An Automatic Code Overlaying Technique for Multicores with Explicitly-Managed Memory Hierarchies*

Choonki Jang
Samsung Advanced Institute of Technology
Samsung Electronics, Yongin 446-712, Korea
choonki.jang@samsung.com

Jun Lee Sangmin Seo Jaejin Lee
School of Computer Science and Engineering
Seoul National University, Seoul 151-744, Korea
{jun,sangmin}@aces.snu.ac.kr, jlee@cse.snu.ac.kr

ABSTRACT

The explicitly-managed memory hierarchies, where a hierarchy of distinct memories is exposed to the programmer and managed explicitly by software, are not only found in typical embedded processors but also found in a class of high performance multicore architectures. Code overlay techniques have been widely used to execute a program whose code is bigger than the available code memory in the system. To generate an efficient overlaid executable with maximum storage savings as well as minimum performance overhead, the overlay structure should be designed carefully. In this paper, we propose an efficient code overlay technique that automatically generates an overlay structure for a given memory size for multicores with explicitly-managed memory hierarchies. We observe that finding an efficient overlay structure with minimum memory copying and run-time check overhead is similar to the problem that finds a code placement with minimum conflict misses in the instruction cache. Our algorithm exploits the temporal-ordering information between functions during program execution. The information is obtained from profiling the program. Experimental results with 11 parallel applications on the Cell BE processor indicate that our approach is effective and promising.

Categories and Subject Descriptors

D.3.4 [**PROGRAMMING LANGUAGES**]: Processors—*Code generation, Compilers, Optimization*

General Terms

Algorithms, Design, Experimentation, Measurement, Performance

Keywords

Code overlays, Temporal ordering, Temporal-relationship graph

*This work was supported in part by grant 2009-0081569 (Creative Research Initiatives: Center for Manycore Programming) from the National Research Foundation of Korea.This work was also supported in part by the Ministry of Education, Science and Technology of Korea under the BK21 Project. ICT at Seoul National University provided research facilities for this study.

1. INTRODUCTION

As the semiconductor fabrication technology scales down, the number of cores in a processor is rapidly increasing. Undoubtedly, multicore computing is now a mainstream. The design and implementation of caches for multicore processors is much more complicated than those for uniprocessors due to the coherence problem introduced by multiple copies of a shared memory block.

Among others, the directory-based cache coherence scheme [4, 22] is well known that it is scalable to the number of cores but its implementation involves expensive hardware cost [5, 13, 30]. For this reason, explicitly managed memory hierarchies, where the programmer or software is responsible for managing code or data between different levels of the hierarchy, can also be found in recent high performance multicores, such as the Intel single-chip cloud computer (SCC) [19], Cell BE [18], Terascale [32], and Cyclops64 [6].

In these processors, a *local store* is tightly coupled to each processor core or each group of processor cores instead of the hardware-based cache coherence protocol. The local store needs to be managed explicitly by the programmer or software. Moreover, they are in charge of memory coherence and consistency.

To alleviate the burden of the programmer from explicitly managing shared data, a *software caching* technique [17] is typically used, especially for Cell BE processors. Software caching partly hides the difficulties of accessing shared data from the programmer when there is no virtual memory available. Some amount of memory space in the local store is dedicated to cache the shared data. Obviously, the larger the available memory space for the software cache, the better the performance. However, the local store is used not only for data but also for code in the Cell BE processor and its size is limited to 256KB. Thus, code memory space reduction in the local store is a very important issue to achieve high performance with the Cell BE processor.

To reduce memory footprints due to instruction fetches, an overlay structure is required unless virtual memory is available. The overlay structure is generated manually by the programmer or automatically by a specialized linker [7, 17, 23]. Manual code overlaying requires the programmer to deeply understand the program to achieve maximum memory savings as well as minimum performance degradation. Even though traditional code overlaying techniques can significantly mitigate the programmer's burden to build an overlaid executable, they have some limitations. First, traditional algorithms do not consider the degree of conflicts between two different code segments when they are mapped to the same memory region. This results in generating an inefficient overlay structure. Second, traditional techniques may cause much redundant memory copying due to the difference in the size of overlaid code segments in the same memory region. Finally, traditional techniques still require user intervention to generate an efficient overlay structure that maximizes memory savings as well as minimizes performance degradation.

In this paper, we propose an efficient automatic code overlaying technique for multicores with explicitly-managed memory hierarchies. We are dealing with a multicore context in which there is no virtual memory for the cores with the explicitly-managed memory hierarchy. In addition, a function is the granularity of the faulting mechanism in our overlaying technique. This incurs heavy run-time checking and memory copying overhead and requires overhead minimization. This paper addresses these two issues as efficiently as possible.

Our approach exploits the temporal relationship graph (TRG) proposed by Gloy *et al.* [11]. It summarizes the temporal-ordering information between functions during program execution and is built upon the profiling information. The TRG estimates the degree of conflict misses between two different code segments when they are overlaid in the same memory region. We build a linker that generates an executable for profiling the given application and then generates an overlaid executable for the given code memory space with the profiling information. The major contributions of this paper are as follows:

- We propose a greedy algorithm that generates an efficient overlay structure for a given code memory space using the TRG.
- We propose an overlay management technique that minimizes the run-time checking and memory copying overhead.
- We show the effectiveness of our approach by implementing a linker for the Cell BE processor. We compare the memory savings and execution time of our approach with those of the available tools for the Cell BE processor using 11 benchmark applications.

The rest of this paper is organized as follows: Section 2 discusses related work. A typical overlaying mechanism is briefly described in Section 3. In Section 4, we explain the temporal-ordering information between functions. The overlay generation algorithm is described in Section 5. Section 6 presents the implementation of our overlay manager and the method to insert the glue code. Section 7 presents the evaluation result of our technique. The paper concludes in Section 8.

2. RELATED WORK

Cytron and Loewner [7] propose a technique that automatically generates an overlay structure for a given program. It is a traditional tree-based overlaying technique that a typical linker implements. Pabalkar *et al.* [25] propose an automatic overlay generation algorithm with a static analysis that abstracts the behavior of the program. For the kernel code of an operating system, He *et al.* [15] propose an on-demand code overlaying technique that keeps infrequently executed kernel code segments in the secondary storage and loads it on-demand.

Scratch-pad memory (SPM) is a sort of explicitly-managed L1 memory and has been studied a lot in embedded systems to reduce power consumption or improve performance. The SPM can be easily found in embedded processors such as ARM embedded and application processors [2] and TI digital signal processors [31]. Code management techniques for the SPM can be roughly categorized into two classes: *static* [1, 3, 29, 33] and *dynamic* [8, 9, 10, 34] placement techniques. Static techniques [1, 3, 29, 33] do not change the contents of the SPM until the application terminates. Dynamic placement techniques can place more code in the SPM than static approach. Most of these SPM management techniques assume that the processor core fetches instructions directly from the external memory or an instruction cache. Thus, these techniques are not directly applicable to the system, such as Cell BE architectures [18], in which a core cannot fetch instructions directly from the external memory.

The code overlay problem is similar to the code placement problem that tries to minimize cache misses. To improve instruction cache performance, Pettis and Hansen [26] propose a code positioning technique with a weighted call graph (WCG) in which each edge is annotated with the call counts. The algorithm proposed by Hashemi *et al.* [14] also exploits a WCG. However, WCGs have a limitation that it cannot reflect temporal conflict misses between two procedures that are not directly connected, i.e., they do not call each other [11, 20]. To estimate the number of conflict misses more accurately than WCGs, Kalamatianos *et al.* [20] propose a conflict miss graph (CMG) and Gloy *et al.* [11] propose a temporal relationship graph (TRG). Both the CMG and TRG can estimate the degree of temporal conflicts between any pair of functions. The CMG estimates the worst-case number of conflict misses between functions in a granularity of a cache block. On the other hand, the TRG summarizes the alternating behavior between each pair of functions. Since a function is the granularity of the faulting mechanism in our approach, the TRG is more appropriate than the CMG. Finding an optimal code placement that minimizes cache conflicts using the TRG is proved to be an NP-complete problem by Guillon *et al* [12].

3. CONVENTIONAL CODE OVERLAYS

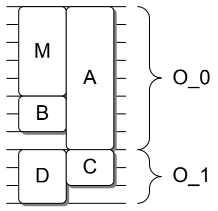
Code overlaying is a widely used technique to run a program whose binary size is larger than the available code memory size when there is no virtual memory available [23]. An *overlay structure* specifies the code layout in the available code memory space. The linker typically provides the user with its own command language in which an overlay structure can be described. An overlay structure consists of one or more *overlay sections*. Each overlay section consists of one

```
void M() {
  for (i=0; i<8; ++i) {
    A();
    B();
  }
}

void A() {
  for (i=0; i<6; ++i) {
    C();
  }
}

void B() {
  for (i=0; i<4; ++i) {
    C();
    D();
  }
}

void C() {
  ...
}

void D() {
  ...
}
```

Figure 1: A sample program.

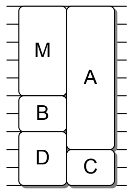
```
OVERLAY O_0 {
  SEGMENT S_MB { M,B }
  SEGMENT S_A  { A }
}
OVERLAY O_1 {
  SEGMENT S_D  { D }
  SEGMENT S_C  { C }
}
```

(a)



(b)

Figure 2: An overlay structure for the program shown in Figure 1. The linker commands in (a) specifies the overlay structure in (b).



(a)

```
OVERLAY O_0 {
  SEGMENT S_MBD { M,B,D }
  SEGMENT S_AC  { A,C }
}
```

(b)

Figure 3: Example of more compact overlay structure. The linker commands in (b) can describe the overlay structure in (a) but the result overlaid executable will not be efficient than expected.

or more code *segments* that overlay each other. Thus, the size of an overlay section is the size of the largest code segment in the overlay section. A code segment is the smallest unit that can be loaded into the memory as a logical entity during execution. The granularity of a code segment is typically an object file or a function.

Figure 2 shows an overlay structure for the program shown in Figure 1. There are five functions in the program. A function M is the entry point of the program. The sizes of function M, A, B, C and D are 5, 8, 2, 2 and 3, respectively. The `SEGMENT` command in Figure 2(a) defines a code segment with a set of functions or object files. The `OVERLAY` command in Figure 2(a) defines an overlay section with a set of code segments. For example, the overlay section O_1 consists of two code segments S_C and S_D. The sizes of overlay sections O_0 and O_1 are 8 and 3, respectively. The memory requirement of the result overlaid executable is 11, which is much less than the total code size 20.

For each inter-segment call, i.e., the caller function and the callee function are not in the same code segment, the overlaid executable must ensure that the target code segment resides in the memory before the control is transferred to the target code segment. If the target code segment is not loaded in the memory, the overlaid executable must load the target code segment into the memory as specified in the overlay structure. This process is performed by an *overlay manager* and each inter-segment call must pass through the overlay manager. To support this, the linker interposes the *glue code* in front of each inter-segment call. To keep track of which code segment is loaded for each overlay section, the linker generates a *segment table* that is managed by the overlay manager.

This conventional overlaying technique has two drawbacks. First, it may introduce redundant memory copying. When a code segment x is being loaded in the memory, the other code segment y that has been loaded and placed in the overlay section of x is discarded because y will be overwritten by x. This is true only if the entire memory region of y is overwritten by x. For example, the code segment `S_MB` has been loaded in the memory and `S_A` is being loaded. Then, `S_MB` is fully overwritten by `S_A`. Now, `S_MB` is being loaded. Then, `S_A` is partially overwritten by `S_MB` and the memory region of `S_A` that does not overlap with `S_MB` will still be in the memory. Thus, when `S_A` is loaded again, only the overwritten part of `S_A` needs to be loaded in the memory.

Second, it is hard and time-consuming for the programmer to describe an efficient overlay structure in the command language provided by a linker. For example, the programmer wants to generate an overlay structure shown in Figure 3(a) for the program shown in Figure 1. With a typical linker, there is no other way to describe the overlay structure shown in Figure 3(a) than the linker script shown in Figure 3(b). Even though this overlay structure consumes less memory than that of the overlay structure shown in Figure 2(b), it loads not only C but also A at every time B calls C because A and C are in the same code segment `S_AC`. On the other hand, the overlay structure in Figure 2(b) will load only C at every time B calls C, resulting in less execution time even though it consumes more memory.

Our automatic code overlaying technique overcomes these drawbacks. Our linker automatically generates an efficient overlay structure that avoids redundant code copying and loading using the temporal-ordering information between functions and function slicing.

4. TEMPORAL ORDERING

To minimize the memory requirement, we may generate an overlay structure by placing every code segment at the same memory region. The overlaid executable obtained by this approach runs successfully if the available memory is larger than the biggest code segment in the program. However this may cause significant overhead due to memory copying.

When two different code segments x and y overlap each other in the memory, we say that x and y *conflict* with each other. A *conflict miss* occurs when a code segment is being referenced by the program and does not reside currently in the memory because it has been replaced with another code segment. Obviously, the more conflict misses, the bigger memory copying overhead. To generate an efficient overlay structure, the conflict misses due to the overlay structure must be minimized. We observe that the conflict miss in overlaid executable is similar to the conflict miss in the conventional direct-mapped instruction cache.

To get better locality in the instruction cache, many code placement techniques have been proposed because the code layout directly affects the behavior of the instruction cache. Among them, we apply the concept of *temporal-ordering* information proposed by Gloy *et al.* [11] to our automatic overlay generation problem. For example, consider the following call trace of the program in Figure 1:

$$\mathsf{M(A(C)^6B(CD)^4)^8}$$

When a function returns to its caller, the caller will be referenced again. Thus, the corresponding call/return trace is as follows:

$$\mathsf{M(A(CA)^6MB(CBDB)^4M)^8}$$

The temporal-ordering information between functions can be obtained from this call/return trace. Gloy *et al.* [11] propose the *temporal relationship graph* (TRG) that summarizes the temporal-ordering information obtained by profiling the program. Each node in the TRG represents a function in the program. There exists an edge between two functions x and y if there exists a reference to y between two consecutive references to x in the call/return trace. Each edge is annotated by a weight $W(x, y)$ that represents the number of times two consecutive references of x are interleaved by at least one reference to y, or vice versa. This weight approximates the degree of conflict misses between x and y during program execution.

For example, the number of conflict misses between two functions A and B can be figured out from the following simplified call/return trace:

$$\mathsf{(A(A)^6B(BB)^4)^8}$$

It is obtained by removing any functions other than A and B from the original call/return trace. Since we are interested in the alternating behavior between references to A and B, it can be further simplified as follows:

$$\mathsf{(AB)^8}$$

In this simplified call/return trace, the number of occurrences of ABA and BAB is 14 as described in Figure 4, and it becomes the weight of an edge (A, B) in the TRG.

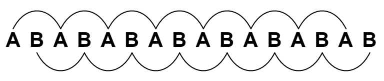


Figure 4: Alternations between A and B.

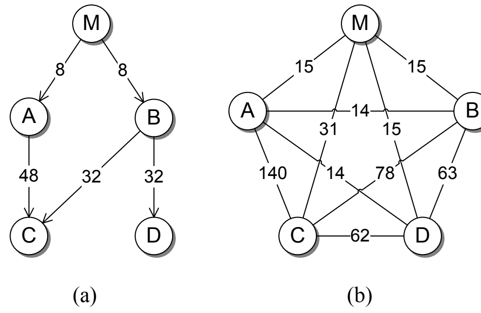


Figure 5: After profiling the program in Figure 1, we obtain (a) the weighted call graph and (b) the temporal relationship graph.

After profiling the program in Figure 1, we obtain the weighted call graph (WCG) and the TRG of the program as shown in Figure 5(a) and Figure 5(b), respectively. The TRG is more general than the WCG because the TRG can contain edges for any pair of functions even though they do not call each other [11]. For example, A and B are not connected in the WCG shown in Figure 5(a). However, when A and B are placed in the same memory region, we can expect that A and B will cause conflict misses that are proportional to the number of iterations of a loop in M. To obtain a better overlay structure, our overlay structure generation algorithm exploits the TRG and each function forms an individual code segment.

5. OVERLAY GENERATION

Our algorithm generates an efficient code overlay structure for a given memory size by exploiting the TRG of a given program. Unfortunately, the problem that identifies an optimal code placement using the TRG is proved to be NP-complete by Guillon *et al.* [12]. Thus, we propose a greedy algorithm that selects the heaviest edge in the TRG and merges the two nodes into one until the TRG has no more remaining edges.

5.1 Overlay Generation Algorithm

Our algorithm considers not only the degree of conflict misses, but also the degree of run-time checks (i.e., the glue code mentioned in Section 3 to ensure the residence of each function in the memory when the function is called). The overhead due to the run-time checks is proportional to the number of invocations of overlaid functions. Consequently, to generate an efficient overlay structure for a given memory size, we need to minimize the overhead due to the runtime checks.

Suppose that we have the TRG shown in Figure 5(b) and the available code memory size, S_{mem}, is 12. Figure 6 illustrates our overlay generation algorithm step by step from this TRG. The heaviest edge in the TRG is (A, C). Thus, nodes A and C are merged into node AC and edge (A, C) is

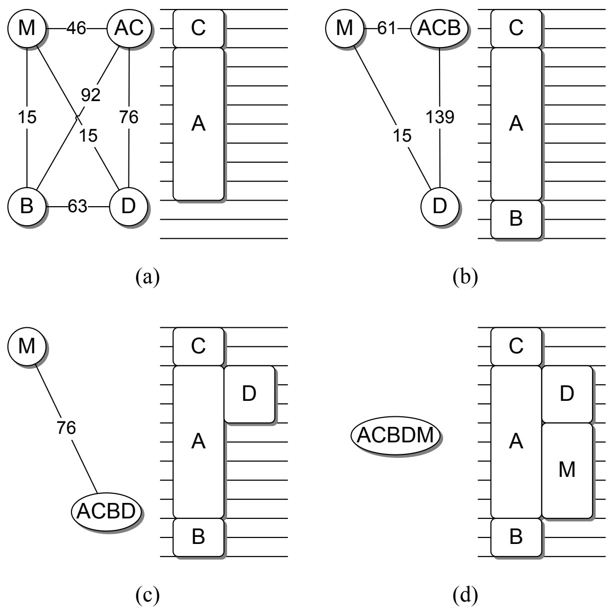


Figure 6: Overlay generation steps for the program in Figure 1 using the TRG shown in Figure 5(b).

removed as shown in the TRG in Figure 6(a). Edges (x, A) and (x, C), where $x \notin \{\mathsf{A}, \mathsf{C}\}$, are combined together into a single edge (x, AC). Its weight is equal to the sum of weights of the combined edges. A and C are placed in the next available memory region. When two nodes of the heaviest edge have not been placed in the memory, the one with a larger number of invocations will precede the other. C has been placed earlier than A in Figure 6(a). Now, the heaviest edge in the TRG in Figure 6(a) is (AC, B). Our algorithm places only B because A and C have been placed. Our algorithm merges AC and B into ACB and places B in the next available memory region as shown in Figure 6(b).

The heaviest edge in the TRG in Figure 6(b) is (ACB, D). Thus, ACB and D are merged into ACBD as shown in Figure 6(c). However, there is no remaining memory region that is large enough to place D. In this case, our algorithm finds a proper location for D by scanning from the beginning of the memory (e.g., offset 0) to offset S_{mem} − size of D by incrementing the offset by S_{inc} at a time. S_{inc} should be selected carefully for the following reasons:

- S_{inc} must be a multiple of the minimum instruction address alignment requirement of the target architecture.
- The greater S_{inc}, the less candidate locations to be tried.
- When overlaid functions are loaded into the memory by exploiting a block transfer functionality (e.g., DMA), S_{inc} must be a multiple of the address alignment requirement of the block transfer functionality.

In addition, to place D at a proper location, we define the cost function $Cost(x, p)$ of a function x at an offset p. The

```
WCG : function → int
TRG : function × function → int
MAP : function → {int offset, bool conflict}
$S_{mem}$ :  the size of available memory

void place($x$) {
  $S_x$ := the size of $x$
  best_offset := 0, best_metric := INFINITE;
  best_F := ∅, best_G := ∅;

  foreach i in (0,1, ...,$\lfloor(S_{mem} - S_x)/S_{inc}\rfloor$) {
    offset := $S_{inc}$ × i;
    // compute $F$, $G$ and cost metric at offset
    F := ∅, G := ∅;
    metric_loading := 0, metric_checks := 0;
    foreach $y$ in MAP {
      $S_{xy}$ := overlap_size(offset,$x$,$y$);
      if ($S_{xy}$ > 0) {
        F := F ∪ {$y$};
        metric_loading += TRG[$x$,$y$] × $S_{xy}$;
        if (MAP[$y$].conflict = false) {
          G := G ∪ {$y$};
          metric_checks += WCG[$y$];
        }
      }
    }
    if (F ≠ ∅) {
      metric_checks += WCG[$x$];
    }
    metric := $\alpha$ × metric_loading + $\beta$ × metric_checks;
    if (metric < best_metric) {
      best_metric := metric, best_offset := offset;
      best_F := F, best_G := G;
    }
  }
  // update $x$'s offset and conflict in MAP
  MAP[$x$].offset := best_offset;
  if (best_F ≠ ∅) {
    MAP[$x$].conflict := true;
  } else {
    MAP[$x$].conflict := false;
  }
  // update conflict of each $y \in$ best_G
  foreach $y$ in best_G {
    MAP[$y$].conflict := true;
  }
}
```

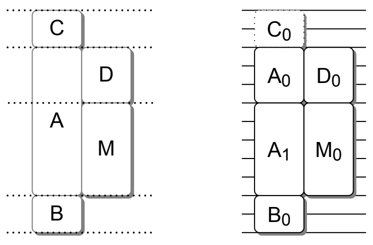
Figure 7: Algorithm for placing a function x.

offset with the minimum cost is chosen as the location of D in Figure 6(c). $Cost(x, p)$ is divided into two different parts: the cost of loading code segments into the memory, $Cost_{load}(x, p)$, and the cost of run-time checks performed by the overlay manager, $Cost_{check}(x, p)$, when x is placed at p.

$$Cost_{load}(x, p) = \sum_{y \in F(p)} S_{xy}(p) \times TRG(x, y)$$

$$Cost_{check}(x, p) = \begin{cases} 0 & \text{if } F(p) = \emptyset \\ WCG(x) + \sum_{y \in G(p)} WCG(y) & \text{otherwise} \end{cases}$$

where $TRG(x, y)$ is the weight of the edge between two nodes x and y in the original TRG and $WCG(x)$ is the number of invocations of function x. In addition, $F(p)$ is the set of all functions that conflict with x at p, $G(p) \subseteq F(p)$ is the set of all functions that conflict only with x at p, and $S_{xy}(p)$ is the size of overlap between x and each $y \in F(p)$ at p. Since $y \in F(p)$ and $y \notin G(p)$ implies that y requires run-time checks regardless of x, the check cost of y will not be introduced by placing x at p. On the other hand, the check cost for a function $y \in G(p)$ will be introduced by placing x at p because y does not require run-time checks unless x is



(a) (b)

Figure 8: The code slices of the overlay structure shown in Figure 6(d). The dotted line in (a) shows the start address or end address of a code segment in the overlay structure. Some code segments are divided into more than one code slice in (b).

placed at offset p.

Then, $Cost(x, p)$ is given by,

$$Cost(x, p) = \alpha \cdot Cost_{load}(x, p) + \beta \cdot Cost_{check}(x, p)$$

where weights α and β are determined empirically.

Using the cost function, the algorithm places D at offset 2 in Figure 6(c). Similarly, when M is being placed in Figure 6(c), if we place M at offset 5 in Figure 6(d), $F(5)$ and $G(5)$ at offset 5 are $\{A\}$ and $\emptyset$, respectively. Note that A is not in $G(5)$ because A already conflicts with D in Figure 6(c). On the other hand, $F(7)$ and $G(7)$ at offset 7 are $\{A, B\}$ and $\{B\}$, respectively. Here, B is in $G(7)$ because B does not conflict with any other function in Figure 6(c) but conflicts only with M when M is placed at offset 7. $Cost_{load}(M, 5)$ and $Cost_{load}(M, 7)$ are the same because the weights of two edges (M, A) and (M, B) are the same in Figure 5(b). M is placed at offset 5 because $Cost_{check}(M, 5)$ is less than $Cost_{check}(M, 7)$.

Figure 7 shows the details of our placement algorithm. The function `place` is invoked to find out an offset with the minimum cost for each function. After placing all functions, the overlay structure is kept in `MAP`. `MAP` maps a function x to a pair of an offset `offset` and a flag `conflict`. The `conflict` indicates if there exists a conflicting function with x in the current overlay structure. The function call `overlap_size(offset, x, y)` calculates the size of overlapping memory regions between x and y when x is placed at `offset`.

The outermost loop of the placement algorithm iterates at most $\lfloor S_{mem}/S_{inc} \rfloor$ times. The innermost loop iterates the number of functions that have been placed. Thus, to generate an overlay structure for a given program with N functions, our placement algorithm takes $O(\lfloor S_{mem}/S_{inc} \rfloor \cdot N^2)$ time.

5.2 Code Slicing

To reduce the redundant memory copying overhead that is discussed in Section 3, each code segment (i.e., a function in our approach) is further divided into *code slices*. The code slice is the unit of loading in our overlaying technique. A code slice is a maximal code fragment in a code segment that entirely overlaps with another single code slice or does not overlap at all in the overlay structure. The start and end addresses of each code segment in the overlay structure divide other code segments into code slices. The start and end addresses of each code segment in the overlay structure in Figure 6(d) are marked with a dotted line in Figure 8(a). Thus, A is divided into two code slices A_0 and A_1 in Figure 8(b). When A is called, the overlay manager loads A_0 and A_1 individually.

```
void bar();

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

(a)

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

(b)

Figure 9: Identifying functions that may be invoked through function pointers. (a) is a sample source code that retrieves the absolute address of `bar`. (b) is an assembly code of (a).

6. CODE GENERATION

In this section, we present the implementation of our overlay manager and the way to insert glue code into the program at link time. We insert glue code for each call site as well as return site to check the residence of the target code segment in the memory.

If a function is invoked via function pointers, there exists the associated relocation information that refers to the absolute address of the function in the input object files to the linker. In other words, the absolute address of the function must be stored somewhere in the input object files. Figure 9(a) is a sample code that retrieves the absolute address of `bar` and its assembly code is shown in Figure 9(b). Symbol `L0` is the memory location in which the value of function symbol `bar` is stored. After the linker resolves all symbol names, the location will store the absolute address of `bar`. To let the linker modify the value of the memory location at `L0`, the input object file contains relocation information. By analyzing the relocation information of all user object files as well as library object files, we can identify all functions that may be invoked through function pointers. We call a function whose absolute address is exposed in the binary an *exposed function*. An exposed function *may* be invoked through function pointers.

The call/return relationship between each function is represented in the static call graph (SCG). We assume that all exposed functions can be invoked at each call site that calls a function through a function pointer. We assign a unique identifier (ID) to each function.

6.1 Overlay Manager

The unit of loading in our overlay manager is a code slice as mentioned in Section 5.2. Our overlay manager consists of three different routines:

- `Load&Branch(ID)`. This function ensures that all code slices of the target code segment specified by `ID` are loaded in the memory and forwards the control to the target code segment.
- `Search&Load&Branch(address)`. This function identifies a target code segment that is associated to `address` for indirect function calls (i.e., calls to a function through a function pointer) by searching the segment table. After identifying the target code segment, the function calls `Load&Branch`.
- `Load&Return(address)`. This function determines if `address` contains the caller's ID and the offset of the original return address in the caller's code by checking the least significant bit of `address`. If not, control is forwarded to `address`. Otherwise, the target code segment may not reside in the memory. It identifies the target code segment using the caller's ID and identifies the return address using the offset. Then it calls `Load&Branch`.

6.2 Glue Code Insertion

We insert glue code for each function call depending on the relationship between the caller and the callee. There are three different cases in the glue code insertion process:

- When the callee is invoked via a function pointer, the callee cannot be identified statically. To ensure that the callee resides in the memory, the glue code invokes `Search&Load&Branch` with the function pointer.
- Otherwise, when the callee is not overlaid (i.e., no code segment overlaps with the callee), the function call is left as-is.
- Otherwise, the callee is overlaid. In this case, the glue code invokes `Load&Branch` with the address of the callee.

In addition, when a function returns, the glue code execution is required to check if the target code segment resides in the memory. The following two different cases are considered when a function returns depending on the returning function type:

- When any of the returning function's descendants in the static call graph do not overlap with the returning function and any of the returning function's predecessors, the return sequence of the returning function is left as-is. In this case, it is guaranteed that the caller (i.e., one of the returning function's predecessors in the static call graph) resides in the memory.
- Otherwise, the caller may have been overwritten by one of the returning function's descendants. In this case, the glue code invokes `Load&Return` with the return address.

However, if the caller is overlaid with other functions, the return address is mapped to multiple functions that can be overlaid in the same memory region as that of the caller. To avoid this situation, the return address is modified depending on the relationship between the caller and the callee.

IBM BladeCenter QS22	
Processor	Dual 3.2GHz Cell BE, 8 SPEs each 512KB L2 cache 256KB local store in each SPE
Main Memory	16GB DDR2 DRAM
OS	Fedora Linux 9
Compiler	IBM Cell SDK 3.1 IBM XL Cell ppuxlc, -O5 IBM XL Cell spuxlc, -O5

Table 1: System Configurations.

- When the caller does not overlap with any of the callee's descendants in the static call graph, the return address is not modified.
- Otherwise, the caller replaces the return address with its ID and the offset of the original return address in its code when the call occurs. The least significant bit (LSB) of the return address is set to 1 to represent that the return address has been modified. `Load&Return` handles the modified return address by checking the LSB.

For example, consider the overlay structure in Figure 8(b) of the program in Figure 1. When M calls A, M (the caller) replaces the original return address with its ID and the offset of the original return address in its code because M and A are overlaid in the same memory region. The glue code inserted in the return sequence of A handles the modified return address and ensures the caller of A resides in memory before A returns to it.

7. EXPERIMENTAL RESULTS

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

7.1 Methodology

We evaluate the proposed overlaying technique on the Cell BE processor. Table 1 summarizes the system configuration of the target machine and the parameters of the compiler used in our experiments. The Cell BE processor consists of a PowerPC processor element (PPE) and eight synergistic processor elements (SPEs) connected by a high bandwidth bus in a single chip. Each SPE has 256KB local store that is in charge of data accesses as well as instruction fetches. The main memory cannot be directly accessed by an SPE. The code or data in the main memory must be copied into the local store before the SPE accesses it.

Benchmark applications. Table 2 summarizes our benchmark applications. We use a set of 11 parallel benchmark applications to evaluate our approach. The set includes four applications from NAS [24], two from SPEC OMP [28], and five from SPLASH2 [35]. The input data set for each application is also shown in Table 2. The benchmark applications are ported to the Cell BE processor using COMIC [21, 27]. COMIC is an open-source coherent shared memory interface for the Cell BE processor. COMIC provides an illusion of a globally shared memory to the programmer. The shared data is divided into a number of pages with a fixed size. Some amount of memory space in each SPE's local

	Source	Input	SPE code		Overlay by the user			Software cache		Generation time			Iterations
			# of functions	Code size [KB]	# of segments	# of sections	Code size [KB]	Buffer size [KB]	Page size [KB]	SDK [sec]	C&L [sec]	TRG [sec]	# of iterations
CG	NAS	class A	19	53	3	1	41	192	8	14.23	0.19	0.29	2.2
FT	NAS	class A	25	32	3	1	24	152	8	7.82	0.11	0.13	2.0
MG	NAS	class A	20	68	10	1	19	176	8	25.77	0.24	0.30	2.7
SP	NAS	class A	43	142	10	1	29	192	4	40.01	0.49	0.53	2.7
ammp	SPEC	reference	22	84	5	1	34	128	4	25.57	0.29	0.29	2.0
swim	SPEC	reference	18	46	6	1	17	160	8	14.68	0.15	0.16	2.0
cholesky	SPLASH2	bcsstk29.psa	79	207	25	5	118	72	4	60.22	0.80	1.35	2.7
ocean	SPLASH2	-n1026	52	452	30	4	105	64	8	172.36	1.17	2.34	2.2
raytrace	SPLASH2	balls4.env	75	193	4	2	146	72	8	53.36	0.74	0.92	2.9
volrend	SPLASH2	head	52	86	6	2	59	144	2	30.18	0.31	0.44	2.0
water	SPLASH2	8000	31	123	7	1	78	120	8	41.49	0.43	0.61	2.2

Table 2: Characteristics of Benchmark Applications.

store, called page buffer, is dedicated to cache these pages of shared data. Table 2 shows the size of the page buffer and the size of page for each benchmark application. Sensitivity of each application to the page size depends on its access pattern. Thus, we select the page size (from 1KB to 16KB) that gives the best performance to each application.

We exclude benchmark applications whose code size is small enough to execute in a Cell BE processor without code overlays. Each of our benchmark applications has been overlaid using an overlay structure specified by the programmer who ports it to the Cell BE processor using COMIC. Since the size of the page buffer is crucial to performance, to obtain enough local store space for software caching in COMIC, the code size needs to be reduced as much as possible using an overlay structure. Our base case for comparison is the user-specified overlay structure.

The linker. To evaluate the proposed overlay technique, we have built a linker for SPEs that implements our approach described in Section 5 and Section 6. It takes the SPE code object files and SPE library object files in ELF format as its inputs. The linker initially generates an overlay structure as described in Section 5. Note that the size of each function may be slightly increased after the glue code insertion step. This implies that the initial overlay structure is not correct any more. For this reason, the linker measures the size of each function when the glue code is inserted. If there exists at least one function whose size is increased, the linker generates a new overlay structure with the function size measured in the previous overlay structure. It repeats this process until there is no function whose size is increased in the generated overlay structure. This iteration always halts because the glue code size and the number of functions are fixed, and we insert the glue code only in function call sites and return sequences of functions. The last column in Table 2 shows the average number of the iterations for all code memory sizes used for each application. The number of iterations is typically 2 and at most 4.

We use 128 for S_{inc} because peak performance is achieved for DMA transfers when both the source and target addresses are 128-byte aligned. The DMA engine in Cell BE shows pretty good performance, e.g., 12.8 GB/s for transferring 16KB memory block. For this reason, we have evaluated with the same value (1) for α and β. For those systems where memory copying from external storage is much slower than memory access, one can use bigger value for α than β.

To compare our technique to the previous technique, the linker also supports the code overlaying technique proposed by Cytron and Loewner [7]. The algorithm constructs the call graph for a given program and collapses each strongly connected component (SCC) into a single node to convert the call graph to a directed acyclic graph (DAG). The DAG is converted to a tree by moving nodes with two or more incoming edges to their immediate dominator. Finally, the algorithm generates the linker script that describes the overlay structure by traversing the tree. The algorithm prevents two different code segments from being mapped to the same memory region when they can be reside in the same call path. Thus, it cannot generate an overlay structure for an arbitrary memory size if the memory size is less than the maximum total size of all code segments in the same call path.

Cell SDK. The compiler in Cell SDK [17] can automatically generate an overlaid executable for a given program by using a call graph partitioning algorithm [16]. Each edge in the call graph is weighted by an estimated call frequency. The available code memory space in the local store is the remaining memory space excluding the space for data and stack. The algorithm merges two nodes (i.e., functions) of an edge in the call graph if their combined size is less than the size of the available code memory space. It repeats this process until no nodes can be merged. Finally, each node in the final call graph forms a code segment that consists of one or more functions. The generated overlay structure is a single overlay section in which all the code segments are overlaid.

7.2 Results

Figure 10 shows our evaluation result for each benchmark application. The execution time of each application is normalized to that of the overlaid executable generated with the user-specified overlay structure (the bar labeled USER). TRG shows the execution time of our approach based on the temporal-ordering information.

The page buffer size (i.e., the software cache size) used in USER and TRG is all the same. The page buffer size for each application is shown in Table 2. The size of code memory space in TRG varies from 20% to 100% of the code memory size in USER. We assume that the page buffer size for each application in USER is optimally selected by the programmer according to the user-specified overlay structure.

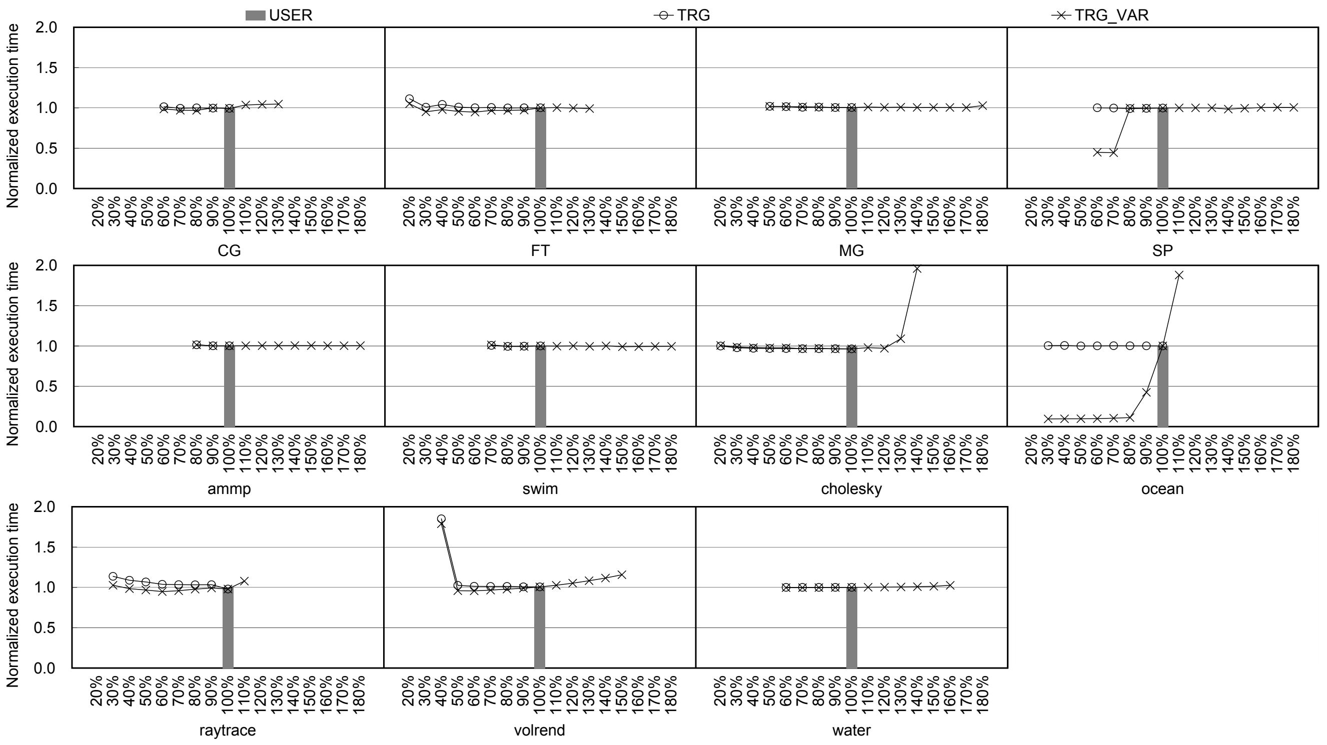


Figure 10: Normalized execution time with 8 SPEs for varying code memory space in the local store.

	Normalized code size to USER			Normalized execution time to USER						
	C&L, C&L_VAR	WCG, WCG_VAR	TRG, TRG_VAR	SDK	C&L	WCG	TRG	C&L_VAR	WCG_VAR	TRG_VAR
CG	95%	60%	60%	1.48	1.00	1.01	1.01	1.00	0.99	0.99
FT	62%	30%	30%	1.32	0.99	1.07	1.01	0.93	1.01	0.95
MG	82%	50%	50%	1.24	1.00	1.03	1.02	1.01	1.03	1.02
SP	69%	60%	60%	1.02	1.00	1.00	1.00	0.45	0.45	0.45
ammp	88%	80%	80%	3.04	1.00	1.60	1.01	1.00	1.01	1.01
swim	87%	70%	70%	1.25	1.00	1.08	1.01	1.00	1.07	1.01
cholesky	99%	20%	20%	1.13	1.04	1.01	1.00	1.04	1.02	1.01
ocean	105%	30%	30%	2.57	1.18	1.01	1.00	1.18	0.10	0.10
raytrace	93%	40%	40%	8.04	55.64	1.16	1.09	55.70	1.06	0.98
volrend	88%	50%	50%	3.89	3.13	1.13	1.03	3.12	1.07	0.96
water	97%	60%	60%	45.15	79.85	5.82	1.00	79.82	5.81	1.00
geomean	86.9%	46.5%	46.5%	2.60	2.42	1.28	1.02	2.24	0.90	0.75

Table 3: Comparison with the traditional techniques.

The saved code memory space due to efficient code overlays can be moved to the page buffer for software caching. On the other hand, we can generate an overlay structure that consumes more code memory space than that in USER by reducing the size of page buffer. TRG_VAR show such case. The entire local store space other than the code, the global data, and the stack is used for the page buffer in TRG_VAR. The size of code memory space in TRG_VAR varies from 20% to 180% of the code memory size in USER. With 180% of the code memory size in USER, either the entire non-overlaid SPE code fits in the code memory space or the page buffer contains just 2 pages for caching that is the minimally required page buffer size by COMIC.

Table 3 compares our approaches to the automatic overlay generation techniques. Code memory sizes and execution times in Table 3 are normalized to those of USER, respectively. For each benchmark application, the code memory size of TRG and TRG_VAR in Table 3 is the smallest code memory size at which TRG_VAR shows performance better than or comparable to USER. SDK shows the execution time of the overlaid executable that is automatically generated by the Cell BE compiler. Note that the code memory size of SDK is the same with that of USER. C&L and C&L_VAR show the code memory size and execution time of the overlaid executable generated by the technique proposed by Cytron and Loewner [7]. The code memory sizes of SDK and C&L do not vary because they cannot generate an overlay structure for an arbitrary memory size as discussed in Section 7.1. To show the effectiveness of the temporal-ordering information on overlay generation, we evaluate our overlay generation algorithm using the WCG instead of the TRG. WCG and WCG_VAR show the execution times of this case at the same code memory size with TRG and TRG_VAR. The page buffer size (i.e., the software cache size) used in C&L, SDK, WCG, and TRG is all the same with that used in USER shown in

Table 2. On the other hand, the page buffer size used in C&L_VAR and WCG_VAR is adjusted in the same manner as that used in TRG_VAR is adjusted.

USER outperforms SDK and C&L for all benchmark applications. USER is on average 2.6 times (up to 45.1 times for water) faster than SDK, and 2.4 times (up to 79.8 times for water) faster than C&L. At the same code memory size in USER, TRG and TRG_VAR also outperform SDK and C&L. When we choose the code memory size that achieves the best performance for each application, TRG_VAR is on average 3.5 times (up to 45.1 times for water) faster than SDK, and 3.2 times (up to 79.8 times for water) faster than C&L_VAR. This can be explained with the following reasons: First, each code segment in the overlay structures of SDK and C&L consists of one or more functions. Thus, when a function is being loaded, all functions in the same code segment are loaded in the memory as discussed in Section 3. Second, all code segments in SDK are placed in a single overlay section, the entire code memory space. Thus, space utilization by the overlays is inefficient. Finally, the call graph does not contain the temporal relationship between functions [11, 20]. Even though these approaches enable function prefetching into the code memory, our experimental results show that the unnecessary memory copying overhead is much bigger than the performance gain obtained by prefetching.

For all applications, TRG at the same code memory size in USER, i.e., 100%, achieves the same as or slightly better performance than that of USER. TRG is on average 0.4% faster than USER at 100%. With a much smaller code memory size, TRG achieves comparable performance to USER for all applications but FT, raytrace, and volrend. Note that the page buffer size in TRG is the same as that of USER.

The effects of temporal-ordering information. In Section 4, we discuss that the TRG will yield more efficient overlay structure than the WCG because the TRG represents the temporal-ordering information better than the WCG does. The effectiveness of the temporal-ordering information is manifest when we compare TRG with WCG in Table 3. TRG and TRG_VAR outperform WCG and WCG_VAR at the same code memory size, respectively. On average, TRG is 1.3 times (up to 5.8 times for water) faster than WCG, and TRG_VAR is 1.2 times (up to 5.8 times for water) faster than WCG_VAR.

The effects on software caching. A large benefit of our approach is that it can automatically generate an efficient overlay structure for an arbitrary code memory size. TRG_VAR at 30% - 80% for ocean and at 60% - 70% for SP achieves much better performance than USER because the page buffer gets more space. On the other hand, a large code memory space does not guarantee better performance because it consumes the space reserved for software caching. TRG_VAR at 130% - 140% for cholesky, at 110% for ocean and raytrace, and at 110% - 150% for volrend show this situation. With our approach, the programmer can easily try different combinations of code memory size and page buffer size for an application and choose the best combination for the application.

Compile time. Table 2 shows the average compile time spent to generate an overlaid executable for an application in SDK, C&L, and TRG. The results show that SDK takes much more time than C&L and TRG. SDK includes not only the time spent to generate an overlaid executable but also the time spent for inter-procedural optimizations. For all applications but cholesky and ocean, our approach does not take more time than a second. This is small enough to ignore compared to the inter-procedural optimization time.

In summary, our approach (TRG_VAR) is, on average (geometric mean), 3.51, 3.24 and 1.33 times faster than SDK, C&L_VAR and USER, respectively with more than 50% code memory savings.

8. CONCLUSIONS

We introduce an automatic overlay generation algorithm and the way to insert glue code for multicores with explicitly managed memory hierarchies. Our overlay generation algorithm generates an efficient overlay structure for an arbitrary memory size as long as the memory size is larger than the biggest function in the program. We propose a cost model that considers the overhead due to memory copying and runtime checks by the glue code. Our cost model exploits the temporal-ordering information between functions, and the information is obtained by profiling. The experimental results on the Cell BE processor indicate that our approach is much more efficient than the overlay technique used by the Cell compiler. Moreover, with much smaller code memory, our approach achieves comparable performance to the overlay structure generated by the programmer who understands the application well. The developer can determine the code memory size as small as possible at the cost of tolerable performance degradation with our approach.

9. REFERENCES

[1] F. Angiolini, F. Menichelli, A. Ferrero, L. Benini, and M. Olivieri. A post-compiler approach to scratchpad mapping of code. In *CASES '04: Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 259–267, New York, NY, USA, 2004. ACM.

[2] ARM Ltd. `http://www.arm.com`, 2009.

[3] 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 *CODES '02: Proceedings of the tenth international symposium on Hardware/software codesign*, pages 73–78, New York, NY, USA, 2002. ACM.

[4] L. M. Censier and P. Feautrier. A new solution to coherence problems in multicache systems. *IEEE Transactions on Computers*, 27(12):1112–1118, 1978.

[5] Y.-C. Chen and A. V. Veidenbaum. A software coherence scheme with the assistance of directories. In *ICS '91: Proceedings of the 5th international conference on Supercomputing*, pages 284–294, New York, NY, USA, 1991. ACM.

[6] J. d. Cuvillo, W. Zhu, Z. Hu, and G. R. Gao. Tiny threads: A thread virtual machine for the cyclops64 cellular architecture. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 14*, page 265.2, Washington, DC, USA, 2005. IEEE Computer Society.

[7] R. Cytron and P. G. Loewner. An automatic overlay generator. *IBM Journal of Research and Development*, 30(6):603–608, 1986.

[8] B. Egger, C. Kim, C. Jang, Y. Nam, J. Lee, and S. L. Min. A dynamic code placement technique for scratchpad

memory using postpass optimization. In *CASES '06: Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 223–233, New York, NY, USA, 2006. ACM.

[9] B. Egger, J. Lee, and H. Shin. Scratchpad memory management for portable systems with a memory management unit. In *EMSOFT '06: Proceedings of the 6th ACM & IEEE International conference on Embedded software*, pages 321–330, New York, NY, USA, 2006. ACM.

[10] B. Egger, J. Lee, and H. Shin. Scratchpad memory management in a multitasking environment. In *EMSOFT '08: Proceedings of the 8th ACM international conference on Embedded software*, pages 265–274, New York, NY, USA, 2008. ACM.

[11] N. Gloy, T. Blackwell, M. D. Smith, and B. Calder. Procedure placement using temporal ordering information. In *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 303–313, Washington, DC, USA, 1997. IEEE Computer Society.

[12] C. Guillon, F. Rastello, T. Bidault, and F. Bouchez. Procedure placement using temporal-ordering information: dealing with code size expansion. In *CASES '04: Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 268–279, New York, NY, USA, 2004. ACM.

[13] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Reactive nuca: near-optimal block placement and replication in distributed caches. In *ISCA '09: Proceedings of the 36th annual international symposium on Computer architecture*, pages 184–195, New York, NY, USA, 2009. ACM.

[14] A. H. Hashemi, D. R. Kaeli, and B. Calder. Efficient procedure mapping using cache line coloring. In *PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 171–182, New York, NY, USA, 1997. ACM.

[15] H. He, S. K. Debray, and G. R. Andrews. The revenge of the overlay: automatic compaction of os kernel code via on-demand code loading. In *EMSOFT '07: Proceedings of the 7th ACM & IEEE international conference on Embedded software*, pages 75–83, New York, NY, USA, 2007. ACM.

[16] IBM. An introduction to compiling for the cell broadband engine architecture. 2006.

[17] IBM. *Software Development Kit for Multicore Acceleration version 3.1, Programmer's Guide*. IBM, 2008.

[18] IBM, Sony, and Toshiba. *Cell Broadband Engine Architecture*. IBM, October 2007. `http://www.ibm.com/developerworks/power/cell/`.

[19] Intel Corporation. Single-chip cloud computer. `http://techresearch.intel.com/articles/Tera-Scale/1826.htm`, 2009.

[20] J. Kalamatianos and D. Kaeli. Temporal-based procedure reordering for improved instruction cache performance. In *HPCA '98: Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, page 244, Washington, DC, USA, 1998. IEEE Computer Society.

[21] J. Lee, S. Seo, C. Kim, J. Kim, P. Chun, Z. Sura, J. Kim, and S. Han. COMIC: a coherent shared memory interface for cell be. In *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 303–314, New York, NY, USA, 2008. ACM.

[22] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The directory-based cache coherence protocol for the dash multiprocessor. In *ISCA '90: Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 148–159, New York, NY, USA, 1990. ACM.

[23] J. R. Levine. *Linkers and Loaders*. Morgan Kaufmann, 2000.

[24] NASA Advanced Supercomputing Division. NAS parallel benchmarks. `http://www.nas.nasa.gov/Resources/Software/npb.html`.

[25] A. Pabalkar, A. Shrivastava, A. Kannan, and J. Lee. Sdrm: simultaneous determination of regions and function-to-region mapping for scratchpad memories. In *HiPC'08: Proceedings of the 15th international conference on High performance computing*, pages 569–582, Berlin, Heidelberg, 2008. Springer-Verlag.

[26] K. Pettis and R. C. Hansen. Profile guided code positioning. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 16–27, New York, NY, USA, 1990. ACM.

[27] S. Seo, J. Lee, and Z. Sura. Design and implementation of software-managed caches for multicores with local memory. In *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, pages 55 –66, 2009.

[28] Standard Performance Evaluation Corporation. SPEC 2000. `http://www.spec.org/benchmarks.html`.

[29] S. Steinke, L. Wehmeyer, B. Lee, and P. Marwedel. Assigning program and data objects to scratchpad for energy reduction. In *DATE '02: Proceedings of the conference on Design, automation and test in Europe*, page 409, Washington, DC, USA, 2002. IEEE Computer Society.

[30] P. Stenström. A survey of cache coherence schemes for multiprocessors. *Computer*, 23(6):12–24, 1990.

[31] Texas Instruments Inc. `http://www.ti.com`, 2009.

[32] S. R. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, A. Singh, T. Jacob, S. Jain, V. Erraguntla, C. Roberts, Y. Hoskote, N. Borkar, and S. Borkar. An 80-tile sub-100-w teraflops processor in 65-nm cmos. *IEEE Journal of Solid-State Circuits*, 43(1):29 –41, jan. 2008.

[33] M. Verma, L. Wehmeyer, and P. Marwedel. Cache-aware scratchpad allocation algorithm. In *DATE '04: Proceedings of the conference on Design, automation and test in Europe*, page 21264, Washington, DC, USA, 2004. IEEE Computer Society.

[34] M. Verma, L. Wehmeyer, and P. Marwedel. Dynamic overlay of scratchpad memory for energy minimization. In *CODES+ISSS '04: Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 104–109, New York, NY, USA, 2004. ACM.

[35] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: characterization and methodological considerations. In *ISCA '95: Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, New York, NY, USA, 1995. ACM.