multicore Processors. *Trans. on Embedded Computing Sys.*, 13(5):472–511, 2013.

[7] K. Bai and A. Shrivastava. Automatic and Efficient Heap Data Management for Limited Local Memory Multicore Architectures. In *Proc. of DATE*, 2013.

[8] K. Bai, A. Shrivastava, and S. Kudchadker. Stack Data Management for Limited Local Memory (LLM) Multi-core Processors. In *Proc. of ASAP*, pages 231–234, 2011.

[9] M. A. Baker, A. Panda, N. Ghadge, A. Kadne, and K. S. Chatha. A Performance Model and Code Overlay Generator for Scratchpad Enhanced Embedded Processors. In *Proc. of CODES+ISSS*, pages 287–296, 2010.

[10] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad Memory: Design Alternative for Cache on-chip Memory in Embedded Systems. In *Proc. of CODES*, pages 73–78, 2002.

[11] G. Bournoutian and A. Orailoglu. Dynamic, Multi-core Cache Coherence Architecture for Power-sensitive Mobile Processors. In *Proc. of CODES+ISSS*, pages 89–98, 2011.

[12] B. Choi et al. DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism. In *Proc. of PACT*, pages 155–166, 2011.

[13] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The Program Dependence Graph and its use in Optimization. *ACM TOPLAS*, 9(3):319–349, 1987.

[14] B. Flachs et al. The Microarchitecture of the Synergistic Processor for a Cell Processor. *Solid-State Circuits, IEEE Journal of*, 41(1):63–70, 2006.

[15] M. R. Guthaus et al. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In *Proc. of the Workload Characterization*, 2001.

[16] M. Heinrich et al. A Quantitative Analysis of the Performance and Scalability of Distributed Shared Memory Cache Coherence Protocols. *IEEE Trans. Comput.*, 48(2):205–217, Feb. 1999.

[17] S. Horwitz, T. Reps, and D. Binkley. Interprocedural Slicing using Dependence Graphs. *ACM Trans. Program. Lang. Syst.*, 12(1):26–60, 1990.

[18] C. Jang, J. Lee, B. Egger, and S. Ryu. Automatic Code Overlay Generation and Partially Redundant Code Fetch Elimination. *ACM Trans. Archit. Code Optim.*, 9(2):10:1–10:32, 2012.

[19] S. Jung, A. Shrivastava, and K. Bai. Dynamic Code Mapping for Limited Local Memory Systems. In *Proc. of ASAP*, pages 13–20, 2010.

[20] J. Lu, K. Bai, and A. Shrivastava. SSDM: Smart Stack Data Management for Software Managed Multicores (SMMs). In *Proc. of DAC*, 2013.

[21] A. Milanova, A. Rountev, and B. G. Ryder. Precise Call Graphs for C Programs with Function Pointers. *Automated Software Engineering*, 11(1):7–26, 2004.

[22] C. D. Polychronopoulos. The Hierarchical Task Graph and its use in Auto-scheduling. In *Proc. of Supercomputing*, pages 252–263, 1991.

[23] S. Udayakumaran, A. Dominguez, and R. Barua. Dynamic Allocation for Scratch-pad Memory using Compile-time Decisions. *Trans. on Embedded Computing Sys.*, 5(2):472–511, 2006.

[24] J. Whitham. Optimal Program Partitioning for Predictable Performance. In *Proc. of ECRTS*, 2012.

[25] Y. Xu, Y. Du, Y. Zhang, and J. Yang. A Composite and Scalable Cache Coherence Protocol for Large Scale CMPs. In *Proc. of ICS*, pages 285–294, 2011.