Designer-Controlled Generation of Parallel and Flexible Heterogeneous MPSoC Specification

Pramod Chandraiah
Center for Embedded Computer Systems
University of California, Irvine
pramodc@uci.edu

Rainer Doemer
Center for Embedded Computer Systems
University of California, Irvine
doemer@uci.edu

ABSTRACT

Programming multi-processor systems-on-chip (MPSoC) involves partitioning and mapping of sequential reference code onto multiple parallel processing elements. The immense potential available through MPSoC architectures depends heavily on the effectiveness of this programming. Existing automatic parallelizing techniques, though effective on shared memory architectures, are insufficient for MPSoCs, which are typically characterized by heterogeneous processing elements and memory architectures. The lack of effective automatic techniques requires designers to manually partition the code and the data structures in the reference application to generate a parallel and flexible specification. Manual creation of this model is time consuming and error prone.
In this work, we present a novel designer-controlled approach to partition existing code and data structures automatically into a parallel and flexible abstract specification model that can be mapped to a heterogeneous MPSoC. Our results show significant productivity gains and improvements in the end design.

Categories and Subject Descriptors

D.1.2 [**Programming Techniques**]: Automatic Programming

General Terms

Algorithms, Design, Languages

Keywords

System specification, Code Partitioning, Re-coding, MPSoC

1. INTRODUCTION

The generation of a parallel specification model from a sequential reference application is the most time-critical and challenging task in programming MPSoC architectures. Concurrency and flexibility are two important characteristics of this model. Concurrency enables the efficient utilization of the underlying parallel architecture, and flexibility directly impacts the size of the design space that can be explored. Traditional parallelizing compilers can be used to expose concurrency. However, completely automatic techniques have been ineffective. First, most rely on a shared memory programming model and hence cannot handle heterogeneous MPSoC architectures such as specialized custom processors and non-uniform memory architectures. Second, though effective in parallelizing applications in scientific computing, completely automatic compilers have been ineffective in handling real-life embedded source codes. Third, more than often exposing concurrency in embedded applications requires algorithm knowledge which cannot be detected by the compilers. Flexibility in the model is necessary to perform large design space exploration. In general, flexibility increases with separation of computation and communication in the model. Moreover, flexibility directly depends on the number of distinct code and data partitions, which communicate through abstract channels. In the later stages of system synthesis, this flexibility increases the number of different code, data and channel mappings onto processors, memories and buses, respectively.
In case of low cost and low power MPSoCs, which cannot afford expensive cache coherence mechanisms, composite variables in the application need to be partitioned and explicitly localized so that they fit into small local memories.

1.1 Designer-Controlled Approach

Creating a parallel and flexible specification from a C model for a heterogeneous MPSoC involves identifying task-level parallelism, code partitioning, selective data structure partitioning and proper communication using abstract channels. This calls for *designer-controlled* parallelization, where the designer chooses the loops to be parallelized and the data structures to be partitioned and localized. Our approach is controlled by the designer to the extent that all the critical analysis and the time-consuming code transformations are performed by automatic transformations, but the decision to apply them is up to the designer. Our goal is to generate a flexible MPSoC specification model in a System Level Design Language (SystemC [7] or SpecC [6]) from a sequential reference C model. In this paper, we present a set of code/data partitioning and flexibility adding transformations using a designer-controlled approach.

1.2 Related Work

The problem of partitioning code and data structures in a reference application to generate a parallel MPSoC specification has not yet received much attention. The onus of performing this tedious recoding lies on the designer.
There has been extensive research in the parallel computing community to automatically parallelize applications. By means of inter-procedural analysis [9], symbolic analysis [8]

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

(a) Original loop

```
int a[32], b[16], c, d, x;
…
//loop partition 1                  //loop partition 3
for (i=0; i<4; i++) {               for (i=8; i<12; i++) {
    x = i *i;                           x = i *i;
    a[i]++;                             a[i]++;
    a[2i] = c+d;                        a[2i] = c+d;
    b[i] = c*d-x; }                     b[i] = c*d-x; }
//loop partition 2                  //loop partition 4
for (i=4; i<8; i++) {               for (i=12; i<16; i++) {
    x = i *i;                           x = i *i;
    a[i]++;                             a[i]++;
    a[2i] = c+d;                        a[2i] = c+d;
    b[i] = c*d-x; }                     b[i] = c*d-x; }
```

(b) Partitioned loops

Figure 1: Code changes resulting from loop splitting

and loop transformations, techniques to extract coarse-grained parallelism have been proposed for shared memory multiprocessors [4, 9, 14]. Intel C compiler [1] for Pentium-3 and 4 implements the above techniques to extract thread-level parallelism for shared memory architectures. In spite of these advanced analysis capabilities, completely automatic techniques have not been effective even for shared memory architectures. Application knowledge becomes necessary to parallelize many real-life applications. This has resulted in OpenMP compilers [2], where the programmer sets OpenMP directives to parallelize code. However, all analysis to ensure that a piece of code can be parallelized and 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 (MPI) [5].

2. CONTROLLED TRANSFORMATIONS

In this section, we present 4 program transformations that implement code and data partitioning to expose parallelism in loops, and encapsulate communication using channels to add flexibility to the model. All transformations are designer-controlled. Designer's application knowledge is used in each transformation step to resolve any dependencies that are not handled by the automatic analysis.

2.1 Loop Splitting

Loop splitting is one of many well-known transformations used in compiler optimization and parallelizing community [12, 13]. Our loop splitting transformation creates different incarnations of a loop with the same loop body in each split iterating over different contiguous portions of the loop index range. Depending on the trip count and the number of unrolls specified by the designer, the resulting partitions become loops with smaller trip count, or code segments with the induction variable completely replaced by constants. Figure 1 shows an example loop with trip count 16 uniformly split into 4 loops each with a trip count of 4. Non-uniform splitting would create splits with unequal index ranges.

Since the designer specifies the parameters of the loop for the transformation, we can also handle loops that cannot be parallelized by automatic compilers. For example, automatic compilers can parallelize only loops whose loop boundaries and trip count can be determined statically. However, in reference applications, it is common for a programmer to use *while* structures instead of *for* structures for loops. *while* loops can also be split if the loop parameters are known to the designer.

2.2 Cumulative Access Type Analysis

This static analysis of a loop reveals scalar and vector variables that are dependent between iterations of the loop. Variables written in one iteration and read in another are considered dependents. We classify cumulative accesses to variables within the loop into 4 categories, Read(R), Write(W), Write-Read(WR) and Read-Write(RW). Figure 1(a) shows variables with 3 different cumulative access types (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 to scalar dependents must be synchronized between the partitions to ensure correct semantics of the program after parallelization. Vector dependents are further analyzed and, if possible, partitioned to avoid unnecessary communication, as discussed in the next section.

2.3 Partitioning of Vector Dependents

If there are dependents between the loop partitions, then it is not possible to have a communication-free parallelism. This transformation splits vector dependents into contiguous sections across different partitions. Communication free partitioning is possible only if array references in different loop partitions do not depend on the same array element. This is a hard problem to solve in presence of sparse array accesses such as $A[B[i]]$. However, the problem is simpler if the array references are limited to affine expressions, which often is the case in embedded source codes. [11] provides techniques to partition an array in a loop across multiple processors for message passing parallel machines when the array references are of the form $x + b$, where x is the induction variable and b a constant. In our transformation, we conduct analysis of general affine expression $(mx+b)$, where m and b are constants.

Inputs to our algorithm are vector V, main loop L, split loops L_p where $0 \leq p \leq NP$, where NP is the number of loop partitions.

1. Check if the loop partitions access the same element of vector (v). 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 start S to end E in increments of Δx, the condition to be satisfied for these two references to be different in the entire iteration space is
$$m_1x + b_1 \neq m_2(x + k\Delta x) + b_2$$
where, $1 \leq k \leq ((E-S+1)/\Delta x)-1$ is the normalized iteration number. We can simplify the equation to
$$((m_1 - m_2)x + b_1 - b_2)/(m_2\Delta x) \neq k$$
$\forall k$: $1 \leq k \leq ((E-S+1)/\Delta x)-1$, $\forall x$: $S \leq x \leq E$. We test this inequality for all pairs of index expressions.
2. If step 1 is true, compute the vector partition start VPS_p and end VPE_p of v 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)*, re-

```
int a[32], b_part1[4], b_part2[4],
b_part3[4], b_part4[4], c, d, x;
…
//loop partition 1
for (i=0; i<4; i++) {
    x = i *i;
    a[i]++;
    a[2i] = c+d;
    b_part1[i] = c*d-x; }
//loop partition 2
for (i=4; i<8; i++) {
    x = i *i;
    a[i]++;
    a[2i] = c+d;
    b_part2 [i-4] = c*d-x; }
//loop partition 3
for (i=8; i<12; i++) {
    x = i *i;
    a[i]++;
    a[2i] = c+d;
    b_part3 [i-8] = c*d-x;
//loop partition 4
for (i=12; i<16; i++) {
    x = i *i;
    a[i]++;
    a[2i] = c+d;
    b_part4[i-12] = c*d-x;
```

Figure 2: Code after partitioning vector b

spectively. That is, $VPS_p = \min(IE_n^{\phi})$ and $VPE_p = \max(IE_n^{\psi})$, where

$$IE_n^{\phi} = \min(IE_n(S_p), IE_n(E_p)), \forall n: \ 1 \leq n \leq N$$
$$IE_n^{\psi} = \max(IE_n(S_p), IE_n(E_p)), \forall n: \ 1 \leq n \leq N$$

3. If the vector portions computed for each loop partition above are non-overlapping, ($VPE_p < VPS_{p+1}, \forall p$: $0 \leq p \leq NP-1$), then for each loop partition, create separate vector variables (V_p) with size $VPE_p - VPS_p + 1$.
4. Normalize the index expressions in each loop partition to account for the smaller vector sizes. An index expression $IE_n(x)$ in loop partition p is replaced with $IE_n(x) - VPS_p$.
5. Replace access to vector V in each loop partition L_p with access to vector partition V_p with the normalized index expression.

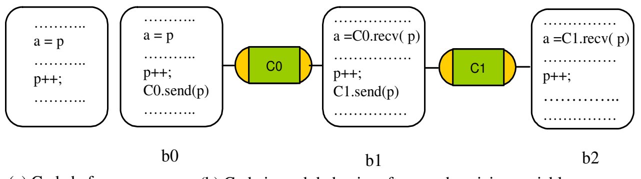
Figure 2 shows the loop partitions of Figure 1 after splitting the array b. Array a is unchanged as it cannot be split without communication to the other partitions. For example, the element 6 of array a is written in partition 1 and read in loop partition 2. If the split vector is used by other parts of the program, these partition details are remembered and correct split and merge codes are generated. For space limitations, this code generation not discussed in this paper.

2.4 Synchronization of Dependent Variables

To allow full parallelism, any scalar dependents in the loop have to be manually resolved by the designer. If this is not possible, the designer can choose to synchronize the access to the scalar dependents using channels. Synchronizing channels implement blocking *send()* and *receive()* communication. Figure 3 shows an example where the variable p, dependent across the behaviors *b0, b1, b2* is synchronized using 3 channels. To synchronize a variable requires the creation of $NP - 1$ channels, where NP is the number of code partitions. Following this, every read access is replaced by a *receive()* call and every write access by a *send()* call to the appropriate channel. This transformation, when applied to each dependent variable, will result in a semantically correct parallel specification.

3. SOURCE RECODER

We have integrated the transformations discussed in the previous section into our source re-coder along with other transformations, including re-scoping variables, creating behaviors and ports, etc. Source re-coder [3] is a controlled,



(a) Code before (b) Code in each behavior after synchronizing variable p

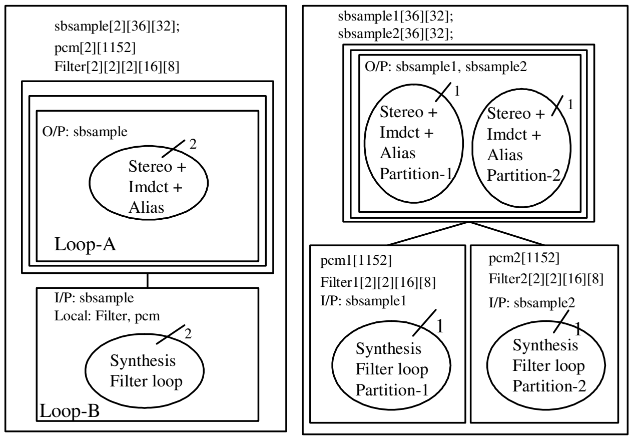
Figure 3: Synchronizing access to a scalar dependent

interactive approach to implement analysis and refinement tasks for MPSoC specification. It is an intelligent union of editor, compiler, and powerful transformation and analysis tools. Our re-coder supports re-modeling of SLDL models at all levels of abstraction. Analysis results of each transformation are kept in an abstract syntax tree and get carried to subsequent transformations automatically. The transformations are performed and presented to the designer *instantly* in the source code. The designer can also modify the code by typing and these changes are applied *on-the-fly* to the data structures, keeping it updated all the time.

4. EXPERIMENTS AND RESULTS

As discussed, the main advantage of giving control to the designer is to enable parallelization of real-life embedded source code which requires application-specific knowledge. Thus, we can parallelize code that cannot be handled by existing state-of-the-art parallelizing compilers. To corroborate this claim, we will now show our experiments with an industrial-strength design example, a MP3 audio decoder. An abstract code portion of the sequential reference code [10] is shown in Figure 4(a). 2 loops implementing *Stereo+Imdct+Alias* operations (Loop-A) and *Synthesis Filter* (Loop-B) exist at different functional levels (indicated by rectangular boxes). Each loop spans a few hundred lines of code. In the MP3 decoder, the processing of the left and right channels of a stereo MP3 stream are independent of each other. However, this is not apparent in the reference code. We tried parallelizing the C code using the Intel C compiler [1], one of the few compilers that can detect coarse-grained parallelism on a shared memory platform. The compiler only detected 5 small loops implementing array copy and initialization, each spanning 1 to 4 lines of code. The computation in these loop was less than 2% of the overall computation in the application. The loops with hidden parallelism in Figure 4 could not be parallelized due to a function call within the loop, false dependencies, and due to unknown trip count of the loops.
The main array variables of interest, *sbsample, filter, pcm,* and their I/O relation with respect to Loop-A and Loop-B is shown in Figure 4. Since the synthesis filter accounts for most of the 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 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, iterations) and accessed *sbsample* using the same references, copying of data was easily avoided by splitting both loops. Using the transformations available in our source recoder, we quickly arrived at the parallel code shown in Figure 4(b), after 6 easy transformations:

- Split Loop-B into 2 parts



(a) Sequential MP3 code structure (b) Parallel MP3 code structure

Figure 4: MP3 code structure

- Find scalar, vector dependents using CAT analysis
- Split vector dependents *filter*, *pcm* at first dimension
- Localize *filter1, filter2* and *pcm1, pcm2* to Loop-B
- Split Loop-A into 2 parts
- Split *sbsample* into 2 parts at first dimension

Having these transformations automated, we arrived at the partitioned model in minutes (which otherwise would take hours). Since there were no scalar dependents between the two partitions, we need no communication between the new partitions of the loop. With just the model in Figure 4, we could explore 2 design alternatives. Because now such a model can be generated quickly, we could partition other loops in the program to create a flexible model with more partitions enabling larger design space explorations.
Besides the MP3 example, we used our recoder to parallelize and partition other practical applications. Table 1 shows details of the transformations performed that could not be parallelized by [1]. Table 1 lists the number of loops and vectors that were affected by different transformations. The other operations mainly include variable re-scoping and localization. The automated recoding time is the time it takes to parallelize using the source recoder, and manual time is the estimated time to implement the same manually. In spite of not being completely automatic, this controlled approach results in a productivity gain in the order of 100.

5. CONCLUSIONS

Concurrency and flexibility are critical features of an MPSoC specification. Concurrency is necessary to exploit the parallel resources available on the MPSoC. Flexibility is necessary for freedom in design space exploration. The heterogeneous nature of MPSoCs and the complexities posed by unstructured input applications severely limit today's automatic compilers in generating parallel MPSoC code from a sequential monolithic application. Completely automatic compilers, though successful in extracting instruction level parallelism, do not offer much help in exposing task-level parallelism. This requires application-specific knowledge.
In this paper, we proposed a designer-controlled approach to create a parallel and flexible MPSoC specification model in a system level language. We developed a set of code and data partitioning transformations that can split loops and vector variables to expose concurrency. Our interactive source recoder integrates our transformations and inter-procedural analysis functions in a text-based editor to assist the designer in modeling and re-modeling of an input specification. It can be employed on C and C-based SLDLs, including SpecC and SystemC, to automatically perform code transformations. In this designer-controlled environment, the automated transformations can be applied to the program quickly and efficiently, to generate a specification that is most suitable for the application on the MPSoC platform. Limitations of existing parallelizing compilers are overcome by the designer-in-the-loop. Our experiments on real-life examples show significant productivity gains, and prove the effectiveness of our approach.

Table 1: Example productivity gains

Properties	JPEG	Fix-Point MP3	Floating-Point MP3
Loops split	1	2	2
Vectors split	1	4	2
Other operations	2	9	6
Automatic Re-coding time	≈2 mins	≈4 mins	≈3 mins
Estimated manual time	85 mins	360 mins	340 mins
Productivity factor	42	90	113

6. REFERENCES

[1] A. Bik et al. Efficient Exploitation of Parallelism on Pentium III and Pentium 4 Processor-Based Systems. *Intel Technology Journal*, 2001.

[2] R. Chandra et al. *Parallel programming in OpenMP.* Morgan Kaufmann Publishers Inc., 2001.

[3] P. Chandraiah and R. Dömer. An interactive model re-coder for efficient SoC specification. In *IESS*, 2007.

[4] J.-H. Chow et al. Automatic parallelization for symmetric shared-memory multiprocessors. In *CASCON*, 1996.

[5] D. E. Culler, A. Gupta, and J. P. Singh. *Parallel Computer Architecture: A Hardware/Software Approach.* Morgan Kaufmann Publishers Inc., 1997.

[6] D. D. Gajski et al. *SpecC: Specification Language and Design Methodology.* Kluwer Academic Publishers, 2000.

[7] T. Grötker et al. *System Design with SystemC.* Kluwer Academic Publishers, 2002.

[8] M. R. Haghighat et al. Symbolic analysis for parallelizing compilers. *ACM Trans. Program. Lang. Syst.*, 18(4), 1996.

[9] M. H. Hall et al. Detecting coarse-grain parallelism using an interprocedural parallelizing compiler. In *Supercomputing*, 1995.

[10] MAD MP3 Decoder. Fix point MP3 algorithm implementation. http://sourceforge.net/projects/mad/.

[11] E. H.-Y. Tseng and J.-L. Gaudiot. Two Techniques for Static Array Partitioning on Message-Passing Parallel Machines. In *PACT*, 1997.

[12] M. Wolf and M. Lam. A Loop Transformation Theory and an Algorithm to Maximize Parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 02(4), 1991.

[13] M. E. Wolf et al. Combining loop transformations considering caches and scheduling. In *MICRO 29*, 1996.

[14] M. J. Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley, 1995.