CMSM: An Efficient and Effective Code Management for Software Managed Multicores

Ke Bai, Jing Lu, Aviral Shrivastava and Bryce Holton
Compiler Microarchitecture Laboratory
Arizona State University, Tempe, Arizona 85287, USA
Email: {Ke.Bai, Jing_Lu, Aviral.Shrivastava, Bryce.Holton}@asu.edu

***Abstract*—As we scale the number of cores in a multicore processor, scaling the memory hierarchy is a major challenge. Software Managed Multicore (SMM) architectures are one of the promising solutions. In an SMM architecture, there are no caches, and each core has only a local scratchpad memory. If all the code and data of the task mapped to a core do not fit on its local scratchpad memory, then explicit code and data management is required. In this paper, we solve the problem of efficiently managing code on an SMM architecture. We extend the state of the art by: i) correctly calculating the code management overhead, ii) even in the presence of branches in the task, and iii) developing a heuristic CMSM (Code Mapping for Software Managed multicores) that results in efficient code management execution on the local scratchpad memory. Our experimental results collected after executing applications from MiBench suite [1] on the Cell SPEs (Cell is an SMM architecture) [2], demonstrate that correct management cost calculation and branch consideration can improve performance by 12%. Our heuristic CMSM can reduce runtime in more than 80% of the cases, and by up to 20% on our set of benchmarks.**

Keywords*—*Code, instruction, local memory, scratchpad memory, SPM, embedded systems, multi-core processor

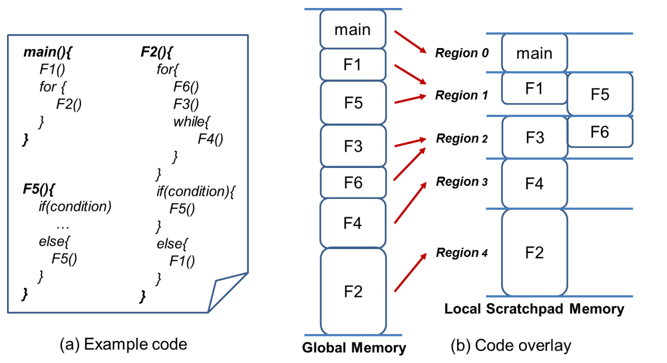
I. Introduction

We are in a transition from multicore processors to many-core processors. While scaling the number of cores is relatively straightforward, scaling the memory hierarchy is a major challenge. Most experts believe that fully cache-coherent architectures will not scale when there are hundreds and thousands of cores, and therefore architects are looking for alternative scalable architecture designs. Recently, a 48-core non-coherent cache architecture named Single-chip Cloud Computer (SCC) was manufactured by Intel [3]. The latest 6-core DSP from Texas Instruments, TI 6472 [4] features non-coherency cache architecture. But caches still consume a large portion of power and die area [5]. A promising option for an even more power-efficient and scalable memory hierarchy is to not have caches, but only scratchpad memories. Scratchpad memories are raw memories that do not have any tags and lookup logic. As a result, they consume approximately 30% less area and power than a direct mapped cache of the same effective capacity [5]. Therefore, such scratchpad based multicore architectures have the potential to be more power-efficient and scalable than traditional cache-based architectures.

However, this improvement in power-efficiency comes at the cost of programmability. Since there is no data management implemented in the hardware in these scratchpad based multicore architectures, data must be managed by the application in software. This means that the data that the application will require must be brought into the local scratchpad memory using a Direct Memory Access (DMA) command before it is used, and can be evicted back to the main memory (also using the DMA command) after it is used. Due to this explicit need of data management in software, these processor designs are termed, Software Managed Multicore (SMM) architectures. A very good example of SMM memory architecture is the Cell processor that is incorporated in the Sony Playstation 3. The Synergistic Processing Elements (SPEs) in the Cell processor have only scratchpad memories. Its power efficiency is around 5 GFlops per watt [2].

SMM architecture is a truly "distributed memory architecture on-a-chip." Applications for it are written in the form of interacting tasks. The tasks are mapped to the cores of the SMM architecture. Each execution core can only access its local scratchpad memory, and to access other local memories or the main memory, explicit DMA instructions are required in the program. The local memory is shared among code, stack data, global data and heap data of the task executing on the core. How to manage the task data on scratchpad memory of the cores is an important problem that has drawn significant attention in recent years [6]–[14]. While management is needed for all code and data of the task when they cannot fit in the local memory, in this paper we will focus on the problem of code management, since efficient code management can be of considerable significance to the performance of the system. The first step in code management is to assign some space in the local scratchpad memory for managing code. Then we divide this space into regions, and functions in the program are mapped to these regions. Functions mapped to the same region are compiled and linked starting with the same start address (that of the region). At runtime, only one function out of the ones that are mapped to the region can be present in the region. At each function call, it is checked whether the function being called is present in the region or not. If not, it is fetched from the main memory using a DMA command [15]. Therefore, the size of region is equal to the size of the largest function mapped to the region, and the total code space required is the sum of the sizes of the regions. Given some space on the local memory, the code management problem is to i) divide the code space into regions, and ii) find a mapping of functions to regions, so that the management overhead is minimized. We estimate the memory management overhead to be proportional to the size of code that needs to be transferred between the local memory and the main memory.

This research was funded by grant from National Science Foundation CCF-0916652.



```
main(){
  F1()
  for {
    F2()
  }
}

F5(){
  if(condition)
  ...
  else{
    F5()
  }
}

F2(){
  for{
    F6()
    F3()
    while{
      F4()
    }
  }
  if(condition){
    F5()
  }
  else{
    F1()
  }
}
```

Fig. 1: ***Code mapping problem on scratchpad memory*** *- when task assigned to the execution core has larger footprint than the available space, code needs to be mapped between external main memory and the local scratchpad memory of the core.*

Finding the number of regions and mapping the functions to regions that will minimize instruction transfer, both have been proven to be intractable [6], [16]. Therefore, several heuristics have been proposed [6]–[8], [12]. A key component of code management schemes is a function that given a mapping of functions to regions, estimates the overhead of code management, or more precisely the amount of memory that will be transferred. This cost function "drives" the technique to map function into regions, and in a sense, the function mapping technique can only be as good as the accuracy of this cost estimation function. Unfortunately, all previous techniques have estimated this overhead of code management inaccurately. We identify three limitations in the state-of-the-art mechanisms to accurately estimate the code management overhead. The limitations are: i) Assumption that the code management overhead when two functions are mapped to the same region is independent of where the other functions are mapped. This is incorrect, because as we show in this paper, the overhead depends on which other functions are mapped to the same region as the two functions. ii) Several cases of inaccurate estimation, e.g., when a function is called at several places in the program. iii) Ignoring the effect of branches. Most previous schemes broadly assumed that both the branch paths are executed. As a result, the estimation of the number of times a function is called inside a branch increases instead of decreasing. This leads to inaccurate code management overhead estimation – especially when branch probability is quite skewed (e.g., 90% or 10%). The inaccuracies in the code management overhead caused due to these limitations, although do not result in wrong results, they do result in bad mapping decisions and therefore sub-optimal performance. We address problems aforementioned and make the following contributions:

1) Correct code management overhead calculation is proposed. It properly considers all circumstances leading to instruction transfer, which cannot be handled in previous works.
2) Our cost calculation considers branches in the program. We assert the consideration of branch is very important and inconsideration of branch in cost calculation in previous works is one of the reasons that leads to inefficient code mapping.
3) A fast polynomial time heuristic for Code Mapping on Software Managed multicore processors (named as CMSM) is proposed. It takes in graphical code representation of the program, and then transforms the representation to cost calculation graph, from which management cost is calculated and code mapping is ultimately executed.

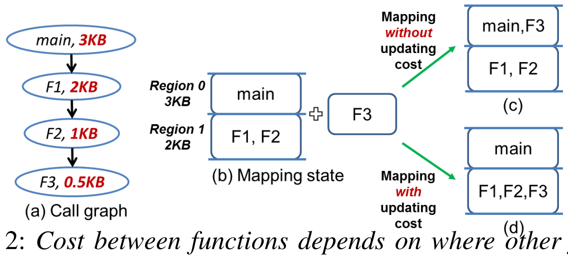


Fig. 2: *Cost between functions depends on where other functions are mapped, and updating the costs as we map the functions can lead to a better mapping.*

We establish the effectiveness of the proposed cost calculation and mapping techniques by executing benchmarks from the MiBench suite [1] on the SPEs of the Cell processor [2], and comparing with the closest approach [7]. We find that when we use correct estimation of code management overhead, even in the previous code management technique, it leads to a mapping that performs 12% better. On top of that, our CMSM heuristic can reduce runtime in more than 80% of the cases, and by up to 20% on our set of benchmarks.

II. Background and Motivation

To run an application with larger code size than what is available on the local scratchpad memory, typically the code overlay scheme is leveraged [15]. Usually, the overlay organization is generated manually by programmers or automatically by a specialized linker. A good code overlay requires deep understanding of the program structure, with the consideration of maximum memory savings and minimum performance degradation. As in Fig. 1, code overlay organization needs to determine both the number of regions and the mapping of functions to regions. Functions mapped to the same region are located in the same physical address, and must replace each other during run time (by instruction fetching function *__ovly_load()* before each function call) [15]. Therefore, the size of a region is the size of the largest function in the region. The total code space required is equal to the sum of the sizes of regions. From the performance perspective, it is best to place each function into a separate region, so that it will not interfere with any other object, but that may increase the code space too much. In contrast, mapping all functions into one region uses the minimum amount of code space, but incurs too many instruction transfers and therefore runtime overhead. *The task of optimizing code mapping is, to organize the application functions into regions that will obtain a balance between the code space used and the data transfers required.*

The work [6]–[8], [12] provide heuristics for code mapping on SMM architectures. However, they are all not efficient enough, which prevents scratchpad memory becoming a competitive alternate of cache on multicore processors in terms of performance. The inefficiency is mainly because of inaccurate or incorrect cost calculation. There are three reasons for their inaccuracies:
i) **No updating of cost calculation**: Previous work [6], [8],

[12] didn't update the cost dynamically. It is incorrect and can lead to inferior mapping. Fig. 2 (a) shows an example where function *main* calls F1, F1 calls F2, and F2 calls F3, and then they all return. The function nodes also indicate the sizes of each of the functions. Let us consider a case which requires us to map all functions into a scratchpad memory of 5 KB. It is slightly tricky to calculate the cost between indirect function calls. For example, when compute the cost between *main* and F2, if *main* and F2 are mapped to the same region, the interference[1] between them depends on where F1 is mapped. If F1 is mapped to another different region, then the interference between *main* and F2 is just sum of their sizes, namely, 3 KB + 1 KB = 4 KB. The calculation is as follows. When F2 is called, 1 KB of function F2 will need to be brought into the memory. When the calling state returns to *main*, 3 KB of the code of *main* needs to be brought into the scratchpad. However, if all *main*, F1 and F2 are mapped to the same region, then the interference cost between *main* and F2 is 0. This is because, when F2 is called, *main* is already replaced with F1, and when the program returns to *main*, F2 is already replaced. In a sense, there is interference between *main* and F1, and between F1 and F2, but there is no interference between *main* and F2. Previous approaches [6], [8], [12] computed the worst case interference cost, i.e., 4 KB for *main* - F2, and never updated it, and therefore obtained inferior mapping. To explain this, Fig. 2 (b) shows a state in mapping when *main*, F1 and F2 have already been mapped. *main* is alone in region 0, F1 and F2 are together in the region 1. Now is the time to map function F3. Size of F3 is 0.5 KB, therefore it can be mapped to either region, without violating the size constraint. The interference cost between region 0 and F3, i.e., between *main* and F3 is 3.5 KB. The interference cost between region 1 and F3 is traditionally computed as the sum of interferences between the functions in region 1 and F3, i.e., 2.5 KB between F1 and F3, and 1.5 between F2 and F3, totalling to 4 KB. Consequently traditional techniques will map F3 to region 0 with *main* (shown in Fig. 2 (c)). Clearly there is a discrepancy in computing the interference cost between region 1 and function F3. If F2 is also mapped to the same region, the interference cost between F1 and F3 should be estimated as 0. Otherwise, the interference cost between region 1 and function F3 are incorrectly (over)estimated. With this fixed, the interference between region 1 and F3 is just the interference between F2 and F3, which is just 1.5 KB. As per this correct interference calculation, F3 should be mapped to region 1 (shown in Fig. 2 (d)). The required total data transfer between the main memory and the local memory, in this case $9.5 = 3 + (2 + 1 + 0.5 + 1 + 2)$ KB, as compared to $11.5 = (3 + 0.5 + 3) + (2 + 1 + 2)$ KB with the previous mapping, resulting in a 18% savings in data transfers.

ii) **Incorrect cost calculation**: The mapping schemes in [6]–[8], [12] are all aimed to reduce the management cost. But none of them has proposed a correct cost calculation scheme. There are two problems with their methods. On one hand, none of their methods is general enough to be applied to situations in which a function is called in multiple locations. When their methods are used to calculate the mapping cost between two functions, information such as whether or not the two

[1]The interference here means the two functions mapped to the same region will replace each other during execution time. We use the amount of data transfer to estimate this interference cost.

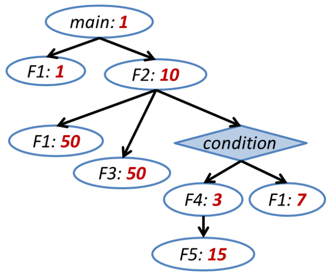


Fig. 3: ***A call graph for which previous cost calculation methods do not work*** *- The weight on each function node represents how many times the function is called.*

functions have caller-callee relationship, as well as what is the least common ancestor (LCA) of the two functions need to be known. However, if a function is called in multiple locations in the application, it is impossible to define its relationship with other functions. Let us consider an example as shown in Fig. 3, the relationship between F1 and F2 is hard to define, as there are several instances of F1 in the program. Such situation is quite common in most of the applications, in the sense that functions such as *malloc()* and *printf()* are frequently called in different locations. Failing to handle those situations makes their methods inappropriate for most applications. On the other hand, even for functions that have simple interference relationship, previous cost calculation methods may still lead to wrong results. We can see the same program shown in Fig. 3 again, where F3 is called by F2 in a loop, F5 is called by F4 in another loop, and the whole sub-tree rooted by F2 is called in the third loop. We further assume F3 and F5 were mapped to the same region and they were the only functions mapped to the region. Therefore, their interference pattern could be illustrated in the trace:

$$F3 \ldots F3F5 \ldots F5F3 \ldots F3F5 \ldots F5F3 \ldots F3F5 \ldots F5$$

In this circumstance, the cost between them should be $3 \times (size_{F3} + size_{F5})$, in which $size_F$ denotes the code size of function F. However, using the methods by [6], [7], [12], the cost between F3 and F5 is $15 \times (size_{F3} + size_{F5})$ instead, as they calculate the interference cost between function F_a and function F_b as $min(count_{F_a}, count_{F_b}) \times (size_{F_a} + size_{F_b})$, in which $count_F$ denotes the execution count of function F. Alternatively, according to [8], in which the interference cost between function F_a and function F_b is calculated as $count_{LCA} \times (size_{F_a} + size_{F_b})$, the cost between $F3$ and $F5$ is $10 \times (size_{F3} + size_{F5})$. All those methods has overestimated the code management cost.

iii) **No consideration of branch**: The mapping schemes in [6]–[8], [12] didn't consider the existence of *branch* in the program, which leads functions in different branches have the same degree when considered for mapping. Let us consider a program that has three functions, F0, F1, and F2, with F1 and F2 called by F0 in *if* and *else* respectively. In addition, they all have function size $\mathcal{S}$ and will be mapped to a code space of $2\mathcal{S}$. If F1 will be called by F0 in a probability of 90%, then it is better to map F0 and F2 into one region and F1 is left in another separate region. However, without the consideration of *branch*, the algorithm might generate a mapping that F0 and F1 are in a region, $F2$ is in another region. Even worse, if F1 is called within a loop in F0, the ignore of *branch* in the program could generate a very unsatisfactory mapping.

In this paper, we will propose correct management overhead estimation for code mapping, as well as an efficient and effective heuristic to map the code for SMM architectures.

III. Related Work

Scratchpad memory has been well known for a decade in the embedded area. Since it sheds hardware required for cache management to enable performance and silicon area advantages over the system cache, all code and data management must rely on compiler or programmer's hand inserted code [5]. There are a number of approaches for selecting what to place into the scratchpad memory and when to place them there. Steinke et al. [17] view the instruction placement problem in terms of minimizing memory accesses, and evaluate the structure of the program in the granularity of basic blocks and functions to formulate an ILP problem. Udayakumaran et al. [18] present an algorithm which looks at timestamps in code sections to determine temporal locality, The work [19], [20] present algorithms which require trace data. Egger et al. [21] implements a paged SPM management and prefetching scheme. These schemes require profiling information which is impractical as program execution varies widely on different input data when there are branches. Besides, these techniques cannot be directly applied to Software Managed Multicore (SMM) architecture. This is because of the difference in the way scratchpad memory has been traditionally used, and the way it is used in SMM architectures. In embedded systems, e.g., ARM processors, the scratchpad memory is present in addition to the regular cache hierarchy of the processor. Programs can be executed correctly without using scratchpad memory, but scratchpad memory can be used to optimize performance and power efficiency. On the contrary, the scratchpad memory is the only memory hierarchy in SMM architecture and is therefore essential, rather than optional. All code and data must go through it. As a result, while the problem of using scratchpad memory in embedded systems is that of optimization, the problem of using scratchpad memory in SMM architectures is to enable the execution of applications.

We have proposed several schemes to manage stack data and heap data for SMM architectures [9]–[11], [13], [14]. To the best of our knowledge, work [6]–[8], [12] are similar to our effort for code and [7] is the most similar one. Two mapping algorithms were proposed in [7]. One is function mapping by updating and merging (FMUM) and the other one is function mapping by updating and partitioning (FMUP). FMUM begins with a mapping in which each function is placed in a separate region. It repeatedly selects and merges a pair of regions with the minimal merge cost among all pairs of regions until all functions can fit in the given scratchpad memory size. In contrast, FMUP starts with a mapping where all functions are placed in only one memory region. It repeatedly selects the function which maximally decreases the cost and places it to another region until the size of the total amount of instruction space is less than the given memory size. In this paper, we address the problems discussed in Section II. In the next section, we formally define the code mapping problem. In Section V, we present a correct cost calculation for code management on SMM architectures, and then our efficient heuristic CMSM is presented in Section VI.

TABLE I: Symbols used for problem definition

Variable	Description
S_p	Size of local scratchpad memory;
F	A set of functions that are in the program;
FID	A set of function IDs;
N	Number of function IDs in *FID*;
$M(fid, f)$	An associate set contains all association between fid and f, where $fid \in$ *FID*, $f \in F$;
S_{fid}	A set contains function sizes of function id *fid*, where *fid* $\in$ *FID*;
R	A set of function regions;
S_f	A set contains code sizes of function f in F.

* Please note that the mapping relation between a function to function IDs is "one-to-many".

IV. Problem Definition

The symbols used in the problem definition are shown in Table I.

Decision variables:

- $x(f, r)$: {0,1}, indicates whether function f is mapped to region r.
- S_r: *integer*, indicates the size of region r, where $r \in R$.

Derived variables:

- $y(fid, r)$: {0,1}, indicates whether function id fid is mapped to region r, where $fid \in$*FID* and $r \in R$.
- $num(id_a, id_b, r)$: *integer*, indicates the number of ids between id_a and id_b that are mapped to region r, where $id_a, id_b \in$*FID* and $r \in R$.
- $switch(id_a, id_b)$: {0,1}, indicates whether id_b causes function context switch during the execution of program, where $id_a, id_b \in$*FID* and $id_a < id_b$.

Constraints:

1) Region Size: The region must be large enough to accommodate all functions mapped to it. Therefore, this region must be larger than any function mapped to it.

$$\forall r \in R, f \in F : S_r \geq x(f, r) * s_f$$

2) Function Mapping Unicity: A function f can and only can be mapped to one region.

$$\forall f \in F : \sum_{r \in R} x(f, r) = 1$$

3) Scratchpad Memory Size: The total size of all regions must not exceed the scratchpad memory size predefined.

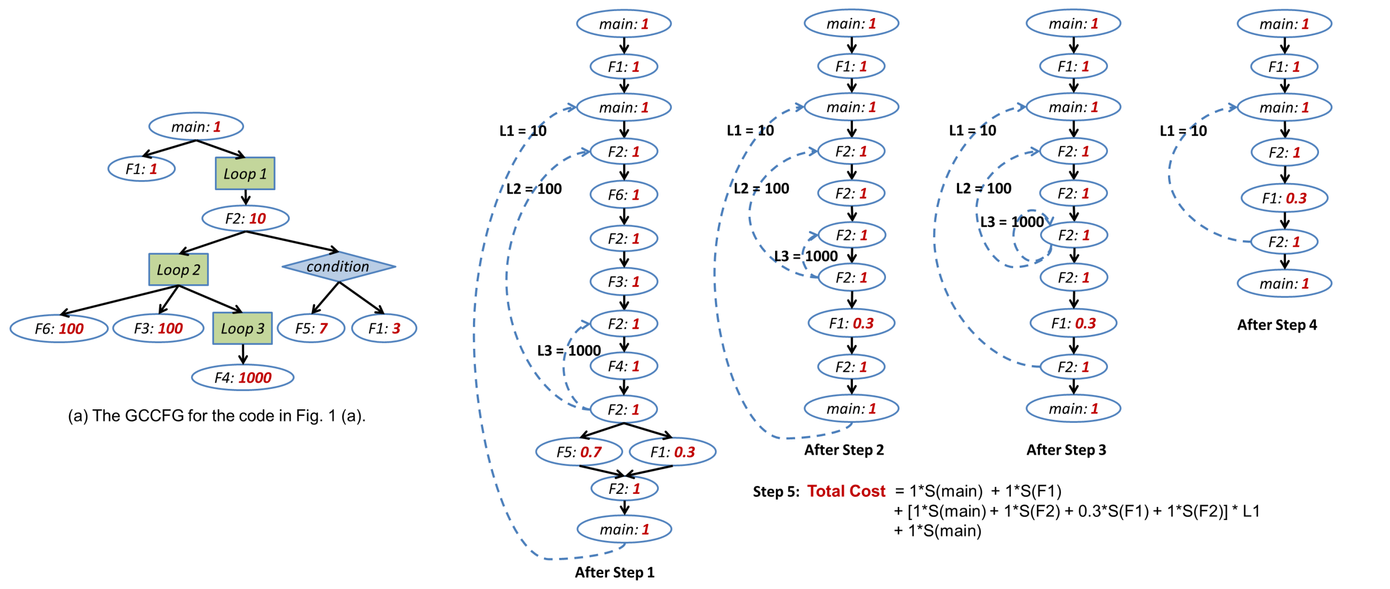
$$\sum_{r \in R} S_r \leq S_p$$

4) Mapping Association: When function f is mapped to a region r, all function id fid corresponds to f must be also mapped to region r.

$$\forall (fid, f) \in M(fid, f), fid \in FID, f \in F, r \in R :$$
$$y(fid, r) = x(f, r)$$

5) Number of Functions: Number of functions between two ids needs to be updated dynamically.

$$num(id_a, id_b) = \sum_{r \in R} \sum_{id_a < id < id_b} x(id, r) * y(id_a, r) * y(id_b, r)$$



(a) The GCCFG for the code in Fig. 1 (a).

(b) Example: calculate the cost of mapping region {main, F1} and region {F2} into one region

Fig. 4: ***GCCFG and an illustration of cost calculation in Algorithm 1***

6) Determination of *switch*:

a) If there are many functions between to-be-determined-two functions, there are no context switch between these two functions.

$$(1 - switch(id_a, id_b)) * N \geq num(id_a, id_b)$$

b) On the other hand, when there are no other functions between to-be-determined-two functions, context switch must happen between them.

$$switch(id_a, id_b) \geq 1 - num(id_a, id_b)$$

Objective Function:

- We choose the number of instruction transfers from the main memory to the local scratchpad memory as the cost metric. When there is a context switch from function id_a to function id_b, we need to get all instructions of function id_b through DMAs. Therefore, our objective must be to minimize the total amount of instruction transfer of all regions in the local scratchpad memory, where s_{id_b} is the code size of the function id_b.

$$\text{minimize} \sum_{id_a, id_b \in FID} switch(id_a, id_b) * s_{id_b}$$

V. Cost Calculation

A. Graphical code representation

In order to calculate the management cost correctly and map code efficiently, we need to deeply understand the structure of the input program and represent the flow information and control information in a proper form. We build this information into an enhanced Control Flow Graph (CFG) known as Global Call Control Flow Graph (GCCFG) presented by Pabalkar et al. [6].

Definition 1: (Global Call Control Flow Graph). A global call control flow graph (V, E) is an ordered acyclic directed graph, where $V = V_F \bigcup V_L \bigcup V_C$. Each node $v_f \in V_F$ with a weight w_f on it represents a function or F-node, $v_l \in V_L$ denotes a loop or L-node, $v_c \in V_C$ represents a conditional or C-node. w_f is the number of times function f is invoked in the program. An edge e_{ij} ($e_{ij} \in E$) shows a directed edge between F-nodes, L-nodes and C-nodes.

Property. If v_i and v_j are functions, then the edge represents a function call. If v_j is an L-node or a C-node then it represents control flow. If v_i is a C-node, then the edge represents one possible path of execution. If v_i is a loop, then the edge represents what is being executed in the body of the loop. If v_j is a loop and its ancestor is a loop then the edge represents a nested loop execution. The edges are ordered, edges to the left execute before edges to the right, except in the case of condition nodes. Edges leaving condition nodes can execute their *true* or *false* children, where all true children are ordered and all false children are ordered.

As an example, Fig. 4 (a) shows the GCCFG of the program in Fig. 1 (a). We ignore direct recursive function calls F_5 in the graph. Since we are concerned with cost between different functions, the effect of a direct recursive call is that the code necessary to run the called function is already in memory, resulting in no instruction transfers. The support for the construction of GCCFG with mutual recursive functions could be a future work. In addition, we conservatively expand indirect function calls invoked through function pointers in much the same way as they were called with equal probability outside of any conditional node.

B. Cost calculation graph

Definition 2: (Cost Calculation Graph). A cost calculation graph (V, E) is a cyclic directed graph, where $E = E_B \bigcup E_N$. Each node $v_f \in V$ with a taken probability p_f on it represents a function node. A backward edge e_{ij} ($e_{ij} \in E_B$) shows a loop in the program, a normal edge e_{ij} ($e_{ij} \in E_N$) shows the order the function is executed.

Algorithm 1 Algorithm cost (R_a, R_b, GCCFG)

```
1: Step 1: Transform GCCFG to cost calculation graph by modifying Depth First Search (DFS) algorithm;
2: Step 2: Remove all other functions from the graph and just keep the functions in the two regions $R_a$ and $R_b$;
3: Step 3: If the first function and the last function in a loop are identical, then the last function in the loop must be moved out of the loop and put right close to the function after the loop;
4: Step 4: Eliminate identical adjoining functions ($f_b$ after $f_a$, where there are the same function) in the graph. If a function is the only function in a loop, then remove the loop and its corresponding loop count.
5:     If $f_a$ and $f_b$ are separated by a loop, then keep the one that is inside the loop;
6:     If $f_a$ is before a branch and $f_b$ is within a branch, then the first function node in all branches must be checked to see whether they are the same function as $f_a$. If not, no removal here; otherwise, we can remove one.
7: Step 5: Calculate the cost totalCost.
8:     Rule: cost (F) = 1 $\times$ branch probability (F), where F is a function in the program. (The function that is not in the branch has probability 1).
9:     For all functions in a loop, we apply the Rule to compute the total cost in the loop. However, the total cost must be multiplied by its loop count, and then be added to totalCost.
10: return totalCost;
```

Cost calculation graph is an estimation of execution trace of a program. The leftmost graph in the Fig. 4 (b) shows the cost calculation graph of GCCFG in Fig. 4 (a). As GCCFG is ordered, the cost calculation graph could be constructed by modifying Depth First Search (DFS) algorithm, where function nodes are put in the same order as the function nodes in GCCFG are touched in DFS. When a loop node is found, a backward edge will be added after the traversal of the whole sub-tree. When a conditional node is met, we would attach all nodes connected to conditional node to its parent node, and function nodes in each diverging path will keep the same order as they are in the order of DFS traversal.

C. Cost calculation algorithm

Algorithm 1 outlines a simplified version of the cost calculation algorithm and Fig. 4 (b) shows an illustration of our algorithm. In this example, we are trying to evaluate the cost of mapping region $\{\text{main}, F_1\}$ and region $\{F_2\}$ in one region. We firstly transform GCCFG to cost calculation graph at step 1, and then remove all irrelevant functions in the program at step 2. The step 3 moves the last function in the loop out of the loop if it is the same as the first function in the loop. At step 4, we remove all redundant functions. When there is only one function in a loop, we can remove the loop information to further eliminate the redundancy. Finally, we calculate the mapping cost at step 5, where we consider the impact of branch probability and the existence of loops.

Proof of correctness. Since GCCFG is an ordered acyclic directed graph, the transformation at step 1 approximately emulates the execution trace of function calls in the program. As we only calculate the cost of mapping functions from two regions into one region, the removal of other unrelated functions at step 2 will not change the correctness of our final cost calculation. Step 3 will not affect the correctness of the calculation, since there is no context switch in the loop if the first and the last function in the loop are the same. However, one copy of node must be put in the graph, since there might exist a context switch right after the loop. When two adjoining functions in the graph are identical, we assert there is no context switch in the target region. Therefore, we can remove redundant copies at step 4. Even more, if the function is the only function in a loop, the loop information must be removed as well, as the function itself will not result in any instruction transfer.

Complexity. At the step 1, we use modified DFS algorithm to generate cost calculation graph, and therefore the timing complexity is $O(|E|)$. We traverse the cost graph from step 2 to step 4. Therefore, it adds $O(|V_f|)$ to the time. Consequently, the timing complexity of cost calculation algorithm is $O(|E|)$ ($\approx O(|V|)$ in the cost calculation graph).

Algorithm 2 Algorithm CMSM (GCCFG, $\mathcal{S}$)

```
1: SPMregions {set of $N$ regions in the scratchpad memory}   ▷ $N$ is the number of functions in the program
2: $R_{dest} \leftarrow 0$, $R_{src} \leftarrow 0$;
3: while SPMSize() > $\mathcal{S}$ do
4:     FindMinBalancedMerge($R_{dest}$, $R_{src}$, GCCFG);
5:     MergeRegions($R_{dest}$, $R_{src}$);
6:     SPMregions.erase($R_{src}$);
7: end while
```

Algorithm 3 FindMinBalancedMerge (&R_{dest}, &R_{src}, GCCFG)

```
1: minMergeCost ← DBL_MAX, tmpCost ← 0;
2: for all combination of regions $R_1$, $R_2 \in$ SPMregions do
3:     size1 ← RegionSize($R_1$), size2 ← RegionSize($R_2$);
4:     max ← max(size1, size2), min ← min(size1, size2);
5:     tmpCost = cost($R_1$, $R_2$, GCCFG) * $\frac{max-min}{(max+min)^2}$;
6:     if tmpCost < minMergeCost then
7:         minMergeCost = tmpCost;
8:         $R_{dest}$ = $R_1$;
9:         $R_{src}$ = $R_2$;
10:    end if
11: end for
```

VI. Code Mapping Heuristic: CMSM

Algorithm 2 outlines our CMSM heuristic. It starts with a mapping, in which each function is mapped to a separate region (line 1). Now all combinations of two regions are tried to be merged until the total space meets memory constraints (while loop, lines 3-7). To do this, we firstly find two "balanced" regions with minimal merge cost through function FindMinBalancedMerge() at line 4. We then merge two regions and update the region information in the set SPMregions (line 5-6). Function FindMinBalancedMerge() is described in Algorithm 3. To do this, we choose a region pair (R_1, R_2) (Algorithm 3, line 2-11), and calculate its merge cost at line 5. Here, we utilize the cost function from Algorithm 1. Besides, there is a balance factor $\frac{max-min}{(max+min)^2}$. It is inclined to place the functions having close object sizes into the same region. It is important, since we can compress the total code space in the local scratchpad memory and use less memory. This remaining space could result in more number of regions as long as there are functions that could be accommodated to it. Even if no more regions would be generated, it is still beneficial to use less space to achieve competitive performance. As stated before, the local scratchpad memory is shared among global data, stack data, heap data and instructions of the managed program, less space consumed by instructions indicates more space for other data that could eventually results in better performance.

Complexity. The *while loop* at line 3 in Algorithm 2 merges two regions at a time. Since in the worst case, all regions might have to be merged into one, this loop can execute $|V_f|$ times. Inside this, the for loop (lines 2-11 in Algorithm 3) runs for each pair of regions. This adds $O(|V_f|^2)$ complexity to the time. Inside the loop, there is a cost calculation which has complexity $O(|V|)$. Thus the worst case timing complexity of CMSM is $O(|V_f|^4)$.

VII. Extension to CMSM: Static Weight Assignment

In previous section, we presented our greedy algorithm for code management on SMM architectures. However, in order to fully automate the code mapping process, we describe our compilation time scheme for estimating the number of function calls on each function node. The basic blocks of the managed application are first scanned for the presence of loops (back edges in a dominator tree), conditional statements (fork and join points) and function calls (branch and link instructions). After capturing these information, we assign the weights on the functions by traversing GCCFG in a top-down fashion. Initially, they are assigned to 1. When a *loop node* is encountered, the weight on all its descendant function nodes equals the weight of *loop node*'s nearest ascendant *function node* in the path multiplying a fixed constant, *loop factor Q*. This ensures that a function which is called inside a deeply nested loop will receive a greater weight than other functions. When a *conditional node* is encountered, the weight on each descendant function node equals to the weight of *conditional node*'s nearest parent *function node* multiplying the branch probability of each edge diverging from the *conditional node*. We adopted a traditional scheme presented by [22] to predict the branch probability. We found the impact of Q is negligible as long as it is larger than 1. As a result, in our scheme, we choose $Q = 10$. The previous Fig. 4 (a) is the resulted GCCFG of the example code with our static weight assignment scheme.

VIII. Empirical Evidence

A. Experimental Setup

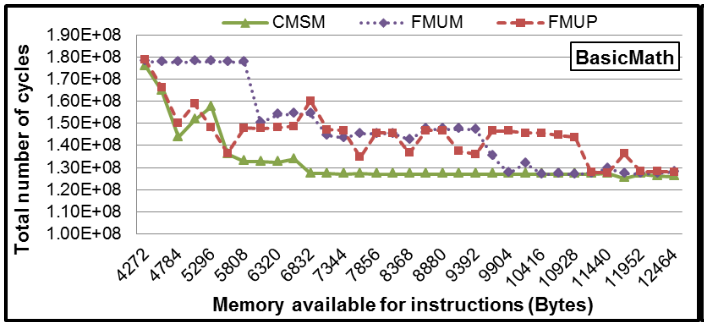
We use IBM Cell BE [2] as our hardware platform. It is a multicore processor, and gives us accesses to 6 of the 8 Synergistic Processing Elements (SPEs). In addition, this architecture has a main memory on main core, and only a scratchpad memory on each execution core, or SPE. Scratchpad memory is small, and therefore the program needs to be managed in software when its code is larger than memory available. The benchmarks used for experimentation are from Mibench suite [1] and presented in Table II. All those information is profiled by compiling programs for SPE. *functions* is the total number of functions in the program, including library functions tailored for SPE. *min code* is the smallest possible mapping size of code space, defined by the size of the largest function in the application. *max code* is the total size of the program. We utilize main core and only 1 SPE available in the IBM Cell BE in most of our experiments, except the one designed for demonstrating scalability of our heuristic in Section VIII-F.

TABLE II: Benchmarks, their minimum sizes of code space, and maximum sizes of code space.

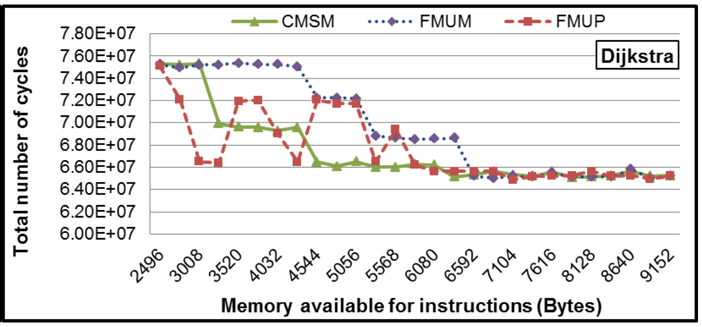
Benchmark	functions	min code (B)	max code (B)
Adpcm_decoding	13	1552	6864
Adpcm_encoding	13	1568	6880
BasicMath	20	4272	12128
Dijkstra	26	2496	9216
FFT	27	2496	12776
FFT_inverse	27	2496	12776
String_Search	17	632	4708
Susan_Edges	24	19356	37428
Susan_Smoothing	24	19356	37428

B. Overall performance comparison

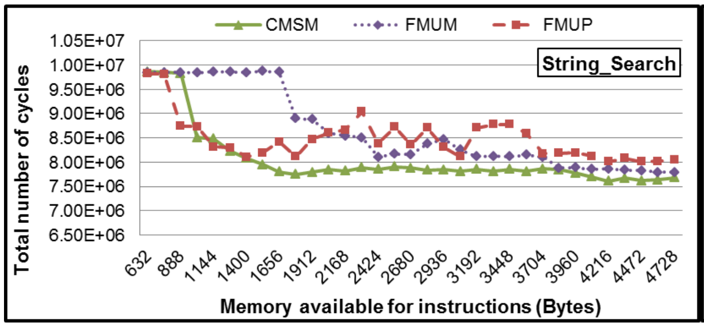
While the results are scalable for all benchmarks, Fig. 5 shows the execution time of the binary compiled using each heuristic for four representative applications. The X-axis shows a wide range from *min code* to *max code* of each program, with the step size 256 bytes. As observed from the figure, when the code space is very tight, all heuristics achieve the same mapping, i.e., mapping all the functions in one region. However, as we start relaxing the code size constraint, CMSM typically performs better than FMUM and FMUP. There are two main reasons. First, our CMSM is inclined to place two functions with small merge cost and similar code size in one region at each step of merging. It is achieved by using a "balance" factor described in our algorithm. The benefit of doing so is to increase the number of regions in the code space. We expect mapping solutions with more regions to give lower overhead costs, as only functions mapped to the same region will swap each other during run time. Second, our CMSM considers the effect of branches in the applications and utilizes a correct management cost calculation scheme. Their impact is further evaluated in Section VIII-C. The reverse effect is also visible. When the code size constraint is extremely relaxed, e.g., larger than 70% of *max code* present in Table II, all three algorithms again achieve very similar code mapping. This is because there are quite few functions mapped to one region when the code space is sufficient enough. The small differences in code mapping generate negligible effect on performance. Note that code mappings created by the CMSM do not always outperform the other two heuristics. For example, when memory available for instructions of benchmark "dijkstra" is 3520 bytes in Fig. 5b, CMSM is worse than FMUP. This is because FMUP has to do very few steps, while CMSM needs to do a lot of merges. The more steps a heuristic has to take, the errors in each step accumulate, and eventually a worse mapping might be generated. Although our heuristic does not consistently gives good results, it gives better results most of the times. We tested the three heuristics for all code size constraints from minimum to maximum. On average over all benchmarks, CMSM gives a better result than other two algorithms 89% of time. Another important observation from Fig. 5 is that, applications are tend to have less execution time when their code space become larger. A large code space usually leads to more number of regions in it, and therefore less functions overlap each other in regions. This explains the trade-off between the performance and the memory available



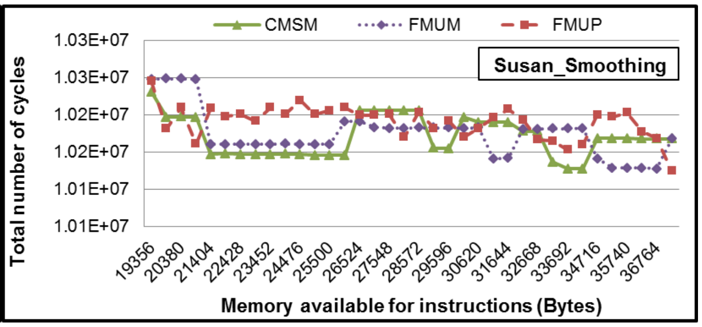
(a)



(b)



(c)



(d)

Fig. 5: ***Performance comparison against FMUM and FMUP***

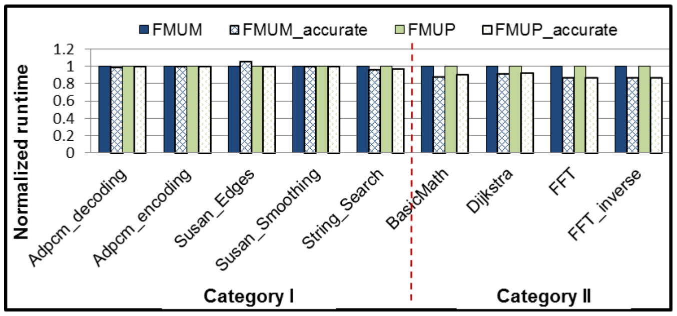


Fig. 6: ***Impact of accurate cost calculation on FMUM and FMUP***

for instructions.

C. Importance of accurate cost calculation

In this experiment, we conducted another set of experiments that evaluates the impact of accurate cost estimation on code mapping heuristics. We change the memory size from 20% of the maximum size to 70% of the maximum size for each benchmark and plot the average results in Fig. 6. y-axis is the normalized execution time, where the runtime of application with FMUM_*accurate* is normalized to its performance with FMUM and the runtime of application with FMUP_*accurate* is normalized to its performance with FMUP, respectively. FMUM_*accurate* and FMUP_*accurate* use previous heuristics [7], but with our correct management overhead calculation instead of theirs. x-axis presents two categories for all our tested programs. Applications in category I have simple structure and very few branches, and applications in category II are the programs that contain complicated call patterns and many branches. As shown in Fig. 6, programs from the category II benefit from accurate cost calculation. Their performance are improved by 12% when managed with accurate management cost estimation method. This benefit comes by considering branch probabilities as weights during the course of management overhead calculation. We believe our improvement is very important, as most large programs do have intricate call patterns and lack of considering branches in the program will lead to inferior function mapping for code management.

D. Compile time comparison

Including branch probabilities and correctly calculating the management overhead complicate the compilation. In this section, we evaluate the compilation time of generating linker script by FMUM, FMUP and CMSM heuristics respectively. We found that, even with an increase for our algorithm (less than 5% on average), the compilation time for the set of benchmarks are always less than a minute. This is definitely tolerable for several embedded applications which are compiled once, distributed as binaries, and executed many times. This profile closely fits the kinds of applications that are intended for embedded processors, e.g., multimedia and office applications.

E. Accuracy of weight assignment

We examined the goodness of our static weight assignment on function nodes of GCCFGs of nine applications. We compared the execution time of each benchmark using static assignment with its execution time using profile-based assignment. Averagely, we found both schemes achieve similar performance for the set of benchmarks. This implies that we can eliminate the compile time overhead to obtain

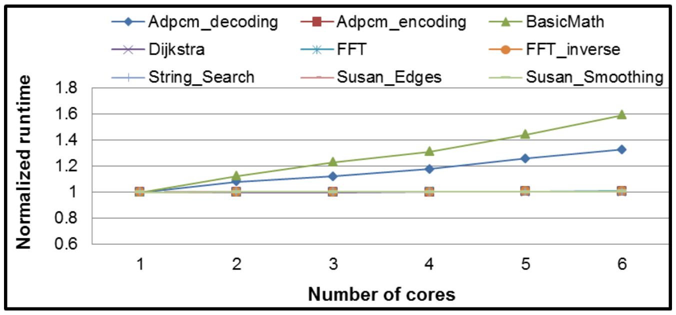


Fig. 7: ***Scalability of CMSM on multicore processors***

profiling information through the loop based function weight assignment. It also makes the code management technique more comprehensive, since profiling large applications is time-consuming and intimidating.

F. Scalability

Fig. 7 shows the results we examined the scalability of our CMSM heuristic. We normalized the execution time of each benchmark with number of SPEs to its execution time with only one SPE, and show them on y-axis. In this experiment, we executed the identical application on different number of cores. According to the graph, the runtime difference with the increased number of SPEs is negligible even in such aggressive configuration. In this configuration, DMA transfer occur almost at the same time when instructions need to be moved between the main memory and the local memory. This will make the Elemental Interconnect Bus (EIB) saturated. Benchmark *BasicMath* increases most steeply, as there are many instruction transfers in the program, which makes each SPE have more execution time.

IX. Summary

Software Managed Multicore (SMM) architectures are one of promising solutions to the problem of scaling the memory hierarchy. However, since scratchpad memory cannot always accommodate the whole task mapped to it, certain schemes are required to mange code, global data, stack data and heap data of the program to enable its execution. This paper focuses on managing code between the main memory and the local memory, since an efficient and effective code management scheme is of utmost importance to the overall performance of the system. In this paper, we formally define the problem of mapping code for SMM architectures at the granularity of function object. Besides, we propose a correct cost calculation for code mapping, as well as a heuristic approach named CMSM for the same problem. Our experimental results show that CMSM generates code mapping which leads to significant performance improvement compared to previous work.

References

[1] M. R. Guthaus et al., "MiBench: A Free, Commercially Representative Embedded Benchmark Suite," in *Proc. of Workload Characterization*, 2001, pp. 3–14.

[2] B. Flachs et al., "The Microarchitecture of the Synergistic Processor for A Cell Processor," *IEEE Solid-state circuits*, vol. 41, no. 1, pp. 63–70, 2006.

[3] "The SCC Programmer's Guide," Tech. Rep.

[4] L. Truong, "Low Power Consumption and a Competitive Price Tag Make the six-core TMS320C6472 Ideal for High-performance Applications," Texas Instruments, Tech. Rep., 2009.

[5] 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+ISSS*, 2002, pp. 73–78.

[6] A. Pabalkar, A. Shrivastava, A. Kannan, and J. Lee, "SDRM: Simultaneous Determination of Regions and Function-to-Region Mapping for Scratchpad Memories," in *Proc. of HPC*, 2008, pp. 569–582.

[7] S. C. Jung, A. Shrivastava, and K. Bai, "Dynamic Code Mapping for Limited Local Memory Systems," in *Proc. of ASAP*, 2010, pp. 13–20.

[8] 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*, 2010, pp. 287–296.

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

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

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

[12] C. Jang, J. Lee, B. Egger, and S. Ryu, "Automatic Code Overlay Generation and Partially Redundant Code Fetch Elimination," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 2, pp. 10:1–10:32, Jun. 2012.

[13] K. Bai and A. Shrivastava, "Automatic and Efficient Heap Data Management for Limited Local Memory Multicore Architectures," in *Proc. of DATE*, 2013, pp. 593–598.

[14] J. Lu, K. Bai, and A. Shrivastava, "SSDM: Smart Stack Data Management for Software Managed Multicores (SMMs)," in *Proc. of DAC*, 2013, pp. 149–156.

[15] "Programmer's Guide: Software Development Kit for Multicore Acceleration Version 3.1," Tech. Rep.

[16] M. Verma and P. Marwedel, "Overlay Techniques for Scratchpad Memories in Low Power Embedded Processors," *IEEE VLSI*, vol. 14, no. 8, pp. 802–815, 2006.

[17] S. Steinke et al., "Reducing Energy Consumption by Dynamic Copying of Instructions onto On-chip Memory," in *Proc. of ISSS*, 2002, pp. 213–218.

[18] S. Udayakumaran, A. Dominguez, and R. Barua, "Dynamic Allocation for Scratch-pad memory Using Compile-time Decisions," *Trans. on Embedded Computing Sys.*, vol. 5, no. 2, pp. 472–511, 2006.

[19] A. Janapsatya, A. Ignjatović, and S. Parameswaran, "A Novel Instruction Scratchpad Memory Optimization Method Based on Concomitance Metric," in *Proc. of ASP-DAC*, 2006, pp. 612–617.

[20] F. Angiolini et al., "A Post-compiler Approach to Scratchpad Mapping of Code," in *Proc. of CASES*, 2004, pp. 259–267.

[21] B. Egger, J. Lee, and H. Shin, "Scratchpad Memory Management for Portable Systems with A Memory Management Unit," in *Proc. of EMSOFT*, 2006, pp. 321–330.

[22] J. E. Smith, "A Study of Branch Prediction Strategies," in *Proc. of ISCA*, 1981, pp. 135–148.