Efficient Code Assignment Techniques for Local Memory on Software Managed Multicores

JING LU, KE BAI, and AVIRAL SHRIVASTAVA, Arizona State University

Scaling the memory hierarchy is a major challenge when we scale the number of cores in a multicore processor. Software Managed Multicore (SMM) architectures come up as one of the promising solutions. In an SMM architecture, there are no caches, and each core has only a local scratchpad memory [Banakar et al. 2002]. As the local memory usually is small, large applications cannot be directly executed on it. Code and data of the task mapped to each core need to be managed between global memory and local memory. This article solves the problem of efficiently managing code on an SMM architecture. The primary requirement of generating efficient code assignments is a correct management cost model. In this article, we address this problem by proposing a cost calculation graph. In addition, we develop two heuristics *CMSM* (Code Mapping for Software Managed multicores) and *CMSM_advanced* that result in efficient code management execution on the local scratchpad memory. Experimental results collected after executing applications from the MiBench suite [Guthaus et al. 2001] demonstrate that merely by adopting the correct management cost calculation, even using previous code assignment schemes, we can improve performance by an average of 12%. Combining the correct management cost model and a more optimized code mapping algorithm together, our heuristics can reduce runtime in more than 80% of the cases, and by up to 20% on our set of benchmarks, compared to the state-of-the-art code assignment approach [Jung et al. 2010]. When compared with Instruction-level Parallelism (ILP) results, *CMSM_advanced* performs an average of 5% worse. We also simulate the benchmarks on a cache-based system, and find that the code management overhead on SMM core with our code management is much less than memory latency of a cache-based system.

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

General Terms: Algorithm, Design, Experimentation, Performance

Additional Key Words and Phrases: Code, instruction, local memory, scratchpad memory, embedded systems, multicore processor

ACM Reference Format:
Jing Lu, Ke Bai, and Aviral Shrivastava. 2015. Efficient code assignment techniques for local memory on software managed multicores. ACM Trans. Embed. Comput. Syst. 14, 4, Article 71 (December 2015), 24 pages.
DOI: http://dx.doi.org/10.1145/2738039

1. INTRODUCTION

Modern processors need to be optimized for low power, high performance, and compact area [Vaidyanathan et al. 2015]. Multicore architectures dominate the processor market since they provide a way to improve peak performance without much increase in the power and area consumption. In addition, several other parameters (e.g., power, temperature, and reliability) can be more easily managed at coarser granularity of thread and core level [Michaud et al. 2007]. As we transition from few-core to many-core

This research was funded by grant CCF-0916652 from National Science Foundation.
Authors' addresses: J. Lu, K. Bai, and A. Shrivastava are with the Compiler Microarchitecture Lab, Arizona State University, 203 Centerpoint, 660 South Mill Avenue, Tempe, AZ 85281, USA; emails: {Jing_Lu, Ke.Bai}@asu.edu; corresponding author's email: Aviral.Shrivastava@asu.edu.

DOI: http://dx.doi.org/10.1145/2738039

processors, scaling the memory hierarchy becomes one of the most important challenges.

Automatic memory management in the hardware that is transparent to applications is increasingly becoming infeasible. Most experts believe that fully cache-coherent architectures will not scale when there are hundreds and thousands of cores [Bournoutian and Orailoglu 2011; Choi et al. 2011; Garcia-Guirado et al. 2011; Xu et al. 2011], and therefore architects are looking for alternative scalable architecture designs. In 2009, a 48-core noncoherent cache architecture named Single-chip Cloud Computer (SCC) was manufactured by Intel [2012]. The latest six-core DSP from Texas Instruments, TI 6472 [Truong 2009], features noncoherency cache architecture. More recently, the MPPA-256 processor that adapts distributed memory architecture was manufactured by KALRAY [de Dinechin et al. 2013]. But caches still consume a large portion of power and die area [Banakar et al. 2002].

Distributed memory systems where each core has access to only a limited local memory (Scratchpad Memory, SPM) seems to be one of promising memory scaling options. 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 [Banakar et al. 2002]. Therefore, such scratchpad-based multicore architectures have the potential to be more power efficient and scalable than traditional cache-based architectures. A very good example of such architectures 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 5GFlops per watt [Flachs et al. 2006], while the power efficiency of an Intel i7 four-core Bloomfield 965 XE is only 0.5GFlops per watt [Intel 2010; Tom's Hardware 2010].

The scratchpad-based multicore is a truly "distributed memory architecture on-a-chip." Applications for it are written in the form of interacting tasks. The tasks are assigned to the cores of the scratchpad-based multicore architecture. Each task executes on one execution core with only one thread. Each execution core can only access its local scratchpad memory, and to access other local memories or the global memory, explicit Direct Memory Access (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. As there is no cache implemented in the hardware in these scratchpad-based multicore architectures, explicit data management is needed, including evicting the "not-so-urgently" needed data out to global memory, and bringing in the required data into the limited local scratchpad memory. A hardware solution of code management for scratchpad memory has been proposed in 2011 [Metzlaff et al. 2011]. Another similar technique has been presented by Schoeberl [2009]. However, hardware managed scratchpad memories require extra hardware components, which incurs larger power consumption than static scratchpad memories, even for the cases when data management is not needed. As a result, we decided to focus on managing data and code with software techniques. Due to the explicit need of data management in software, we term the scratchpad-based multicore architectures that are purely software managed as Software Managed Multicore (SMM) architectures.

How to manage the task data on scratchpad memory of the cores is an important problem that has drawn significant attention in recent years [Bai et al. 2013; Baker et al. 2010; Holton et al. 2014; Jang et al. 2012; Jung et al. 2010; Pabalkar et al. 2008]. While management is needed for all code and data of the task when they cannot fit in the local memory, in this article we focus on the problem of code management, since efficient code management can significantly impact the performance of the system. Even for applications that are data intensive, code management is still very important. The memory space is shared between code and data, and the more space is given to the data, the better performance will be achieved [Bai et al. 2011a, 2011b; Bai and Shrivastava 2010, 2013a, 2013b, Lu et al. 2013]. A good code management scheme can reduce the space that is required to hold the code, and save more space for the data.

The code assignment problem requires us to determine what to place into SPM, and where and when to place it there. In this article, we consider the situation that a portion of code is located in a space of the local memory, and other code is placed to the global memory. The code in the global memory must be dynamically fetched into the local memory when needed. In order to do this more efficiently, we divide this space into regions, and also assigning functions into regions. Functions assigned to one region are compiled and linked starting with the same physical address (that of the region). At runtime, only one function out of the ones that are assigned 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 global memory using a Direct Memory Access (DMA) command [IBM 2006]. The size of the region is equal to the size of the largest function assigned 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 goal of the code assignment problem on SMM architectures is then to (i) divide the code space into regions, and (ii) find an assignment of functions to regions, such 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 global memory.

Finding the number of regions and assigning the functions to regions that will minimize instruction transfer, have both been proven to be intractable [Pabalkar et al. 2008; Verma and Marwedel 2006]. Therefore, several greedy algorithms have been proposed [Baker et al. 2010; Jang et al. 2012; Jung et al. 2010; Pabalkar et al. 2008]. A key component of code management schemes is a cost model to estimate the overhead of code management for a given assignment of functions to regions. Specifically, the cost herein is the amount of instructions that will be transferred between local scratchpad memory and global memory. This cost model “drives” the technique to assign functions into regions, and in a sense, the function assignment technique can only be as good as the accuracy of this cost estimation function. Unfortunately, all previous schemes 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 as follows: (i) The assumption that the code management overhead when two functions are assigned to the same region is independent of where the other functions are assigned. This is incorrect, because as we show in this article, the overhead depends on which other functions are assigned to the same region as the two functions. (ii) Several cases of inaccurate estimation; for example, 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%). Although the inaccuracies in the estimation of code management overhead caused by these limitations do not result in wrong results, they do affect generating efficient assignment decisions, which therefore degrade the performance of applications being managed.

We address the aforementioned problems and make the following contributions:

—We formulate the code assignment problem on SMM architectures. As far as we know, this is the first time of formal problem formulation for SMM architectures.

—We devise a novel cost calculation graph, which properly considers all circumstances leading to instruction transfer that cannot be handled in previous works. This is very important, as the cost calculation graph is the base for all assignment algorithms, not only for ours.
—Branches in the program are considered when calculating the management overhead. We found the consideration of branch is of utmost importance and the neglect of branches in cost calculation in previous works is one of the reasons that leads to inefficient code assignment.
—Two fast polynomial time heuristics for Code Mapping on Software Managed multi-core processors (named as CMSM and CMSM_advanced) are proposed. CMSM takes in the graphical code representation of the program, and then transforms the representation to a cost calculation graph, from which management cost is calculated and a code assignment is ultimately executed. CMSM_advanced further optimizes the CMSM algorithm by expanding functions to possible remaining space.

We establish the effectiveness of the proposed cost calculation and assignment techniques by executing benchmarks from the MiBench suite [Guthaus et al. 2001], and comparing with the closest approach [Jung et al. 2010]. We find that when we use correct estimation of code management overhead, even in the previous code management technique, it leads to an assignment that performs 12% better. On top of that, our heuristics can reduce runtime in more than 80% of the cases, and by up to 20% on our set of benchmarks. When compared with Instruction Level Parallelism (ILP) results, CMSM_advanced performs an average of 5% worse. We also simulate the benchmarks on a cache-based system, and find that the code management overhead on the SMM core with our code management is much less than memory latency of a cache-based system. It should be noted that, our method is based on a reasonable assumption that there is only one thread on each execution core of SMM architectures. This is because the local scratchpad memory is usually small and it is not affordable to support multiple threads on each core (because thread management itself consumes local memory). In addition, our method is orthogonal to genuine multithreaded parallelism techniques. This means proper parallelism is handled by other algorithms, programming models, or programmers themselves. Once the process on each core is defined, our scheme can manage its memory.

2. BACKGROUND OF CODE ASSIGNMENT

To run an application with a larger code footprint than what is available on the local scratchpad memory, typically the code overlay scheme is leveraged [IBM 2006]. 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 shown in Figure 1, a code overlay organization needs to determine both the number of regions and the assignment of functions to regions. Functions assigned to the same region are located in the same physical address, and must replace each other during runtime (by instruction fetching function *_ _ovly_load()* before each function call[1]) [IBM 2006]. The size of a region is the size of the largest function assigned to 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 objects, but that may increase the code space too much. On the contrary, assigning all functions into one

[1]As code never get dirty, we do not need to update it to the global memory. This is the reason why only *_ _ovly_load()* is sufficient.

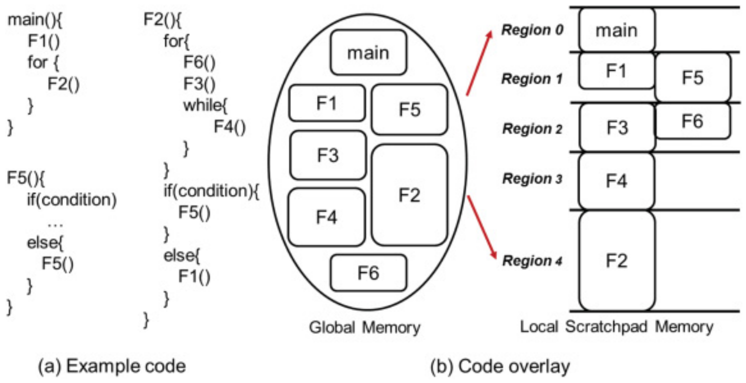

Fig. 1. Code assignment problem on scratchpad memory—when the task that is assigned to the execution core has a larger footprint than the available space, code needs to be assigned between external global memory and the local scratchpad memory of the core.

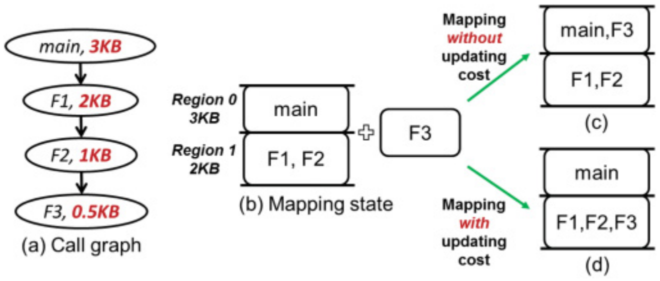

Fig. 2. Cost between functions depends on where other functions are assigned, and updating the costs as we assign the functions can lead to a better assignment.

region uses the minimum amount of code space, while incurring too many instruction transfers and therefore runtime overhead. *The task of optimizing code assignment is, to organize the application functions into regions that will obtain a balance between the code space used and the data transfers required.*

3. MOTIVATION

The works by Pabalkar et al. [2008], Baker et al. [2010], Jung et al. [2010], and Jang et al. [2012] provide heuristics for code assignment on SMM architectures. However, they are all not efficient enough, which prevents scratchpad memory from becoming a competitive alternate of cache on multicore processors in terms of performance. The inefficiency is mainly because of inaccurate or incorrect cost calculation.

3.1. Importance of Updating Cost

Previous works [Baker et al. 2010; Jang et al. 2012; Pabalkar et al. 2008] statically calculate the assignment cost and generate an assignment. They did not dynamically update the cost during the course of the assignment algorithm. It is *incorrect* and can lead to inferior assignment. Figure 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 functions. Let us consider a case that requires us to assign all functions into a scratchpad memory of 5KB. It is slightly tricky to calculate the cost between multistep calls. For example, when computing the cost between *main* and

F2, if *main* and F2 are assigned to the same region, the interference[2] between them depends on where F1 is assigned. If F1 is assigned to another different region, then the interference between *main* and F2 is just sum of their sizes, namely, 3KB + 1KB = 4KB. The calculation is as follows. When F2 is called, 1KB of function F2 will need to be brought into the memory. When the calling state returns to *main*, 3KB of the code of *main* needs to be brought into the scratchpad. However, if all *main*, F1, and F2 are assigned 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 [Baker et al. 2010; Jang et al. 2012; Pabalkar et al. 2008] computed the worst case interference cost, that is, 4KB for *main*-F2, and never updated it, and therefore obtained inferior assignment. To explain this, Figure 2(b) shows a state in assignment when *main*, F1, and F2 have already been assigned. *main* is alone in region 0, and F1 and F2 are together in the region 1. Now is the time to assign function F3. The size of F3 is 0.5KB, therefore it can be assigned to either region, without violating the size constraint. The interference cost between region 0 and F3, that is, between *main* and F3, is 3.5KB. The interference cost between region 1 and F3 is traditionally computed as the sum of interferences between the functions in region 1 and F3, that is, 2.5KB between F1 and F3, and 1.5 between F2 and F3, totaling to 4KB. Consequently, traditional techniques will assign F3 to region 0 with *main* (shown in Figure 2(c)). Clearly, there is a discrepancy in computing the interference cost between region 1 and function F3. If F2 is also assigned 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 1.5KB. As per this correct interference calculation, F3 should be assigned to region 1 (shown in Figure 2(d)). The required total data transfer between the global 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 assignment, resulting in an 18% savings in data transfers.

3.2. Correct Cost Calculation

The assignment schemes in Pabalkar et al. [2008], Baker et al. [2010], Jung et al. [2010], and Jang et al. [2012] 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 assignment cost between two functions, information such as whether or not the two functions have a 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 Figure 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 a simple interference relationship, previous cost calculation methods may still lead to wrong results. We can

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

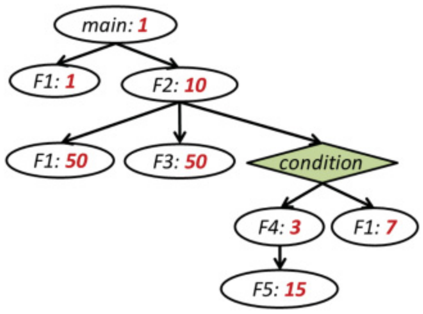

Fig. 3. A call graph for which previous cost calculation methods do not work—The weight on each function node represents how many times the function is called.

see the same program shown in Figure 3 again, where F3 is called by F2 in a loop, F5 is called by F4 in another loop, and the whole subtree rooted by F2 is called in the third loop. We further assume F3 and F5 were assigned to the same region and they were the only functions assigned 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 Pabalkar et al. [2008], Jung et al. [2010], and [Jang et al. 2012], 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 Baker et al. [2010], 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 have overestimated the code management cost.

3.3. Consideration of Branch

The assignment schemes in Pabalkar et al. [2008], Baker et al. [2010], Jung et al. [2010], and Jang et al. [2012] did not consider the existence of *branch* in the program, which leads functions in different branches to have the same degree when considered for assignment. 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 assigned 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 assign F0 and F2 into one region and F1 is left in another separate region. However, without the consideration of *branch*, the algorithm might generate an assignment that F0 and F1 are in a region, and $F2$ 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 assignment.

In this article, we address three aforementioned limitations, and propose correct management overhead estimation for code assignment, as well as two efficient and effective heuristics to assign code for SMM architectures. After this evolution, scratchpad memory can become a better option than cache for instructions on multicore processors, not only because of its power efficiency [Banakar et al. 2002], but also owing to its better performance.

4. 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 a compiler or programmer's hand-inserted code [Banakar et al. 2002]. There are a number of approaches for selecting what to place into the scratchpad memory and when to place them there. Steinke et al. [2002] view the instruction assignment 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. [2006] present an algorithm that looks at time stamps in code sections to determine temporal locality. The work by Janapsatya et al. [2006] and Angiolini et al. [2004] present algorithms that require trace data. Egger et al. [2006, 2010] implement a paged SPM management and prefetching scheme. These schemes require profiling information that is impractical as program execution varies widely on different input data when there are branches. Li et al. [2009] and Li et al. [2005] introduce a general-purpose compiler approach, called memory coloring, to assign static data aggregates in a program to an SPM.

However, these techniques cannot be directly applied to SMM architectures. 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. In contrast, the scratchpad memory is the only memory hierarchy in SMM architectures 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.

In this article, we only solve the problem of code assignment for SMM architectures, assuming data can be properly handled by the works of Bai and Shrivastava [2010], Bai et al. [2011a, 2011b], Bai and Shrivastava [2013b], and Lu et al. [2013]. To the best of our knowledge, the works by Baker et al. [2010], Jang et al. [2012], Jung et al. [2010], and Pabalkar et al. [2008] are similar to our effort and Jung et al. [2010] is the most similar one as only this one updates management cost during the decision of code assignment. Two assignment algorithms were proposed in Jung et al. [2010]. One is Function Mapping by updating and Merging (FMUM) and the other one is Function Mapping by Updating and Partitioning (FMUP). FMUM begins with an assignment 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 space. In contrast, FMUP starts with an assignment where all functions are placed in only one memory region. It repeatedly selects the function that 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 article, we address the problems discussed in Section 3. In the next section, we formally define the code assignment problem. In Section 6, we present a correct cost calculation for code management on SMM architectures, and then our efficient heuristic *CMSM* is presented in Section 7. In Section 8, we further improve the code assignment algorithm by proposing *CMSM_advanced*.

5. PROBLEM DEFINITION

The symbols used in the problem definition are shown as follows:

—S_p: the total size of local scratchpad memory;
—F: a set of functions that are in the program;
—FID: a set of function IDs;
—N: the total number of function IDs in FID;
—$M(fid, f)$: an associate set contains all associations between fid and f, where $fid \in FID$, $f \in F$. Since the assignment relation between a function to function IDs is "one-to-many," we use this set to store all function IDs for a given function f.
—S_{fid}: a set contains function sizes of function ID fid, where $fid \in FID$;
—R: a set of function regions in the local scratchpad memory;
—S_f: a set contains code sizes of functions in the function set F.

Decision variables:

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

Derived variables:

—$y(fid, r)$: {0,1}, indicates whether function I fid is assigned 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 assigned 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 loading/eviction during the execution of program, where $id_a, id_b \in FID$ and $id_a < id_b$.

As mentioned before, a region must accommodate all functions assigned to it. Therefore, its size must be larger than any function assigned to it.

$$\forall r \in R,\ f \in F : S_r \geq x(f, r) * s_f. \tag{1}$$

Since we manage the code at the granularity of function object, a function cannot be split into several different regions, namely, a function f ($f \in F$) can only be assigned to one region.

$$\forall f \in F : \sum_{r \in R} x(f, r) = 1. \tag{2}$$

The size of the total space required for all regions is equal to the sum of sizes of all regions. This size must not exceed the scratchpad memory size predefined.

$$\sum_{r \in R} S_r \leq S_p. \tag{3}$$

Because the assignment relation between function and function ID is "one-to-many," we must update all function IDs associated with a function when the status of a function is changed. In other words, when function f is assigned to a region r, all function IDfid corresponds to f must be also assigned to region r.

$$\begin{gathered}\forall (fid,\ f) \in M(fid,\ f), fid \in FID,\ f \in F, r \in R : \\ y(fid, r) = x(f, r).\end{gathered} \tag{4}$$

The number of functions between two IDs needs to be updated dynamically.

$$num\ (id_a, id_b) = \sum_{r \in R} \sum_{id_a < id < id_b} x(id, r) * y(id_a, r) * y(id_b, r). \tag{5}$$

As discussed in Section 3, function loading/eviction should not be counted on two functions F_1 and F_2, if there are other functions between them. Therefore, there are two cases to be considered when we update *switch* variable. (i) Condition (6): If there are many functions between to-be-determined-two functions, there is no loading/eviction between these two functions. (ii) Condition (7): On the other hand, when there are no other functions between to-be-determined-two functions, loading/eviction must happen between them.

$$(1 - switch\ (id_a, id_b)) * N \geq num\ (id_a, id_b), \tag{6}$$

$$switch\ (id_a, id_b) \geq 1 - num\ (id_a, id_b). \tag{7}$$

We choose the number of instruction transfers from the global memory to the local scratchpad memory as the cost metric. When there is a loading/eviction from function id_a to function id_b, we need to get all instructions of function id_b through DMAs. As a result, 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. In summary, we could formulate our objective function as follows:

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

6. COST CALCULATION

6.1. Graphical Code Representation

In order to correctly calculate the management cost and efficiently assign code, 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. [2008].

6.1.1. Definition of GCCFG.

Definition 1 (*Global Call Control Flow Graph*). A global call control flow graph (V, E) is an ordered acyclic directed graph, where $V = V_F \cup V_L \cup V_C$. 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, and $v_c \in V_C$ 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, Figure 4 shows the GCCFG of the program in Figure 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. To be noted here, for mutual recursive functions in the program, we treat them as one function node in the GCCFG. Namely, when code needs to be fetched/evicted, their instructions will be transferred together.

6.1.2. Construction of GCCFG. In this section, we present our complete algorithm to construct the GCCFG of an application. The input of our algorithm is all CFGs in the program. Then all the CFGs are integrated into a GCCFG in two steps:

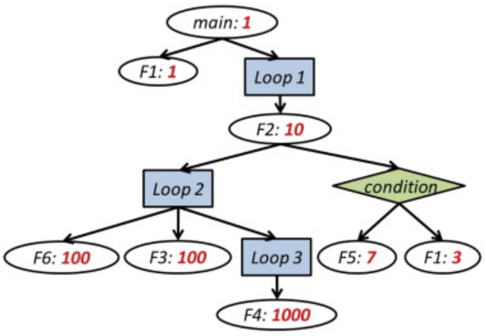

Fig. 4. The GCCFG for the code in Figure 1.

Step 1 (*Extracting Useful Information*): Basic blocks are 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). The basic blocks containing a loop header are labeled as *loop nodes*, those containing a fork point are labeled as *conditional nodes*, and the ones containing a function call are labeled as *function nodes*. If a function is called inside a loop, the corresponding *function node* is joined to the loop header *loop node* with an edge. If any *loop nodes* representing nested loops exist, they are also joined. *Function nodes* not inside any loop are joined to the first node of the CFG. The first node, *function nodes*, *loop nodes*, and corresponding edges are retained, while all other nodes and edges are removed. Essentially, this step trims the CFG, while retaining the control flow and call flow information.

Step 2 (*Merging Subgraphs*): At the second step, all CFGs are merged by combining each *function node* with the first node of the corresponding CFG. The merge ensures that strict ordering is maintained between the CFGs, that is, if two functions are called one after another, the first function is a left child and the other function is a right child of the caller function. One thing that needs to be mentioned herein is that 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.

At the first step, all basic blocks are scanned. Therefore, the complexity is $O(|b|)$, where $|b|$ is the total number of basic blocks in the application. At the second step where most basic blocks are trimmed off, the valid number of nodes is much less than $|b|$. We can denote this total number of nodes in the final GCCFG as $|n|$. When we merge all subgraphs found at the first step, the worst case is that we need to scan all subgraphs for each node. As a result, the time complexity of the GCCFG construction algorithm is $O(|n|^2)$.

Weight Assignment. The weight assignments for *function nodes* usually have two ways: profiling or static estimation. The profiling method is straightforward, as the exact number of times the loop to be executed can be determined by executing the program with its input. For example, we could get the number of iterations of a *while* loop with an input dependent condition. However, profiling is time consuming, as it needs to be done every time the input for the application is changed.

To generalize our work, we expect a static compile-time weight assignment scheme. Here, we present our methodology for estimating the number of function calls on each function node. We assign the weights on the functions by traversing the 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 that is called inside a deeply nested loop will receive a greater

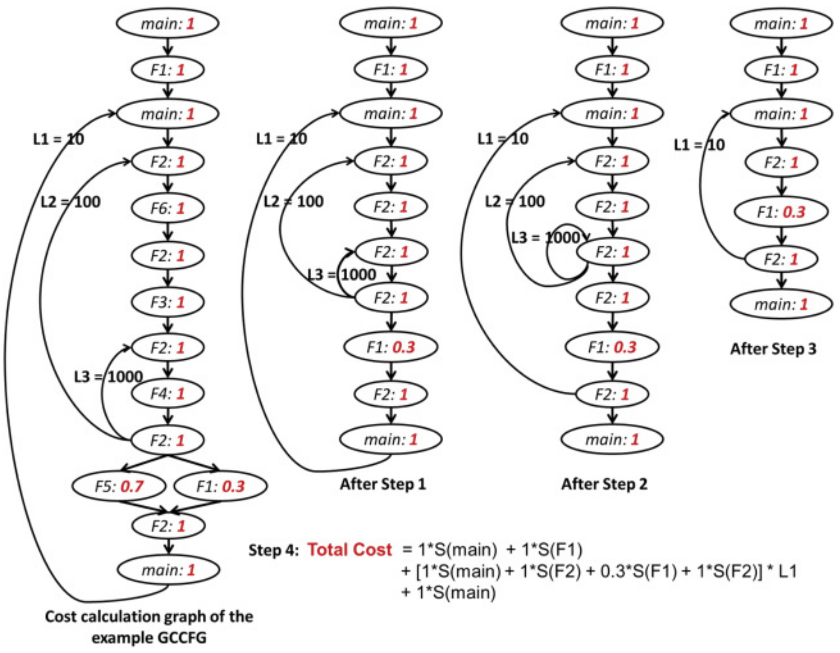

Fig. 5. An example shows calculating the cost of assignment region {main, F1} and region {F2} into one region, when using the cost calculation method in Algorithm 1.

weight than other functions. When a *conditional node* is encountered, the weight on each descendant function node equals 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 Smith [1981] to predict the branch probability. We found the impact of Q is negligible as long as it is larger than 10. Therefore, in the previous Figure 4, Q is chosen to be 10 in the resulted GCCFG of the example code with our static weight assignment scheme.

6.2. Cost Calculation Graph

6.2.1. Definition of Cost Calculation Graph.

Definition 2 (*Cost Calculation Graph*). A cost calculation graph (V, E) is a directed cyclic graph, where $E = E_B \cup E_N$. E_B and E_N are the sets of "backward edges" and "normal edges," respectively. 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.

A cost calculation graph is an estimation of execution trace of a program. The leftmost graph in Figure 5 shows the cost calculation graph of the GCCFG in Figure 4.

6.2.2. Construction of Cost Calculation Graph. As GCCFG is ordered, the cost calculation graph could be constructed by modifying the 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 subtree. When a conditional node is met, we would attach all nodes connected to the 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.

ALGORITHM 1: Algorithm cost (R_a, R_b, G) ▷ G is the cost calculation graph of the managed application; R_a and R_b stand for two separate regions.

```
1: for each vertex $v$ in $G$ do ▷ Step 1: remove all other functions from the graph and just keep
   the functions in the two regions $R_a$ and $R_b$
2:     if $v \notin R_a$ and $v \notin R_b$ then
3:         remove $v$ in $G$ and connect from its source function node to its destination function
   node in a path;
4:     end if
5: end for

6: find all loops in $G$ and put all nodes of each loop into a vector of vector $\mathcal{LV}$.

7: for each loop $l \in \mathcal{LV}$ do                                                      ▷ Step 2
8:     if the first function $v_f$ and the last function $v_l$ in the loop $l$ are identical then
9:         move $v_l$ out of the loop and put it right close to the function after the loop;
10:    end if
11: end for

12: for each function node $v \in G$ do                                                  ▷ Step 3
13:     eliminate identical adjoining functions ($v_n$ after $v$, where there are the same function) in
    the graph;
14:     if $v$ is the only function in a loop then
15:         remove the back edge on $v$ and its corresponding loop count;
16:     end if
17:     if $v$ is the source node of the back edge and its destination node $v_n$ which is beyond the
    loop then
18:         keep $v$ and remove $v_n$;
19:     end if
20:     if more than two outgoing edges on $v$ (i.e., branch) then
21:         save all first nodes $v_{1i}$ on all paths $p_i$ ($p_i \in P$, where $P$ are all outgoing edges connecting
    to $v$) to a set $F$;
22:     end if
23:     set boolean variable $flag$ to 1;
24:     for each node $v_f \in F$ do
25:         if $v_f$ is not the same as $v$ then
26:             $flag \leftarrow 0$;
27:             break;
28:         end if
29:     end for
30:     if $flag == 1$ then
31:         remove all nodes in $F$;
32:     end if
33: end for

34: totalCost ← 0          ▷ Step 4: Calculate the cost totalCost. Rule: cost (F) = 1 × branch
    probability (F), where F is a function in the program. (The function that is not in the branch
    has probability 1).
35: 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.
36: return totalCost;
```

6.3. Cost Calculation Algorithm

Algorithm 1 outlines the cost calculation algorithm. Line 3 removes all nodes that are not in code regions R_a and R_b, since we only calculate the cost by merging functions in two regions together. At step 3, we first eliminate identical adjoining functions in the graph (line 13). If a function is the only function in a loop, then the loop and its

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 ← ∞, 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
```

corresponding loop count are removed (lines 14–16). If two identical function nodes are separated by a loop, then we only keep the one that is inside the loop (lines 17–19). From line 20 to line 22, we collect all first nodes on all paths connecting to the function node v. If they denote the same function as v, then we remove all first nodes in all diverging paths (lines 23–32). At the final step (step 4), we can get the code management cost when merging two regions R_a and R_b (lines 34–36), as we traverse the cost graph from step 1 to step 3. Therefore, the timing complexity of the cost calculation algorithm is $O(|V_f|)$, where $|V_f|$ is the total number of nodes in the cost calculation graph.

Figure 5 shows an illustration of our algorithm. In this example, we are trying to evaluate the cost of assigning region {main, F_1} and region {F_2} in one region. We first remove all irrelevant functions in the program at step 1. Step 2 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 3, 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 assignment cost at step 4, where we consider the impact of branch probability and the existence of loops.

Proof of Correctness. As we only calculate the cost of assigning functions from two regions into one region, the removal of other unrelated functions at step 1 will not change the correctness of our final cost calculation. Step 2 will not affect the correctness of the calculation, since there is no function loading/eviction 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 function loading/eviction right after the loop. When two adjoining functions in the graph are identical, we assert there is no function loading/eviction in the target region. Therefore, we can remove redundant copies at step 3. 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.

7. CODE ASSIGNMENT HEURISTIC: CMSM

Algorithm 2 outlines our CMSM heuristic. It starts with an assignment, in which each function is assigned 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 first 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 (lines 5 and 6).

Function FindMinBalancedMerge() is described in Algorithm 3. To do this, we choose a region pair (R_1, R_2) (Algorithm 3, lines 2–11), and calculate its merge cost at line 5. Here, we utilize the cost function from Algorithm 1. In addition, 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 regions as long as there are functions that could be accommodated to them. 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 result 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, where $|V_f|$ is the total number of function nodes in the GCCFG. 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 that has complexity $O(|V_f|)$. Thus, the worst case timing complexity of CMSM is $O(|V_f|^4)$.

8. CMSM OPTIMIZATION: CMSM_ADVANCED

After running CMSM, we may have unused bytes remaining in the available local scratchpad memory. In the event CMSM returns a solution with unused memory space at least as large as the smallest function, we can further improve the management performance by creating new regions and splitting functions into them. We name this optimization to the CMSM algorithm given in Algorithm 4, CMSM_advanced. At lines 8–20, the algorithm removes functions to new regions beginning with the function having the greatest split benefit as long as they fit into remaining memory and the source region contains more than one function. We continue the algorithm until the remaining memory is too small to hold the smallest function. As this expansion process does not increase the computational complexity more than $O(|V_f|^4)$, the worst case timing complexity of CMSM_advanced is still $O(|V_f|^4)$. Expanding a function to a new region is guaranteed to reduce the management cost since removing a function from a region with multiple functions must reduce the region's interference cost, and adding a function to an empty region results in no additional cost.

9. EMPIRICAL EVIDENCE

9.1. Experimental Setup

We use the IBM Cell processor [Flachs et al. 2006] as our hardware platform. It is a multicore processor, and gives us access to six of the eight SPEs. However, we utilize the main core and only one SPE available in the IBM Cell processor in most of our experiments, except the one designed for demonstrating the scalability of our heuristics in Section 9.7. This architecture has a global memory on the main core, and only a local scratchpad memory on each execution core, or SPE. Scratchpad memory is small, and

Table I. Benchmarks, their Minimum Sizes of Code Space, and Maximum Sizes of Code Space

Benchmark	Number of functions	min code (B)	max code (B)
Adpcm_decoding	13	1,552	6,864
Adpcm_encoding	13	1,568	6,880
BasicMath	20	4,272	12,128
Dijkstra	26	2,496	9,216
FFT	27	2,496	12,776
FFT_inverse	27	2,496	12,776
String_Search	17	632	4,708
Susan_Edges	24	19,356	37,428
Susan_Smoothing	24	19,356	37,428

ALGORITHM 4: Algorithm CMSM_advanced (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
8: while SPMSize() $< \mathcal{S}$ do                          ▷ expand the assigned regions
9:     remainning_space = $\mathcal{S}$ - SPMSize();
10:    max_split = 0, max_f;
11:    for all function $f \in F$, where $F$ = {all functions which are less than remainning_space} do
12:        split = cost($R$, $\emptyset$, GCCFG) - cost($R - \{f\}$, $\emptyset$, GCCFG), $R$ is the region $f$ is originally assigned.
13:        if split > max_split then
14:            max_split = split;
15:            max_f = f;
16:        end if
17:    end for
18:    remove max_f from its current region
19:    create a new region and add max_f
20: end while
```

therefore the program needs to be managed in software when its footprint is larger than the memory available.

The benchmarks used for experimentation are from the Mibench suite [Guthaus et al. 2001] and are presented in Table I. All the information is profiled by compiling programs for SPE. *Number of functions* is the total number of functions in the program, including library functions tailored for SPE. *min code* is the smallest possible assignment size of code space, defined by the size of the largest function in the application. *max code* is the total size of the program.

9.2. Overall Performance Comparison

While the results are scalable for all benchmarks, Figure 6 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,

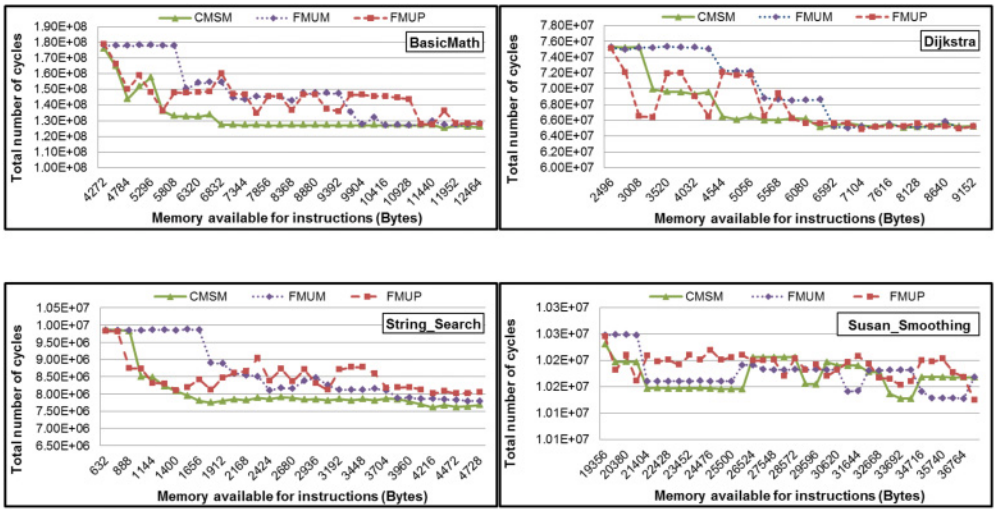

Fig. 6. Performance comparison against FMUM and FMUP.

all heuristics achieve the same assignment, that is, assigning 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 assignment solutions with more regions to give lower overhead costs, as only functions assigned to the same region will swap each other during runtime. 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 9.4. The reverse effect is also visible. When the code size constraint is extremely relaxed, for example, larger than 70% of *max code* present in Table I, all three algorithms again achieve very similar code assignment. This is because there are quite a few functions assigned to one region when the code space is sufficient enough. The small differences in code assignment generate negligible effect on performance. Note that code assignments created by the CMSM do not always outperform the other two heuristics. For example, when memory available for instructions of benchmark "dijkstra" is $3,520$ bytes in Figure 6(b), 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 assignment might be generated. Although our heuristic does not consistently gives good results, it gives better results most of the time. 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 the time.

Another important observation from Figure 6 is that, applications tend to have less execution time when their code space becomes larger. A large code space usually leads to a greater 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 for instructions.

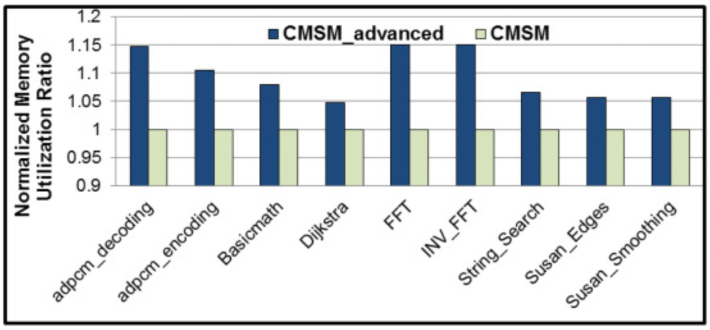

Fig. 7. Memory utilization ratio comparison between CMSM and CMSM_advanced.

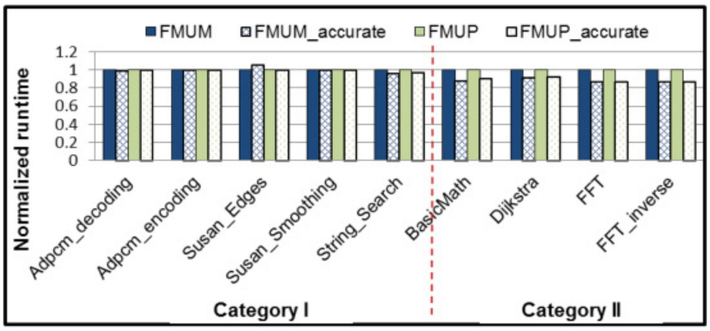

Fig. 8. Impact of accurate cost calculation on FMUM and FMUP.

9.3. Comparison Among CMSM, CMSM_Advanced, and ILP

When we compare the performance of applications that managed with CMSM and that with CMSM_advanced, we notice that optimized algorithm CMSM_advanced produces better code assignment than CMSM. The performance can be improved by an average of 5%, and up to 11%. The improvement is particularly large when memory is severely restricted. The limited performance of the CMSM algorithm is due to the fact that it can produce solutions that do not fully utilize available memory as optimized by the CMSM_advanced algorithm, and the optimized algorithm tends to produce solutions with more regions. As shown in Figure 7, CMSM_advanced can utilize the local scratchpad memory 10% better than CMSM. When compared with ILP results, our greedy algorithm *CMSM_advanced* performs an average of 5% worse.

9.4. Importance of Accurate Cost Calculation

In this experiment, we conducted another set of experiments that evaluates the impact of accurate cost estimation on code assignment 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 Figure 8. The 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 [Jung et al. 2010], but with our correct management overhead calculation instead of theirs. The 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

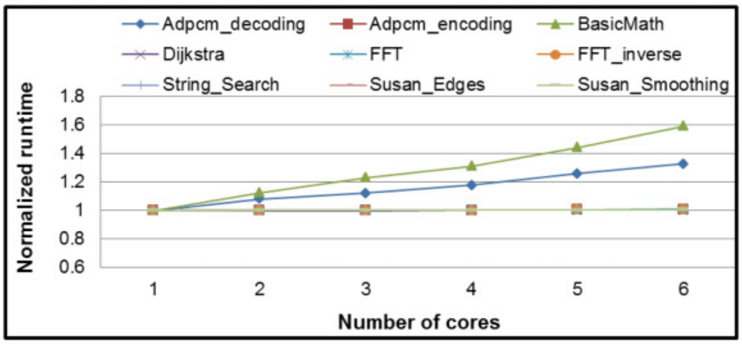

Fig. 9. Scalability of CMSM_advanced on multicore processors.

branches. As shown in Figure 8, programs from the category II benefit from accurate cost calculation. Their performance is 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 assignment for code management.

9.5. 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, CMSM, and CMSM_advanced heuristics, respectively. We found that, even with an increase for our algorithms (less than 5% on average), the compilation time for the set of benchmarks is always less than a minute. This is definitely tolerable for several embedded applications that are compiled once, distributed as binaries, and executed many times. This profile closely fits the kinds of applications that are intended for embedded processors, for example, multimedia and office applications.

9.6. Accuracy of Weight Assignment

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

9.7. Scalability

Figure 9 shows the results when we examined the scalability of our CMSM_advanced heuristic on SMM architectures. We normalized the execution time of each benchmark with the number of SPEs to its execution time with only one SPE, and show them on the y axis. In this experiment, we executed the identical application on different numbers 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 occurs almost at the same time when instructions need to be moved between the global 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. It should be noted that, in this experiment, though different cores are running the same benchmark, a separate binary executable file is generated for each core. In other words, each core has a separate image of the benchmarks that are located in different addresses in the virtual memory space. As a result, this experiment will not profit from the second level cache of CELL.

10. COMPARISON AGAINST HARDWARE CACHE

When there is a code miss in local scratchpad memory (SPM), DMA is required for instruction transfer between global memory and local memory. Likewise, when there is an instruction miss in hardware instruction cache, there is also a penalty for applications. In this section, we show the comparison of code management overhead between SPM-based architecture and cache-based architecture. The management overhead models for them are as follows:

$$O(cache) = N_{misses} * penalty(miss),$$
$$O(SPM) = N_{insns} * cycle(insn)/freq + \sum DMA\_cost(f),$$

where N_{misses} is the total number of cache misses, $penalty(miss)$ is cache penalty per miss, N_{insns} is the total extra instructions incurred by code management library function *__ovly_load()*, $cycle(insn)$ is cycles per instruction, $freq$ is the system frequency, and f means a function. $\sum DMA\_cost(f)$ is the total DMA cost by transferring instructions. The DMA overhead measurements taken on the IBM Cell processor can be modeled to within an average error of 0.4% using the following equation [Kistler et al. 2006]:

$$DMA\_cost\ (f) = \begin{cases} 3.9E-5 * size\ (f) + 0.17\mu s & \text{if } size\ (f) \leq 2kB; \\ 7.3E-5 * size\ (f) + 0.1\mu s & \text{if } size\ (f) > 2kB. \end{cases} \quad (8)$$

We collect instruction cache misses (N_{misses}) through SimpleScalar [Austin et al. 2002] by configuring a four-way associative instruction cache with the size equaling the size of code space in the local scratchpad memory. In addition, we compute the number of extra instructions (N_{insns}) as the average number of instructions per function *__ovly_load()* call multiplied by the total number of function calls in the managed application. The number of instructions of *__ovly_load()* is estimated statically by manually going through its disassembly code.

In this experiment, for each benchmark, we set the scratchpad memory size and cache size as 2X bytes, varying from the minimum size to the maximum size and doubling the space at each step. In addition, we use the average number of misses and the average management overhead of nine applications to demonstrate the design alternative. When counting the miss penalty for hardware cache, we vary the miss penalty per miss from 0.01ps to 1000ns. As shown in Figure 10, when the penalty per miss ($penalty(miss)$) of cache approaches 260 pico seconds, the total miss penalty of cache is similar to our CMSM_advanced. When $penalty(miss)$ is less than 260ps, cache achieves better performance. We believe it is impossible, as 260ps are merely 0.83 cycle on the Cell processor [Flachs et al. 2006]. Under feasible circumstance, our CMSM_advanced has much less penalty than cache has. The main reason for less overhead of software scheme is that, the management granularity is coarser with software scheme (namely, function object granularity), but the cache line sizes in cache-based architectures are not large enough. We can further deduce less total execution time from less penalty on SPM-based architectures. Since scratchpad memory is more power efficient than

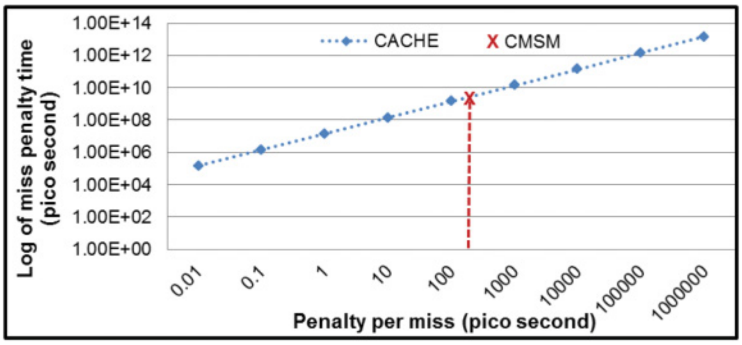

Fig. 10. Performance comparison against cache.

Table II. Comparison between Hardware Management Technique and Software Management Technique

Compare	Hardware management	Software management
Performance	Worse than cache	Better than cache
Hardware Overhead	Yes	No
Software Overhead	No	Yes
Flexibility	Bad	Good

a cache [Banakar et al. 2002], the less execution time of applications on SPM-based architectures implies less energy consumption.

11. COMPARISON AGAINST HARDWARE MANAGED SCRATCHPAD MEMORIES

In this section, we compare our technique with hardware managed scratchpad memories. A hardware solution of code management for scratchpad memory has been proposed in 2011 [Metzlaff et al. 2011]. Another similar technique has been presented by Schoeberl [2009]. Hardware techniques use extra hardware components to manage code. However, it will incur extra power consumption and extra chip area overhead. Compared to hardware managed scratchpad memories, our technique can manage application code without any hardware overhead. Moreover, our software code management technique is very flexible. Instead of managing code for every function call, it inserts management library functions only when the code management is needed. This largely reduced the average software overhead caused by the extra management library instructions. However, hardware managed scratchpad memories incur extra hardware overhead anyway, even for the cases when data management is not needed. As for performance, hardware managed scratchpad memories cannot compete with caches in terms of performance [Metzlaff et al. 2011], while as proved in Section 10, our management technique can outperform caches. The comparison results are shown in Table II.

12. SUMMARY

SMM architectures are one of the promising solutions to the problem of scaling the memory hierarchy. However, since scratchpad memory cannot always accommodate the whole task assigned to it, certain schemes are required to mange code, global data, stack data, and heap data of the program to enable its execution. This article focuses on managing code between the global 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 article, we formally define the problem of assigning code for

SMM architectures at the granularity of function objects. In addition, we propose a correct cost calculation for code assignment, as well as two heuristic approaches named CMSM and CMSM_advanced for the same problem. Our experimental results show that CMSM and CMSM_advanced generate code assignments that lead to significant performance improvement compared to previous work. When compared to hardware cache, we also found that scratchpad memory performs much better.

REFERENCES

Federico Angiolini et al. 2004. A post-compiler approach to scratchpad mapping of code. In *Proceedings of the 2004 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'04)*. 259–267.

Todd Austin, Eric Larson, and Dan Ernst. 2002. SimpleScalar: An infrastructure for computer system modeling. *Computer* 35, 2 (Feb. 2002), 59–67.

Ke Bai, Di Lu, and Aviral Shrivastava. 2011a. Vector class on limited local memory (LLM) multi-core processors. In *Proceedings of the 14th International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES'11)*. 215–224.

Ke Bai, Aviral Shrivastava, and Saleel Kudchadker. 2011b. Stack data management for limited local memory (LLM) multi-core processors. In *Proceedings of the 2011 IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*. 231–234.

Ke Bai, Jing Lu, Aviral Shrivastava, and Bryce Holton. 2013. CMSM: An efficient and effective code management for software managed multicores. In *2013 IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. 1–9.

Ke Bai and Aviral Shrivastava. 2010. Heap data management for limited local memory (LLM) multi-core processors. In *Proceedings of the 8th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'10)*. 317–326.

Ke Bai and Aviral Shrivastava. 2013a. A software-only scheme for managing heap data on limited local memory (LLM) multicore processors. *ACM Transactions on Embedded Computing Systems (TECS)* 13, 1 (2013), 5.

Ke Bai and Aviral Shrivastava. 2013b. Automatic and efficient heap data management for limited local memory multicore architectures. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'13)*. 593–598.

Michael A. Baker, Amrit Panda, Nikhil Ghadge, Aniruddha Kadne, and Karam S. Chatha. 2010. A performance model and code overlay generator for scratchpad enhanced embedded processors. In *Proceedings of the 2010 IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'10)*. 287–296.

Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, Mahesh Balakrishnan, and Peter Marwedel. 2002. Scratchpad memory: Design alternative for cache on-chip memory in embedded systems. In *Proceedings of the 10th International Symposium on Hardware/Software Codesign*. 73–78.

Garo Bournoutian and Alex Orailoglu. 2011. Dynamic, multi-core cache coherence architecture for power-sensitive mobile processors. In *Proceedings of CODES+ISSS*. 89–98.

Byn Choi, Rakesh Komuravelli, Hyojin Sung, Robert Smolinski, Nima Honarmand, Sarita V. Adve, Vikram S. Adve, Nicholas P. Carter, and Ching-Tsun Chou. 2011. DeNovo: Rethinking the memory hierarchy for disciplined parallelism. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques (PACT'11)*. 155–166.

Benoît Dupont de Dinechin, Pierre Guironnet de Massas, Guillaume Lager, Clément Léger, Benjamin Orgogozo, Jérôme Reybert, and Thierry Strudel. 2013. A distributed run-time environment for the Kalray MPPA®-256 integrated manycore processor. *Procedia Computer Science* 18 (2013), 1654–1663.

Bernhard Egger, Seungkyun Kim, Choonki Jang, Jaejin Lee, Sang Lyul Min, and Heonshik Shin. 2010. Scratchpad memory management techniques for code in embedded systems without an MMU. *IEEE Transactions on Computers* 59, 8 (2010).

Bernhard Egger, Jaejin Lee, and Heonshik Shin. 2006. Scratchpad memory management for portable systems with a memory management unit. In *Proceedings of the 6th ACM & IEEE International Conference on Embedded Software (EMSOFT'06)*. 321–330.

Brian Flachs, Shigehiro Asano, Sang Dhong, Peter Hofstee, Gilles Gervais, Roy Kim, Tien Le, Peichun Liu, Jens Leenstra, John Liberty, Brad Michael, Hwa-Joon Oh, Silvia Melitta Mueller, Osamu Takahashi, Akiyuki Hatakeyama, Yukio Watanabe, Naoka Yano, Daniel A. Brokenshire, Mohammad Peyravian, VanDung To, and Eiji Iwata. 2006. The microarchitecture of the synergistic processor for a cell processor. *IEEE Solid-State Circuits* 41, 1 (2006), 63–70.

Antonio García-Guirado, Ricardo Fernández-Pascual, Alberto Ros, and José M. García. 2011. Energy-efficient cache coherence protocols in chip-multiprocessors for server consolidation. In *Proceedings of the 2011 International Conference on Parallel Processing (ICPP'11)*. 51–62.

Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of Workload Characterization*. 3–14.

Bryce Holton, Ke Bai, Aviral Shrivastava, and Harini Ramaprasad. 2014. Construction of GCCFG for interprocedural optimizations in software managed manycore (SMM) architectures. In *2014 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES'14)*. 1–10.

IBM. 2006. *Programmer's Guide: Software Development Kit for Multicore Acceleration Version 3.1*. Technical Report.

Intel. 2010. Intel core i7 processor extreme edition and intel core i7 processor datasheet, volume 1. In *White paper*. Intel.

Intel. 2012. The SCC Programmer's Guide. https://communities.intel.com/servlet/JiveServlet/previewBody/5684-102-8-22523/SCCProgrammersGuide.pdf. (2012).

Andhi Janapsatya, Aleksandar Ignjatović, and Sri Parameswaran. 2006. A novel instruction scratchpad memory optimization method based on concomitance metric. In *Proceedings of the Asia and South Pacific Conference on Design Automation (ASP-DAC)*. 612–617.

Choonki Jang, Jaejin Lee, Bernhard Egger, and Soojung Ryu. 2012. Automatic code overlay generation and partially redundant code fetch elimination. *ACM Transactions on Architecture and Code Optimization* 9, 2 (June 2012), 10:1–10:32.

Seung Chul Jung, Aviral Shrivastava, and Ke Bai. 2010. Dynamic code mapping for limited local memory systems. In *Proceedings of the 21st IEEE Internatonal Conference on Application-Specific Systems Architectures and Processors (ASAP'10)*. 13–20.

Michael Kistler, Michael Perrone, and Fabrizio Petrini. 2006. Cell multiprocessor communication network: Built for speed. *IEEE Micro* 26, 3 (May 2006), 10–23.

Lian Li, Hui Feng, and Jingling Xue. 2009. Compiler-directed scratchpad memory management via graph coloring. *ACM Transactions on Architecture and Code Optimization* 6, 3, Article 9 (Oct. 2009), 17 pages.

Lian Li, Lin Gao, and Jingling Xue. 2005. Memory coloring: A compiler approach for scratchpad memory management. In *Proceedings of 14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*. 329–338.

Jing Lu, Ke Bai, and Aviral Shrivastava. 2013. SSDM: Smart stack data management for software managed multicores (SMMs). In *Proceedings of the 50th Annual Design Automation Conference (DAC'13)*. 149–156.

Stefan Metzlaff, Irakli Guliashvili, Sascha Uhrig, and Theo Ungerer. 2011. A dynamic instruction scratchpad memory for embedded processors managed by hardware. *Architecture of Computing Systems* 6566 (2011), 122–134.

Pierre Michaud, André Seznec, Damien Fetis, Yiannakis Sazeides, and Theofanis Constantinou. 2007. A study of thread migration in temperature-constrained multicores. *ACM Transactions on Architecture and Code Optimization* 4, 2, Article 9 (2007).

Amit Pabalkar, Aviral Shrivastava, Arun Kannan, and Jongeun Lee. 2008. SDRM: Simultaneous determination of regions and function-to-region mapping for scratchpad memories. In *Proceedings of 15th International Conference on High Performance Computing (HPC'08)*. 569–582.

Martin Schoeberl. 2009. Time-predictable cache organization. In *Software Technologies for Future Dependable Distributed Systems*. 11–16.

James E. Smith. 1981. A study of branch prediction strategies. In *Proeedings of 8th Annual Symposium on Computer Architecture (ISCA'81)*. 135–148.

Stefan Steinke, Nils Grunwald, Lars Wehmeyer, Rajeshwari Banakar, Mahesh Balakrishnan, and Peter Marwedel. 2002. Reducing energy consumption by dynamic copying of instructions onto on-chip memory. In *Proceedings of 15th International Symposium on System Synthesis (ISSS'02)*. 213–218.

Tom's Hardware. 2010. Raw performance: SiSoftware sandra 2010 pro (GFLOPS).

Loc Truong. 2009. *Low Power Consumption and a Competitive Price Tag Make the Six-Core TMS320C6472 Ideal for High-Performance Applications*. Technical Report. Texas Instruments.

Sumesh Udayakumaran, Angel Dominguez, and Rajeev Barua. 2006. Dynamic allocation for scratch-pad memory using compile-time decisions. *Transactions on Embedded Computing Systems* 5, 2 (2006), 472–511.

Kaushik Vaidyanathan, Qiuling Zhu, Lars Liebmann, Kafai Lai, Stephen Wu, Renzhi Liu, Yandong Liu, Andzrej Strojwas, and Larry Pileggi. 2015. Exploiting sub-20-nm complementary metal-oxide semiconductor technology challenges to design affordable systems-on-chip. *Journal of Micro/Nanolithography, MEMS, and MOEMS* 14, 1 (2015), 011007–011007.

Manish Verma and Peter Marwedel. 2006. Overlay techniques for scratchpad memories in low power embedded processors. *IEEE VLSI* 14, 8 (2006), 802–815.

Yi Xu, Yu Du, Youtao Zhang, and Jun Yang. 2011. A composite and scalable cache coherence protocol for large scale CMPs. In *Proceedings of the International Conference on Supercomputing (ICS'11)*. 285–294.

Received February 2014; revised October 2014; accepted February 2015