Code and Data Structure Partitioning for Parallel and Flexible MPSoC Specification Using Designer-Controlled Recoding

Pramod Chandraiah, *Student Member, IEEE*, and Rainer Dömer, *Member, IEEE*

***Abstract*—MultiProcessor Systems-on-Chip (MPSoCs) are increasingly being used to build efficient and cost-effective embedded systems that meet the necessary real-time requirements. However, programming heterogeneous MPSoCs is highly challenging. The existing automatic parallelizing techniques, although effective on homogeneous shared-memory architectures, are insufficient for MPSoCs, which are typically characterized by heterogeneous processing elements and memory architectures. The lack of effective automatic techniques for recoding and parallelization requires designers to manually partition the code and the data structures in the reference application to generate a parallel and flexible specification model. Such manual algorithm partitioning by the designer is time consuming and error prone. In this paper, we motivate the need for automation in system specification and present a novel designer-controlled approach to recode applications written in a C-based System-Level Description Language. We present six automated source code transformations that, under the control of the designer, automatically partition and reorganize code and data structures to create a parallel and flexible abstract specification model that can be mapped onto a heterogeneous MPSoC using a top-down system-level design flow. Our experimental results show significant productivity gains and quality improvements in the end design.**

***Index Terms*—Code and data partitioning, design automation, multi-processor systems-on-chip, recoding, source code transformation, system level design.**

I. Introduction

THE DESIGN of embedded systems imposes a mutually conflicting set of constraints on the design, such as real-time performance, low power, low cost, and short time-to-market. By means of task-level parallelism and by adapting the system architecture to the application needs, MultiProcessor Systems-on-Chip (MPSoCs) have emerged as an effective way to realize complex embedded systems. The need to generate an effective optimized implementation and the need to speed up the design process have driven past research in the direction of configurable processors [1], hardware/software codesign, flexible platforms [2], on-chip communication networks [3], new programming models [4], and so on. Despite the decades of research, a push-button tool/compiler to map a sequential application onto a heterogeneous MPSoC architecture is still not realistic. One major obstacle can be attributed to the heterogeneous nature of many MPSoCs, which are typically characterized by specialized processors, irregular memory structure (shared and local memories), and a custom interconnecting network.

An effective parallel programming of these MPSoCs is the key in reaping the potential benefits of MPSoC architectures. An abstract system-level model, typically known by names such as specification model or Transaction Level Model (TLM) or MPSoC specification, forms the basis for this programming. In particular, this programming involves splitting typically sequential application codes into multiple parallel partitions, programming each processor, and minimizing the communication overhead between them. One critical research area, which has not received much attention, is the development of this initial specification model, necessary for programming the underlying MPSoC platform.

For instance, we used the top-down refinement-based automatic design flow [5], shown in Fig. 1, to design a Moving Pictures Experts Group Audio Layer 3 (MP3) audio decoder. In this flow, the design models are shown in ellipses, and the refinement tasks in between models are shown as rectangles. The design process starts with an abstract parallel specification model that is then refined to create models at lower abstraction levels, including an architecture model, TLM, and Bus-Functional Model. After a series of refinement steps, an actual implementation is finally derived. Each of the refinement steps in the design flow is automated to the extent that model generation is fully automatic, and the designer has to only make the design decisions, such as component allocation, mapping, and scheduling. Due to automatic refinement, the final implementation of our MP3 decoder was derived in less than one week. However, the very first design step, i.e., recoding the sequential C code into an MPSoC specification, took more than 90% of the overall design time [6]. In this paper, we address this bottleneck of creating a suitable MPSoC specification from readily available C code so that the overall design time is reduced.

II. MPSoC Specification

The design of MPSoCs starts with an initial specification model that serves as the golden model for the generation and validation of all subsequent models. The language in which

Manuscript received June 8, 2007; revised October 24, 2007. This work was supported in part by Nicholas Endowment through the Henry T. Nicholas III Research Fellowship. This paper was recommended by Associate Editor L. Benini.

The authors are with the University of California, Irvine, CA 92697 USA (e-mail: pramodc@uci.edu; doemer@uci.edu).

Digital Object Identifier 10.1109/TCAD.2008.923244

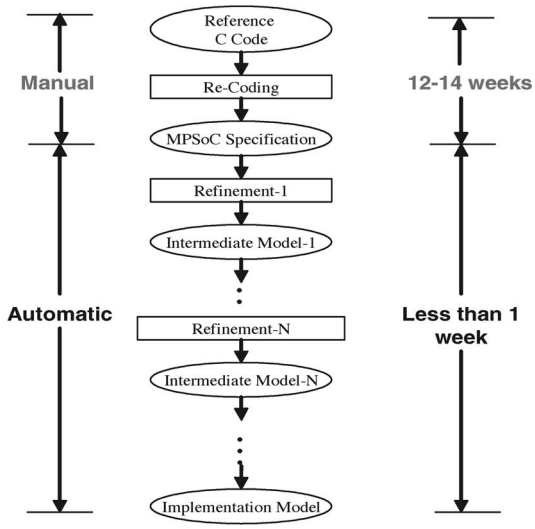

Fig. 1. Motivation: design time of the MP3 decoder in a refinement-based design flow.

the model is implemented depends on the design flow and the tools, but System-Level Description Languages (SLDLs), such as SystemC [7], SpecC [5], and System Verilog [8], are often used as they can capture systems containing both hardware and software components at different levels of abstraction. Concurrency and flexibility are two important characteristics of this model. Concurrency directly determines the efficient utilization of the underlying parallel architecture, and flexibility directly impacts the number of design alternatives that can be explored. In general, flexibility increases with explicit separation of computation and the communication in the model. More specifically, flexibility directly depends on the number of distinct code and distinct data partitions that communicate through abstract message-passing channels. During the later stages of system synthesis, the model flexibility increases the number of different code, data, and channel mappings onto processors, memories, and busses, respectively. Often, code and data partitions can be moved by the designer across processors and memories to balance computation and optimize communication.

A. Recoding C Code Into MPSoC Specification

Although it is possible, the system specification model is rarely written from scratch. More than often, reference models of applications are reused and recoded into System-on-Chip (SoC) specifications. Reference implementations in C are popular choices due to their availability and familiarity, and since they aptly fit when recoded into C-based SLDLs. However, such reference applications are usually designed to run on a regular PC environment with a single processor and are not directly suitable as an MPSoC specification. The monolithic application code and data need to be split into partitions with task-level parallelism exposed. Generating the MPSoC specification from a sequential code involves partitioning the code and data structures, and isolating the partitions so that they can later be mapped to different architectural components. The source code transformations to create a flexible and parallel model can be broadly classified into the following three categories:

1) code partitioning;
2) data structure partitioning and data relocation;
3) flexibility-adding transformations.

B. Why Source-Level Code and Data Structure Partitioning?

Due to the absence of effective tools to perform automatic partitioning and mapping of sequential C codes onto an MPSoC, it becomes necessary to first create models with explicit features. Explicit models are conducive for automatic design exploration and synthesis as they can be statically analyzed by the later design tools. The explicit features that make the analysis easier include exposing concurrency, explicit code, and data partitions; explicit locality of data variables; and so on. As an example, we emphasize the importance for explicit data structure partitioning.[1] In general, partitioning of composite data structures (structures, vectors) and localization of the partitions to processors reduce the communication between processors. In low-cost MPSoCs, designed for low-power applications, communication is premium and must be reduced to gain performance and power benefits. In such cases, the composite variables in the application need to be partitioned and explicitly localized to each code partition so that they can reside in respective local memories.

III. Designer-Controlled Approach

Traditional parallelizing compilers could be used to create an MPSoC specification for the purpose of exposing concurrence. However, these completely automatic techniques have been ineffective for three reasons. First, they are mainly based on a shared-memory programming model, hence insufficient to handle heterogeneous MPSoC characteristics such as specialized custom processors and nonuniform memory architectures. Second, although effective in parallelizing applications in scientific computing, the completely automatic compilers are often ineffective in handling real-life embedded source codes. Embedded source codes, typically obtained as reference codes from standardizing committees and open-source projects, are often complex with parallel loops potentially spanning hundreds of lines of code, containing function calls, conditional statements, and complex dependencies. Third, more than often, exposing concurrency in embedded applications requires algorithm knowledge and thus cannot be automatically detected by the compilers. Neither can such knowledge be efficiently fed into the compiler.

All these reasons call for a designer-controlled approach to parallelization, where the designer can choose the code to be parallelized and select data structures to be partitioned and localized. Instead of attempting to parallelize and create an

[1]Note that explicit data partitioning is not a necessity in case of shared-memory-based architectures, where there is only one memory. In such cases, communication is implicit through a cache-coherence mechanism, and communication is typically optimized using data locality techniques (e.g., loop transformations [9]).

MPSoC specification in one single step, we achieve this intractable task using designer-controlled automatic steps. Our approach to application partitioning and parallelization is controlled by the designer to the extent that all the critical analysis and the time-consuming code transformations are performed by a set of automatic transformations, but the decision to apply them is under the control of the designer. Our goal is to generate a flexible MPSoC specification model in SLDLs, such as SystemC and SpecC, from a sequential reference C model. To achieve this goal, we propose a set of six code and data partitioning and flexibility-adding transformations on the C-based source code that can be instantly activated by the designer "on a button click."

IV. Related Work

Up to now, the problem of partitioning code and data structures in a reference application to generate a flexible parallel MPSoC specification has not received much attention. The onus of performing this tedious recoding task still relies on the designer. Our approach toward automated recoding partially overlaps with related work on parallelizing compilers that we compare against. We also compare our designer-controlled approach to a software engineering technique known as refactoring.

A. Parallelizing Compilers

There has been extensive research in the parallel computing community to automatically parallelize applications. By means of interprocedural analysis [10], symbolic analysis [11], and loop transformations, techniques [10], [12], [13] to extract coarse-grained parallelism have been proposed for shared-memory multiprocessors. The Intel C compiler [14] for Pentium-3 and Pentium-4 implements the above techniques to extract thread-level parallelism for shared-memory architectures. Despite these advanced analysis capabilities, completely automatic techniques have not been effective even for shared-memory architectures. Application knowledge is necessary to parallelize many real-life applications. This has resulted in OpenMP compilers [15], where the programmer sets OpenMP directives to parallelize the code. However, all the analysis to ensure that a piece of code can be parallelized and the resolution of nasty dependencies must be performed by the programmer.

For distributed computer systems, on the other hand, the programmer manually writes parallel code using a message-passing interface [16]. These techniques have been effective in parallelizing data-extensive scientific applications. However, these approaches have serious limitations and are not effective in analyzing real-life embedded applications where loops often span hundreds of lines of code and contain function calls, variable accesses through pointers, and complex dependencies that cannot be automatically resolved.

In contrast, our designer-controlled approach breaks the overall task of exposing parallelism into a distinct set of transformations that the designer can easily understand and apply using his application knowledge.

B. Code Refactoring

Refactoring [17] is a software engineering technique used in object-oriented programming to change the structure of a program. This paper is different in many aspects. First of all, our transformations are different from popular refactorings [18]. Second, the intent of our transformations and the area of application are different from refactoring. Traditional refactoring is intended toward improving the human readability, understandability, and maintainability of the source code. Our transformations are intended to create a multiprocessor specification model, and the primary goals are exposing concurrency and adding flexibility to explore different design alternatives. Further, refactoring is mainly used in type-safe languages such as Java. Our strict designer-controlled approach is designed for embedded source codes in C-based languages by tightly involving the designer in the recoding process.

C. Previous Work

In previous works, we have proposed a subset of our transformations in a different context and in less detail. In [19], we discussed four of the important transformations to create parallel and flexible models in a short paper. In this paper, we discuss those transformations in more detail and add two new transformations (structure partitioning in Section V-D and variable rescoping in Section V-F) that are necessary to complete the parallelization task. We also provide detailed algorithms for the transformations that are not included in [19]. In the context of data structure partitioning, [19] only covers vector partitioning. In this paper, we present both partitioning of structures and vectors. Further, we have previously discussed variable localization in the context of creating a specification with contained communication [20]. Reference [6] discusses the data structures and tools that constitute our source recoding environment.

In summary, this paper extends the transformations proposed in [19] and [20] and adds two new ones. We also report new experimental results that demonstrate the benefits of an explicitly parallel and flexible design specification toward implementation as an MPSoC.

V. Controlled Transformations

In this section, we present six program transformations that implement code and data partitioning to expose parallelism in loops, rescope variables, and introduce channels to add flexibility to the model. The designer's application knowledge is used between each transformation step to resolve any dependencies that cannot be handled by the automatic analysis.

A. Loop Splitting

Loop splitting is one of the many well-known loop transformations used in the compiler optimization and parallelizing community [9], [21]. Our loop splitting transformation creates different incarnations of the loop with the same loop body, where each split loop iterates over different contiguous subsets

```
1. int a[32], b[16], c, d, x;
2. …
3. //loop
4. for (i=0; i<16; i++) {
5.    x = i *i;          //CAT(x) in this loop is WR
6.    a[i]++;            //CAT(a) in this loop is RW
7.    a[2i] = c+d;       //CAT(c,d) in this loop is R
8.    b[i] = c*d-x; }    //CAT(b) in this loop is W
```

(a)

```
1. int a[32], b[16], c, d, x;
2. …
3. //loop partition 1
4. for (i=0; i<4; i++) {
5.    x = i *i;
6.    a[i]++;
7.    a[2i] = c+d;
8.    b[i] = c*d-x; }
9. //loop partition 2
10. for (i=4; i<8; i++) {
11.   x = i *i;
12.   a[i]++;
13.   a[2i] = c+d;
14.   b[i] = c*d-x; }
15. //loop partition 3
16. for (i=8; i<12; i++) {
17.   x = i *i;
18.   a[i]++;
19.   a[2i] = c+d;
20.   b[i] = c*d-x; }
21. //loop partition 4
22. for (i=12; i<16; i++) {
23.   x = i *i;
24.   a[i]++;
25.   a[2i] = c+d;
26.   b[i] = c*d-x; }
```

(b)

Fig. 2. Code changes resulting from loop splitting. (a) Original loop. (b) Partitioned loops.

of the loop index range. Depending on the trip count and the number of unrolls specified by the designer, the resulting partitions are loops with smaller trip count or just straight line code segments with the induction variable completely replaced by constants. Fig. 2 shows an example loop with trip count of 16 uniformly split into four loops, each with a trip count of four. The Cumulative Access Type (CAT) in Fig. 2(a) is described in Section V-B. Nonuniform splitting involves splitting loops to have splits with unequal index ranges.

Since the designer can specify the parameters of the loop to the transformation, loops that are typically not parallelized by automatic compilers can also be handled. For example, many automatic compilers attempt to parallelize only *for* loops, whose loop boundaries and trip count can be statically determined. However, often in reference code, it is common for a programmer to use *while* structures instead of *for* structures to specify loops. The *while* loops can also be split if the loop parameters are known.

For reasons of load balancing, the designer might also want to perform nonuniform splitting of a loop. Unlike the completely automatic approaches, our designer-controlled approach provides a way to do so. This is specifically important in the context of embedded source codes, where loop iterations exist with different computation load, and in the context of MPSoC design, where processors with different capabilities are given.

B. Cumulative Access-Type Analysis

Splitting a loop only creates sequential loop partitions. To have communication-free parallelism between these loop partitions, the dependents between partitions must be analyzed and resolved. This static analysis of the loops reveals scalar and vector variables that are dependent between iterations of the loop. Our analysis step presents the found dependencies to the designer in an easy understandable way for resolution.

Variables written in one iteration and read in the other are considered dependents. We classify cumulative accesses of the variables within the loop into four CAT categories, i.e., Read (R), Write (W), Write–Read (WR), and Read–Write (RW). Fig. 2 shows an example loop with variables with different CATs. Variables with access-type RW are considered dependents. Variables with WR access are not dependents between iterations as they are written first before being read in the same iteration. Access of scalar dependents (CAT = RW) must be synchronized between the partitions (as discussed in Section V-E) to ensure correct execution semantics of the program after parallelization. The composite variable dependents, such as structures and vectors, must be further analyzed and if possible partitioned to avoid any communication, as discussed in the next section.

C. Partitioning Vector Dependants

If there exist dependents between the loop partitions, then it is not possible to have communication-free parallelism. The scalar dependents must be manually resolved or synchronized using the transformation presented below in Section V-E. The vector dependents can be analyzed to check if they can be partitioned to have communication-free parallelism. In this section, we present analysis and transformations to check if a vector can be partitioned into contiguous sections across different partitions [19].

It is possible to split vector dependents without communication overhead between the loop partitions only if the array references in different loop partitions do not depend on the same array elements. This is a hard problem to solve in the presence of sparse array accesses such as $A[B[i]]$. However, the problem becomes tractable if the array references are affine expressions, which is often the case in practical source codes.

For message-passing parallel machines, Tseng and Gaudiot [22] provide techniques to partition an array across multiple processors. Communication-free partitioning is possible only if the array references differ by a constant, i.e., the array reference is of the form $x + b$, where x is the induction variable and b a constant. Unlike this approach, our transformation conducts a general affine expression $(mx + b)$ analysis of the form $mx + b$, where m and b are integers. Extensive work in the area of affine data access analysis can be found in [23].

Our algorithm to partition the vector consists of two main parts. First, we check if the vector can be partitioned, and second, we generate the code to access the new split vectors.

The input to our checking Algorithm 1 are the vector v, main loop L, and split loops L_p, where $0 \leq p \leq NP$, where NP is the number of loop partitions and their parameters. The algorithm analyzes the array access expressions of the form $mx + b$ to determine if the vector can be partitioned. If $m_1x + b_1$ and $m_2x + b_2$ are two affine references to vector v in a loop L with induction variable x, iterating from S to E (inclusive) in increments of Δx, the condition to be satisfied

for these two references to be different in the entire iteration space is

$$m_1x + b_1 \neq m_2(S + k\Delta x) + b_2 \tag{1}$$

$\forall k$, where $1 \leq k \leq ((E - S + 1)/\Delta x) - 1$.

k is the normalized iteration number, and S and E are the start and end values attained by the index variable x in the original loop. The above inequality, if true, ensures that the same element of the vector is not accessed through two index expressions $m_1x + b_1$ and $m_2x + b_2$.

Algorithm 1 Check if vector can be partitioned

Input: vector v, start S, end E, indexvar x, increment Δx, iterations per partition ipp, loop L

Output: true/false

```
   //Generate list of index expressions of x
2: for every access a to v in L do
     expr = a → getindexexpr(x)
4:   if expr → affine(x) then
       list = list ∪ expr
6:   else //nonaffine expr
       return false
8:   end if
   end for
10: k_max = ((E − S + 1)/Δx) − 1
    for every expression cmp1 in the list do
12:   (m1, b1) = cmp1 → getaffineconsts()
      for every other expression cmp2 in the list do
14:     (m2, b2) = cmp2 → getaffineconsts()
        for (x = S, iter = 0; x ≤ E; x+ = Δx,
             iter + +) do
16:       np = iter/ipp //Partition to which iteration
          belongs to
          k=(m1 * x − m2 * S + b1 − b2)/(m2 * Δx)
18:       rem=(m1*x−m2*S+b1−b2)%(m2*Δx)
          //If k is an integer, check if it overlaps with
          other partitions
          lowRange={l}∀l, (ipp*(np+1)) ≤ l ≤ k_max
20:       highRange={h}∀h, 0≤h≤((ipp*np)−1)
          interfereRange = {lowRange ∪
            highRange}
22:       if k is an integer and k ⊂ interfereRange then
24:         return false
          end if
        end for
26:   end for
    end for
28: return true
```

The pseudocode of Algorithm 1 implements a modified form of (1) as

$$(m_1x - m_2S + b_1 - b_2)/(m_2\Delta x) \neq k \tag{2}$$

$\forall k$, where $1 \leq k \leq ((E - S + 1)/\Delta x) - 1$, and $\forall x$, where $S \leq x \leq E$. The inequality in (2) means that, if the left expression evaluates to an integer, and if this integer corresponds to an iteration number belonging to a different partition, then the vector cannot be partitioned. This is illustrated in Fig. 3, which depicts the iterations assigned to each loop partition for the example loop in Fig. 2. If k falls in the range indicated by the dotted lines, then for this value of x there will be interference and partitioning is not possible. The innermost loop (lines 15–24) in Algorithm 1 checks the above inequality for every value attained by the index variable x.

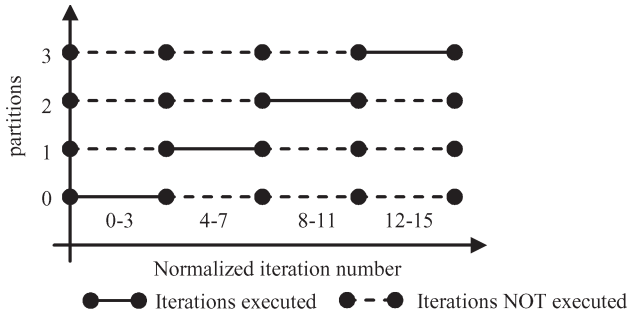

Fig. 3. Loop partitions and their iterations, for example, in Fig. 2.

The vector b in Fig. 2, which is accessed by index expression i, will pass this check. However, vector a accessed using index expressions i and $2i$ will fail the test as some elements are accessed across different partitions (e.g., elements 4 and 6 of vector a are accessed between loop partitions 1 and 2). If the test succeeds, as for vector b, then the main vector can be replaced with the split vectors as described below.

To replace the main vector with the split vector, we need to determine the size of the split vectors, which are calculated by first determining the start and end split points in the main vector. The start (VPS_p) and end (VPE_p) of V are computed for each loop partition L_p with index limits S_p and E_p. Given N different index expressions ($IE(x)$) in the loop body, these limits are the minimum and maximum values of all $IE(x)$, respectively. That is, $VPS_p = \min(IE_n^{\phi})$, and $VPE_p = \max(IE_n^{\psi})$, where

$$IE_n^{\phi} = \min\left(IE_n(S_p), IE_n(E_p)\right) \qquad \forall n \ 1 \leq n \leq N$$

$$IE_n^{\psi} = \max\left(IE_n(S_p), IE_n(E_p)\right) \qquad \forall n \ 1 \leq n \leq N.$$

The pseudocode for computing these limits is shown in Algorithm 2. The inner loop in lines 5–12 determines the minimum and maximum values for index expression and computes the start and end of each vector partition. If the vector portions computed for each loop partition above are nonoverlapping ($VPE_p < VPS_{p+1}$ $\forall p$ $0 \leq p \leq NP - 1$), then for each loop partition, separate vector variables (V_p where $0 \leq p \leq NP-1$) with size $VPE_p - VPS_p + 1$ are created. An index expression $IE_n(x)$ in loop partition p is normalized with $IE_n(x) - VPS_p$ to account for the smaller vector. Now the access to the vector V in each loop partition (L_p) is replaced with access to vector partition V_p with normalized index expression. Fig. 4 shows the loop partitions after splitting the array b. If the split vector is used by other parts of the program, these partition details are remembered, and correct split and merge codes are generated.

```
1. int a[32], b_part1[4], b_part2[4],
   b_part3[4], b_part4[4], c, d, x;
2. …
3. //loop partition 1
4. for (i=0; i<4; i++) {
5.    x = i *i;
6.    a[i]++;
7.    a[2i] = c+d;
8.    b_part1[i] = c*d-x; }
9. //loop partition 2
10. for (i=4; i<8; i++) {
11.   x = i *i;
12.   a[i]++;
13.   a[2i] = c+d;
14.   b_part2 [i-4] = c*d-x; }
15. //loop partition 3
16. for (i=8; i<12; i++) {
17.   x = i *i;
18.   a[i]++;
19.   a[2i] = c+d;
20.   b_part3 [i-8] = c*d-x; }
21. //loop partition 4
22. for (i=12; i<16; i++) {
23.   x = i *i;
24.   a[i]++;
25.   a[2i] = c+d;
26.   b_part4[i-12] = c*d-x; }
```

Fig. 4. Code after partitioning vector b.

Algorithm 2 Compute the start and end index of each vector partition

Input: start S, end E, increment Δx, number of partitions $nparts$ iterations per partition ipp, list of index expressions $list$

Output: Vector partitions Start and End

for $(p = 0; p < nparts; p++)$ **do**
  $s_{part} = S + p * ipp * \Delta x$
  $e_{part} = S + (\Delta x) * (ipp - 1)$
  for $(i = 0; i < list \rightarrow numelements(); i++)$ **do**
    $E = list[i]$
    $(m, b) = E \rightarrow getaffineconsts()$
    $\min = \min(m * s_{part} + b, m * e_{part} + b)$
    $\max = \max(m * s_{part} + b, m * e_{part} + b)$
    $VPS_p = \min(\min, VPS_p)$
    $VPE_p = \max(\max, VPE_p)$
  end for
end for

D. Breaking Composite Structures

In the previous section, we discussed the splitting of arrays. In this section, we discuss the partitioning of another type of composite variable prevalent in C structures. Splitting structures is of little use in specifications meant for single-memory architectures. However, it is necessary for creating an explicit model for a heterogeneous platform with irregular memory architecture.

Breaking a C structure consists of creating two new structures from the subsets of members from the original structure and replacing all the accesses to the original structure with accesses to new structures. Fig. 5 shows code changes in a typical C code segment after splitting the structure $s1$. As shown in lines 1–5 of Fig. 5(b), two new structures (*s1_part1* and *s1_part2*) are created with member subsets a and b, c. For every variable of the original structure type, a pair of new variables of split structure type is created (lines 6–9). New structure initializers are obtained by splitting the original initializer. Every member access expression involving the original structure is replaced with an expression formed from the appropriate variable of the split type (lines 10 and 12). Expressions such as structure assignments (line 12) result in two expressions, one for each split variable. Functions as $f1()$ in line 15, which take structures as arguments, are similarly

```
1.  struct  s1 {
2.    int a;
3.    int b;
4.    int c; } S1;
5.
6.  S1 X;
7.
8.  S1 Y = {1, 2, 3};
9.
10. X.a = 1;
11. int *pint = &Y.a;
12. X = Y;
13. f1(X);
14.
15. void f1(S1 s1) {
16.   s1.a = 1;
17.   s1.b = 2;
18. }
```

(a)

```
1.  struct  s1_part1 {
2.    int a; } S1_PART1;
3.  struct  s1_part2 {
4.    int b;
5.    int c; }S1_PART2;
6.  S1_PART1  X_part1;
7.  S1_PART2  X_part2;
8.  S1_PART1 Y_part1 = {1};
9.  S1_PART2 Y_part2 = {2, 3};
10. X_part1.a = 1;
11. int *pint = &Y_part1.a;
12. X_part1 = Y_part1; X_part2 = Y_part2;
13. f1(X_part1, X_part2);
14.
15. void f1(S1_part1  s1_part1, S1_part2 s1_part2)
16.   s1_part1.a = 1;
17.   s1_part2.b = 2;
18. }
```

(b)

Fig. 5. Splitting structure S1 at b. (a) Before. (b) After.

replaced with two split arguments and then optimized to remove the unused argument.

Pointers to structures are also similarly handled. As shown in Fig. 6, the initial pointer p is now replaced with two pointers *p_part1* and *p_part2*. The member access expressions (lines 4 and 5) are modified to use the proper split pointer. Functions with pointer to structure as argument are also split to create two arguments and later optimized to eliminate unused arguments, as shown in lines 10–12.

This transformation is interprocedural and completely automatic given the structure to be split and the split point in the structure. Our algorithm can split structures even in the presence of pointers to structures and pointer structure members, and even linked lists are automatically handled. Note that our approach is very general and handles most of the code in real-world examples. However, we currently do not handle the following situations.

1) We do not split structures that are returned as value or pointer from a function.
2) Expressions involving pointer arithmetic to structures are not supported. This implies that if the structure is an array element, then our algorithm will not split the structure.
3) Expressions involving pointer arithmetic to members of structures are not portable and are not supported.

Barring these few limitations, our automatic transformation is fast, robust, and effective on many examples.

An advantage of breaking structures is to clear the code from false dependencies. After splitting structure $S1$ in Fig. 5, the function $f1()$ has two parameters of split type (line 15). Similarly, the function $f2()$ in Fig. 6 is also modified to contain two parameters. However, static analysis reveals that $f2()$ depends only on *s1_part1*; thus, part2 of the structure can be omitted from the argument list. In general, after breaking a structure, each function is optimized by deleting unused arguments, thus eliminating false dependencies on unused members. Note that

```
1.  S1 *p;                          1.  S1_PART1  *p_part1;
2.                                  2.  S1_PART2  *p_part2;
3.  p = &X;                         3.  p_part1 = &X_part1; p_part2 = &X_part2;
4.  p →a = 2;                       4.  p_part1→a = 2;
5.  p →b++;                         5.  p_part2→b++;
6.                                  6.
7. if (X.c == 2) {                  7.  if(X_part2.c == 2)  {
8.  f2(&X); }                       8.    f2(&X_part1); }
9. else f2(&Y);                     9.  else f2(&Y_part1);
10. void f2(S1 * q) {               10. void f2(S1_part1 * q_part1) {
11.    q → a = 1;                   11.   q_part1 → a = 1;
12. }                               12. }
        (a)                                 (b)
```

Fig. 6. Handling pointers to structures. (a) Before. (b) After.

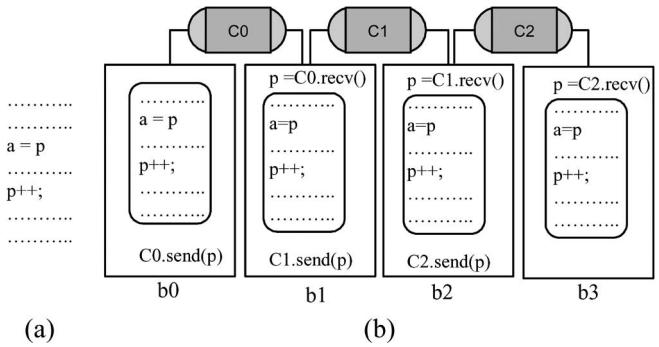

Fig. 7. Synchronizing access to a scalar dependent. (a) Code before. (b) Code in each behavior after synchronizing variable p.

our static analysis recursively works on the function call tree so that indirect dependencies are properly found and supported.

E. Synchronizing Dependent Variables

After loop partitioning, to have complete parallelism, any scalar dependents in a loop have to be resolved by the designer. As one option, the designer can use a transformation that synchronizes access to the scalar dependents through channels. We should note, however, that communication and synchronization resulting from such dependencies can significantly reduce the effective parallelism. Thus, this decision has to be made by the designer. We provide this transformation so that the designer can quickly create a semantically correct model with explicitly synchronized dependents and thereby study the communication by simulation. Explicitly capturing the dependents using channels can also enhance the readability of the program and aid in resolving the dependents. We use double-handshake channels for synchronization, which implement a blocking *send()* and *receive()* communication protocol.[2] Fig. 7 shows an example where the variable p, dependent across the behaviors $b0$, $b1$, $b2$, and $b3$, is synchronized using three message-passing channels. As shown in Fig. 8, synchronizing a variable requires the

[2]Note that, here, in the specification model, these channels are implementation independent. They can be later implemented by actual message passing or through shared memory communication.

Input: number of partitions NP
1. for every dependent *d*
2. create *NP-1* channels ($ch_0, ch_1, ch_2 \ldots ch_{NP-2}$) in place of variable *d*
3. for every behavior b_l (l = 0 to NP-2)
4. Introduce ch_{l-1}.*send( )* to send dependent *d*
5. for every behavior b_l (l=1 to NP-1)
6. Introduce ch_{l-1}.*recv( )* to receive d

Fig. 8. Procedure to synchronize dependents.

creation of $NP - 1$ number of channels, where NP is the number of code partitions. Following this, at the beginning and the end of each loop partition, a *receive()* and a *send()* call, respectively, of the appropriate channel are introduced. This transformation, when applied to each dependent variable, will result in a semantically correct specification.

F. Variable Rescoping

For the reference code running on a PC environment, the scope of a variable (local/global) does not matter much. However, in case of MPSoC specification meant for a heterogeneous platform, the location of the variables is critical. For many design flows, moving a variable to the right program scope in the specification is a way to control to which memory it gets mapped to. This in turn is critical in reducing unnecessary synchronization.

Breaking the composite structures/arrays by itself will not help in avoiding communication overheads. After splitting a structure or array, the variable partitions have to be rescoped to a desired code partition so that it can be mapped to the desired memory in the target platform. Any global variables left will be mapped to a global memory resulting in unnecessary synchronization. Thus, global variables should be relocated to the appropriate program scope.

Rescoping, in general, means migrating a variable from one scope to another scope (higher or lower), obeying the access restrictions, of course. Localizing is a specific case of rescoping that applies to variables that are completely localized to a code partition. Localizing a variable to a partition will give an opportunity to map that variable into the local memory of the processing element, which will likely reduce synchronization in the end implementation. Like localizing, rescoping a variable to a scope higher or making it global might also be desirable depending on the target architecture. Our transformation automatically rescopes the variable to a specified program scope. The procedure to implement localizing of global variables is as follows.

1) Find the lowest common parent behavior accessing the given variable.
2) Move the variable to the scope corresponding to the lowest common parent.
3) Provide access to the variable by recursively inserting ports and function parameters in all the behaviors accessing this variable.

Given the destination scope, the procedure to implement the general rescoping is as follows.

1) Validate the destination scope to obey access restrictions.
2) Move the variable to the destination scope.

3) If rescoping to a scope at a lower level, delete any unused connections; if rescoping to a level above, establish new connections by inserting new ports and function parameters.

Determining accesses to variables is performed across functions and behaviors by tracking parameter or the port. When a variable is accessed through pointer indirection, a simple flow-insensitive and context-insensitive pointer analysis is used determine all the accesses. In the presence of ambiguity, safe assumptions are made to generate the correct code.

G. Summary

In the previous section, we discussed six different transformations to partition the code and data structures to create an MPSoC specification. As mentioned in the beginning of the paper, concurrency and flexibility are the important characteristics of this model. Transformations such as loop splitting, CAT analysis, and vector splitting are primary toward creating code partitions and resolving dependencies between them to expose concurrency. The transformations to break structures and to rescope variables add flexibility into the model by explicitly isolating the code and data partitions. Such clearly isolated code and data partitions result in more design explorations by increasing the number of mapping opportunities later in the design flow.

VI. Interactive Recoding

In this section, we will present an effective way to make our transformations available to the designer.

A. Need for Interactive Recoding

Creating an MPSoC specification model from a reference C code involves some decision making and many mundane textual operations. Although most of the recoding can be automated, some of the decision making can only be done by the designer. For example, although the transformation to break a structure/vector is completely automatic, the decision about the specific variable to be split should be made by the designer. Although it is possible to automatically break all variables in the application, this is not desired. Note that the composite variables in the initial application are used for keeping related information together. By splitting more than necessary variables/loops, the readability of the code will be negatively affected. Further, to suite the target heterogeneous architectural platform, a target-specific modeling style needs to be employed. In essence, we believe that, at this first step in the design flow, it is critical for the designer to be in complete control. Our interactive approach therefore closely keeps the designer in the loop.

B. Source Recoder

Following these arguments, we have proposed an automatic source recoder [6]. Our source recoder is a controlled interactive approach to implement analysis and transformation tasks.

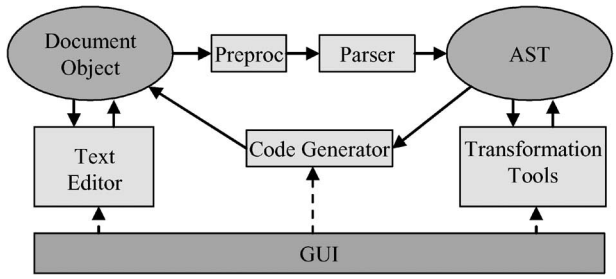

Fig. 9. Conceptual structure of the source recoder.

In other words, it is an intelligent union of editor, compiler, and powerful transformation and analysis tools. The conceptual organization of the source recoder is shown in Fig. 9. Unlike other program transformation tools, our approach provides complete control to generate and modify a specification model suitable for the design flow. By making the recoding process interactive, we rely on the designer to concur, augment, or overrule the analysis results of the tool, and use the combined intelligence of the recoder and the designer for the modeling task. Our recoder supports the remodeling of SLDL models at all levels of abstraction. It consists of the following five main components:

1) a textual editor (based on Qt and Scintilla) maintaining the textual document object;
2) an Abstract Syntax Tree (AST) of the design model;
3) preprocessor and parser to convert the document object into AST;
4) transformation and analysis tool set;
5) code generator to apply changes in the AST to the document object.

The parser and the code generator support C and SpecC source codes. The analysis results of each transformation are remembered in the AST and automatically get carried to the subsequent transformations. The transformations are *instantly* performed and presented to the designer. The designer can also make changes to the code by typing, and these changes are applied *on-the-fly*, keeping it updated all the time. More details of this interactive environment are discussed in [6].

VII. Experiments and Results

In this section, we will show that our designer-controlled approach is both feasible and effective. We will provide experiments and results that support four aspects of our approach.

1) We discuss the interactivity of the source recoder and show that our designer-controlled interactive approach is feasible and sufficiently responsive.
2) We describe a case study that creates a parallel and flexible specification of an MP3 audio decoder.
3) We provide the results of implementing the MP3 decoder specification using the SoC Environment (SCE) [24] to demonstrate the created flexibility and effectiveness.
4) We present the productivity gains achieved using our source recoder compared to manual coding of a specification model.

TABLE I
DESIGNER INTERACTIONS FOR DIFFERENT OPERATIONS

Operation	Number of interactions	Input by user
Setting loop parameters	1-5	Loop, index variable start index value, end index value, trip-count
CAT analysis	1	Loop
Loop splitting	2	Loop, No. of partitions
Vector splitting	2	Vector, No. of partitions
Structure splitting	1	Structure member
Synchronization	1	Target variable
Variable Re-scoping	1	Target variable

TABLE II
EXECUTION TIME OF DIFFERENT TRANSFORMATIONS [IN SECONDS]

Operations	Simple	JPEG	Fix-MP3	Float-MP3
Lines of code	174	1642	7086	7492
Setting loop parameters	0.0003	0.0003	0.02	0.059
CAT	0.019	0.013	0.017	0.017
Loop split	0.12	0.182	0.96	0.55
Vector split	0.18	0.249	0.95	0.6
Structure split	0.145	0.187	0.832	0.76
Synchronization	0.142	0.209	0.98	0.87
Re-scoping	0.13	0.239	0.83	0.62

A. Interactivity

In our approach, the task of creating a parallel specification is addressed through multiple iterative designer-controlled transformations instead of a single monolithic completely automatic compilation. In this section, we will look at the amount of interactivity associated with each of these transformations. Table I shows that only few designer interactions are needed to invoke these transformations. By interaction, we mean the number of inputs the designer has to provide to the transformation. The table also shows the inputs needed for each transformation. For example, before splitting a loop, it may be necessary to specify the loop parameters. Specifically, this operation requires only one designer interaction to specify the loop if the other parameters of the loop can be automatically determined as in the case of *for* loops with constant parameters. On the other hand, in the worst case, it requires five designer interactions to specify the loop, i.e., the index variable of the loop, start, end, and the trip count. Vector splitting requires two interactions, one to specify the vector, and one to specify the number of partitions. Both loop splitting and vector splitting are preceded by the step of setting loop parameters. Splitting structures takes one interaction to specify the split point, and rescoping of variables requires one interaction to specify the destination scope.

Table II shows the time to execute each of the transformations on a 3-GHz Pentium-4 Linux machine for different examples. The times indicated are in seconds, and the table shows the combined time to run the transformations and to update the graphical interface. However, it does not include the time to specify the inputs to the transformations. In general, the execution time of the transformations depends on the size of the example and the number of changes introduced by the transformation. The table shows the execution times for the transformations on some typical objects (loops and variables). We provide these results to show that our source recoder is sufficiently responsive on real-life embedded source codes (and not to prove the runtime efficiency of the algorithms). All times are less than a second even for the programs that span thousands of lines. They can be instantly applied and realized.

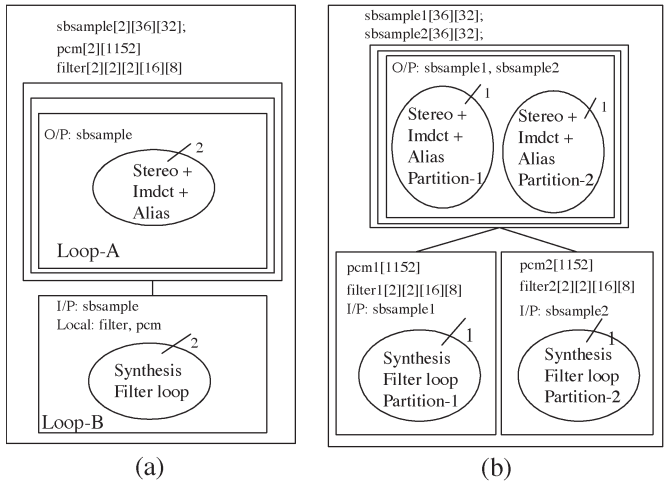

Fig. 10. MP3 code structure. (a) Sequential MP3 code structure. (b) Parallel MP3 code structure.

B. MP3 Design Example

As mentioned in the beginning of this paper, the main advantage of giving control to the designer is to enable the parallelization of a real-life embedded source code that requires application-specific knowledge (and cannot be done by the existing state-of-the-art parallelizing compilers). To corroborate this claim, we use our experiments with an industrial-strength design example, i.e., a fix-point MP3 audio decoder.[3] An abstract code structure of the reference sequential code [25] is shown in Fig. 10(a). The two loops implementing $Stereo + Imdct + Alias$ operations (Loop-A) and *Synthesis Filter* (Loop-B) are at a different functional hierarchy (indicated by rectangular boxes), and each loop spanned hundreds of lines of code. In the MP3 decoder, the processing of the left and right channels of a stereo MP3 stream is independent. However, this was not apparent in the reference code. We tried parallelizing the C code using the Intel C compiler [14], one of the few compilers that can detect coarse-grained parallelism on a shared-memory platform. The compiler was only able to detect five small loops implementing array copy and array initialization, each spanning just one to four lines of code. The computation within these loops was less than 2% of the overall computation of the application. The loops containing the algorithm-level parallelism in Fig. 10 could not be parallelized due to a function call within the loop, false dependencies, and the statically unknown trip count of the loops.

Thus, parallelizing compilers are ineffective in exploiting parallelism at the algorithmic level.

The array variables shown in Fig. 10 (*sbsample, filter*, and *pcm*) are the main vector data of interest between Loop-A and Loop-B. *sbsample* is the output of Loop-A and the input of Loop-B, whereas *filter* and *pcm* are only used by Loop-B. Since

[3]We have also performed the same transformations with a floating-point model of an MP3 decoder and obtained very similar results as shown in Section VII-C.

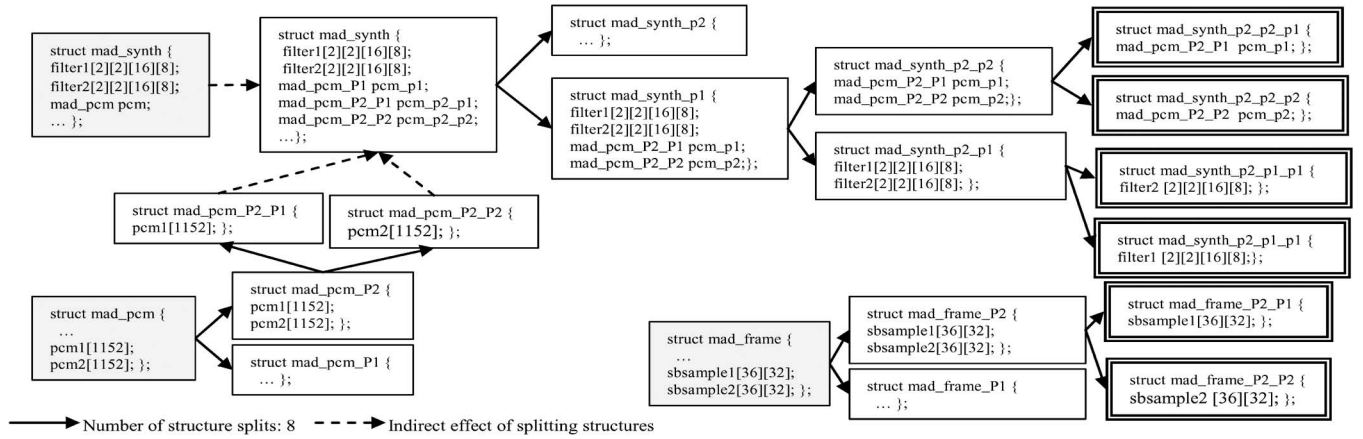

Fig. 11. Structures splitting in MP3 example.

the synthesis filter accounted for the most significant chunk of computation, we decided to parallelize Loop-B. Splitting Loop-B also requires splitting *filter* and *pcm* vectors so that they can be made private to each code partition. Since *sbsample* is also accessed in Loop-A, splitting it requires the generation of split data structures (copying of data from *sbsample* to *sbsample1*, *sbsample2*) at the end of loop-A. Since Loop-A and Loop-B had identical parameters (start, end, and iterations) and accessed *sbsample* using the same array references, the copying of data was easily avoided by splitting both loops. This simultaneous analysis of the two loops, which are in a different functional hierarchy, and considering the effect of splitting one vector in the other part of the program, requires a global knowledge of the program, which is only known to the designer. This is the main advantage of our approach. The designer can invoke the transformations on a specific program scope (context), independent of the rest of the program, and achieve the desired model by invoking the necessary complementing transformations on the dependent program scopes.

Further, the three vector splits were not standalone variables, but were part of the C structures (*mad_synth*, *mad_pcm*, and *mad_frame*), shown in shaded boxes in Fig. 11. In order to create a specification that can be easily mapped onto different processing elements, we need to explicitly separate the vector variables from these structures. Since each structure split generates two structures, a series of eight splits were necessary to separate the six vector partitions into independent structures, shown in emphasized boxes in Fig. 11.

In the end, using the transformations available in our source recoder, we arrived at the parallel code shown in Fig. 10(b). The transformations applied to arrive at this specification are summarized below.

1) Split Loop-B into two parts.
2) Find scalar and vector dependents in Loop-B using CAT analysis.
3) Split vector dependents *filter* and *pcm* at the first dimension.
4) Localize *filter1*, *filter2* and *pcm1*, *pcm2* to Loop-A.
5) Split Loop-A into two parts.
6) Split *sbsample* into two parts at the first dimension.
7) Iteratively split the three structures *mad_synth*, *mad_pcm*, and *mad_frame* to isolate the six vector pieces (eight splits in total are needed as shown in Fig. 11).
8) Localize the split vectors into each loop partition.

By having these transformations automated, we could arrive at the partitioned model in minutes, which would otherwise have taken hours. Since there were no scalar dependents between the two partitions, there exists no communication between the newly created partitions of the loop.

C. Implementation and Evaluation

The primary purpose of a flexible parallel specification is to enable multiple design explorations. With the model created in Fig. 10, we could explore two distributed parallel design alternatives and two shared-memory-based parallel architectures for both fixed-point and floating-point MP3 decoders. Fig. 12 shows these generic architectures. Note that the shared memory and the first-in first-out (FIFO) shown in Fig. 12 are mutually exclusive. That is, in the shared-memory implementation, the memory is used instead of the FIFO blocks, and vice versa for the distributed memory implementation. In the distributed version, the processing elements have private memories and communicate by passing the data over the buses. In the shared-memory architecture, a shared memory houses the variables shared by the three processing elements.

We used the *ARM7TDMI* and *Coldfire* processors for the fixed and floating-point implementations, respectively. Fig. 12(a) shows the main *decoder* mapped to the processors and the two parallel *synthesis filter* partitions mapped to two custom hardware blocks. Fig. 12(b) shows an alternative where the *decoder* and the *left filter* are executed on the processor and only the *right synthesis filter* is mapped to a hardware accelerator. Note that all these architectures are only possible due to the flexibility added by our transformations. We have implemented each design alternative using the SCE [24].

Tables III and IV show the results for the fix-point and floating-point implementations, respectively. For each design, the tables list the main components and their clock frequencies, as well as the performance achieved in decoding one frame of MP3 data. Note that, for a *bit rate* of 96 000 bits/s and *sampling*

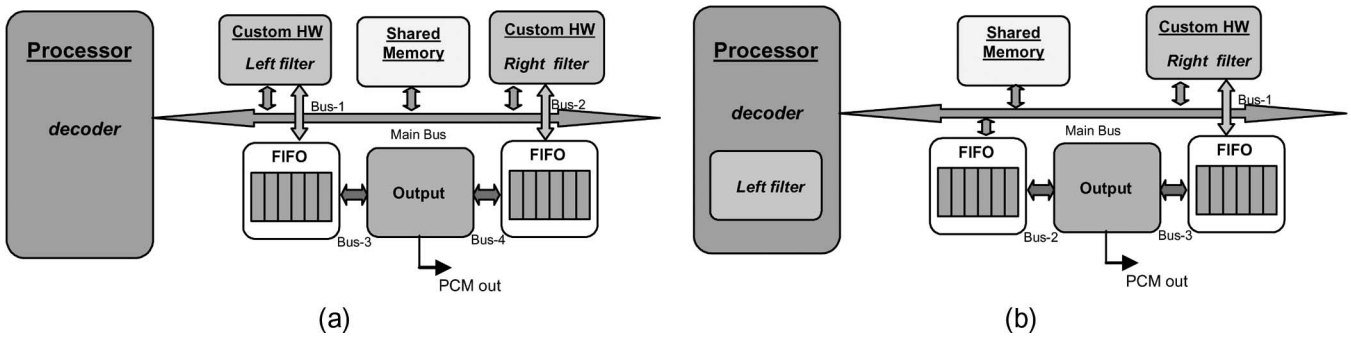

Fig. 12. Generic architectures explored for fix-point and floating-point MP3 decoder. (a) Multiprocessor architecture 1. (b) Multiprocessor architecture 2.

TABLE III
FIX-POINT MP3 DECODER IMPLEMENTATIONS

Models	Arch-1	Arch-2	Arch-3	Arch-4	Arch-5
	ARM7TDMI (50MHz)	ARM7TDMI (50MHz) 1 HW (100MHz)	ARM7TDMI (50MHz) 1 HW (100MHz) Shared Mem.	ARM7TDMI (50MHz) 2 HW (100MHz)	ARM7TDMI (50MHz) 2 HW (100MHz) Shared Mem.
TLM	48.62 ms	32.12 ms	32.12 ms	17.27 ms	17.27 ms
BFM	48.90 ms	33.83 ms	36.08 ms	20.33 ms	21.23 ms
< 26.12 ms	—	—	—	OK	OK

TABLE IV
FLOATING-POINT MP3 DECODER IMPLEMENTATIONS

Models	Arch-1	Arch-2	Arch-3	Arch-4	Arch-5
	Coldfire (66 MHz)	Coldfire (66 MHz) 1 HW (66 MHz)	Coldfire (66 MHz) 1 HW (66 MHz) Shared Mem.	Coldfire (66 MHz) 2 HW (66 MHz)	Coldfire (66 MHz) 2 HW (66 MHz) Shared Mem.
TLM	35.61 ms	21.83 ms	21.83 ms	18.84 ms	N/A
BFM	35.61 ms	21.99 ms	22.08 ms	19.07 ms	N/A
< 26.12 ms	—	OK	OK	OK	N/A

frequency of 44.1 kHz, a frame must be decoded in less than 26.12 ms. This timing constraint is only met by five out of the ten possible architectures, as shown in the last row of Tables III and IV. The floating-point architecture *Arch-5* was not feasible as the *Coldfire* bus cannot support more than two masters. Note that the required performance was met only due to the explicit parallelism exposed by our transformations. This clearly shows that the parallelism exposed in the specification using the source recoder is indeed effective.

D. Productivity Gains

As motivated in Section I, the manual recoding of a C reference code into an MPSoC specification is immensely time consuming. Automating this recoding task, even partially, can significantly reduce the overall design time of the system. In this section, we will estimate these productivity gains.

In addition to the MP3 example, we used our source recoder to parallelize and partition a Joint Photography Experts Group (JPEG) picture encoder. Table V shows the transformations performed on these examples (which could not be parallelized by [14]). The table lists the number of loops, vectors, and structures that were recoded. Table VI shows the time taken to implement these transformations. The automated recoding time is the time our source recoder took, including the time needed by the designer to enter his decisions in the dialog boxes. The manual time is the time we estimate it takes to manually implement the same transformations. For both MP3 decoders, we actually measured the manual time by manually implementing them. For the JPEG encoder, the amount of manual effort was estimated based on the experience with the MP3 decoder and the number of recoding operations needed to be performed. Both manual and automatic recoding were conducted by the same designer. The times indicated are only the recoding time and do not include the decision making time which will be the same in both cases.

TABLE V
IMPORTANT TRANSFORMATIONS APPLIED ON DIFFERENT EXAMPLES

Transformations	JPEG	Fix-Point MP3	Floating-Point MP3
Loops split	1	2	2
Vectors split	1	4	2
Structures split	0	8	3
Variables re-scoped	2	8	4

In general, measuring productivity is a difficult task. Factors such as designer's experience and tools used must be considered for the accurate measurement of productivity gains. Further, experiments with multiple designers are necessary. However, since we achieve productivity factors of multiple orders of

TABLE VI
PRODUCTIVITY GAINS

Properties	JPEG	Fix-Point MP3	Floating-Point MP3
Automatic Re-coding time	≈ 2 mins	≈ 5 mins	≈ 4 mins
Estimated Manual time	145 mins	480 mins	460 mins
Productivity factor	72	96	115

magnitude (see Table VI), we conclude that more accurate measurements will not significantly change the result that the approximate time to recode a model using our source recoder is on the order of minutes, whereas manual recoding takes hours. We conclude that, despite not being completely automatic, our controlled approach results in significant productivity gains.

VIII. CONCLUSION

Concurrency and flexibility are two critical features of an MPSoC specification. Concurrency in the specification is necessary to exploit the parallel resources available on MPSoCs. Flexibility in the specification is necessary for freedom in design space exploration.

Two main factors limit today's compilers in automatically generating a parallel and flexible MPSoC specification from a sequential monolithic application: first, the heterogeneous nature of MPSoCs with customized processors and nonregular memory hierarchy; and second, the complexity of the unstructured input application.

Completely automatic compilers, although successful in extracting instruction-level parallelism on shared-memory architectures, cannot expose a task-level parallelism that requires application-specific knowledge.

In this paper, we have proposed a designer-controlled approach to create a parallel and flexible MPSoC model. We proposed a set of six code and data partitioning transformations that can split loops and composite variables to expose concurrency and create flexibility.

In our approach, the task of creating a specification is based on iterative designer-controlled transformations instead of a monolithic completely automatic compilation. The discrete transformation steps are combined by the designer to create a desired specification model. Our transformations implement recoding tasks that are intuitive even to a programmer with limited compiler knowledge.

Our interactive source recoder integrates these transformations and interprocedural analysis functions with a text-based editor that assists the designer in modeling and remodeling the input specification. It can be employed on C and C-based SLDLs, including SpecC [5] and SystemC [7], to automatically perform complex transformations. Programming issues in the reference code, which limit the effectiveness of most of the existing compilers, are overcome by involving the designer in the loop.

In our designer-controlled environment, the transformations are quickly and efficiently applied on the program. Our experimental results show that this approach is feasible and effective. The added flexibility and parallelism result in a specification model that is most suitable for the target MPSoC platform and the design flow. Finally, we have shown that the automation of recoding results in a significant productivity gain.

In a broader perspective, the MPSoC programmability is a critical problem that needs to be addressed by further automation to keep up with the growing customer demand and rising system's complexity. The approach described in this paper can be seen as a significant first step into this direction. More research, however, is clearly necessary.

In a future work, we plan to work on further transformations. In particular, we would like to expose functional parallelism in addition to the presented data parallelism. We also intend to increase the analyzability and synthesizability of the design models.

ACKNOWLEDGMENT

The authors would like to thank the members of the SoC Environment Group, Center for Embedded Computer Systems, UC Irvine, for providing the SCE tool set for their experiments, and the editor and the reviewers for their valuable feedback to improve this paper.

REFERENCES

[1] M. Reshadi and D. Gajski, "A cycle-accurate compilation algorithm for custom pipelined datapaths," in *Proc. 3rd CODES+ISSS*, 2005, pp. 21–26.

[2] F. Dumitrascu, I. Bacivarov, L. Pieralisi, M. Bonaciu, and A. A. Jerraya, "Flexible mpsoc platform with fast interconnect exploration for optimal system performance for a specific application," in *Proc. Conf. DATE*, 2006, pp. 166–171.

[3] L. Benini and G. D. Micheli, "Networks on chips: A new SoC paradigm," *Computer*, vol. 35, no. 1, pp. 70–78, Jan. 2002.

[4] P. G. Paulin, C. Pilkington, M. Langevin, E. Bensoudane, and G. Nicolescu, "Parallel programming models for a multi-processor SoC platform applied to high-speed traffic management," in *Proc. 2nd CODES+ISSS*, 2004, pp. 48–53.

[5] A. Gerstlauer, R. Dömer, J. Peng, and D. D. Gajski, *System Design: A Practical Guide with SpecC*. Norwell, MA: Kluwer, 2001.

[6] P. Chandraiah and R. Dömer, "An interactive model re-coder for efficient SoC specification," in *Proc. IESS Embedded Syst. Des.: Topics, Techniques Trends*, A. Rettberg *et al.*, Eds., 2007, pp. 193–206.

[7] T. Grötker, S. Liao, G. Martin, and S. Swan, *System Design With SystemC*. Norwell, MA: Kluwer, 2002.

[8] S. Sutherland, S. Davidmann, P. Flake, and P. Moorby, *System Verilog for Design: A Guide to Using System Verilog for Hardware Design and Modeling*. Norwell, MA: Kluwer, 2004.

[9] M. Wolf and M. Lam, "A loop transformation theory and an algorithm to maximize parallelism," *IEEE Trans. Parallel Distrib. Syst.*, vol. 2, no. 4, pp. 452–471, Oct. 1991.

[10] M. H. Hall, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, and M. S. Lam, "Detecting coarse-grain parallelism using an interprocedural parallelizing compiler," in *Proc. ACM/IEEE Conf. Supercomputing*, 1995, p. 49.

[11] M. R. Haghighat and C. D. Polychronopoulos, "Symbolic analysis for parallelizing compilers," *ACM Trans. Program. Lang. Syst.*, vol. 18, no. 4, pp. 477–518, Jul. 1996.

[12] J.-H. Chow, L. E. Lyon, and V. Sarkar, "Automatic parallelization for symmetric shared-memory multiprocessors," in *Proc. Conf. CASCON*, 1996, p. 5.

[13] M. J. Wolfe, *High Performance Compilers for Parallel Computing*. Reading, MA: Addison-Wesley, 1995.

[14] A. Bik, M. Girkar, P. Grey, and X. Tian, "Efficient exploitation of parallelism on Pentium III and Pentium 4 processor-based systems," *Intel Technol. J.*, vol. 5, no. 1, pp. 1–9, Feb. 2001.

[15] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon, *Parallel Programming in OpenMP*. San Mateo, CA: Morgan Kaufmann, 2001.
[16] D. E. Culler, A. Gupta, and J. P. Singh, *Parallel Computer Architecture: A Hardware/Software Approach*. Amsterdam, The Netherlands: Elsevier, 1997.
[17] M. Fowler, "Refactoring: Improving the design of existing code," in *Proc. 2nd XP Universe, 1st Agile Universe Conf. Extreme Program. Agile Methods-XP/Agile Universe*, 2002, p. 256.
[18] *Index of Refactorings*. [Online]. Available: http://www.refactoring.com/catalog/index.html
[19] P. Chandraiah and R. Dömer, "Designer-controlled generation of parallel and flexible heterogeneous MPSoC specification," in *Proc. DAC*, 2007, pp. 787–790.
[20] P. Chandraiah, J. Peng, and R. Dömer, "Creating explicit communication in SoC models using interactive re-coding," in *Proc. ASPDAC*, Yokohama, Japan, Jan. 2007, pp. 50–55.
[21] M. E. Wolf, D. E. Maydan, and D.-K. Chen, "Combining loop transformations considering caches and scheduling," in *Proc. MICRO 29*, 1996, pp. 274–286.
[22] E. H.-Y. Tseng and J.-L. Gaudiot, "Two techniques for static array partitioning on message-passing parallel machines," in *Proc. Int. Conf. PACT*, 1997, p. 225.
[23] A. W. Lim, G. I. Cheong, and M. S. Lam, "An affine partitioning algorithm to maximize parallelism and minimize communication," in *Proc. 13th ICS*, 1999, pp. 228–237.
[24] S. Abdi, J. Peng, H. Yu, D. Shin, A. Gerstlauer, R. Dömer, and D. Gajski, "System-on-chip environment (SCE version 2.2.0 beta): Tutorial," Center Embedded Comput. Syst., Univ. California, Irvine, CA, Tech. Rep. CECS-TR-03-41, Jul. 2003.
[25] *MAD (MPEG Audio Decoder) Fix Point MP3 Algorithm Implementation*. MAD MP3 Decoder. [Online]. Available: http://sourceforge.net/projects/mad/

Pramod Chandraiah (S'05) received the B.E. degree in electronics and communication engineering, in 2000, from the University of Mysore, Mysore, India, and the M.S. degree, in 2005, from the University of California, Irvine (UCI), where he is currently working toward the Ph.D. degree in the Department of Electrical Engineering and Computer Science.

He is affiliated with the Center for Embedded Computer Systems, UCI. His research interest currently focuses on specification modeling and model optimization for system synthesis.

Rainer Dömer (S'95–M'00) received the Ph.D. degree in information and computer science from the University of Dortmund, Dortmund, Germany, in 2000.

He is currently an Assistant Professor in electrical engineering and computer science with the University of California, Irvine (UCI). He is also a member of the Center for Embedded Computer Systems (CECS), UCI. His research interests include system-level design and methodologies, embedded computer systems, specification and modeling languages, system-on-chip design, and embedded software.