A Novel Memory Size Model for Variable-Mapping In System Level Design

Lukai Cai, Haobo Yu, Daniel Gajski
Center for Embedded Computing Systems
University of California, Irvine, USA
{lcai,haoboy,gajski}@cecs.uci.edu

Abstract— It is predicted that 70% of the chip area will be occupied by memories in future system-on-chips. The minimization of on-chip memory hence becomes increasingly important for cost, performance and energy consumption. This paper proposes a novel memory size model for algorithms which map the variables of a system behavior to memories of a system architecture. To our knowledge, it is the first memory estimation approach that analyzes the variable lifetime for the system behavior, which consists of hierarchically-modelled and concurrently-executed processes and contains variables with different sizes. Experimental results show that significant improvements can be achieved.

I. Introduction

It is predicted that 70% of the chip area will be occupied by memories in future system-on-chips. The minimization of on-chip memory hence becomes increasingly important for cost, performance and energy consumption. Minimizing memory not only requires smart algorithms to efficiently map/bind the variables/codes of a system behavior into the memories of a system architecture, but also demands accurate estimation of the system memory size occupied by a system behavior .

In order to minimize the memory size, many memory/register allocation algorithms have been proposed in the field of compiling optimization and high level synthesis. However, they cannot be employed at the system level design because of the following characteristics of a system behavior.

1. Concurrency and hierarchy. Both concurrency and hierarchy exist in the system behavior, which cannot be handled by traditional high level synthesis and compiling optimization techniques.

2. Inaccurate lifetime estimation. In the compiling optimization or high-level-synthesis domains, the lifetime of variable can be accurately determined in terms of variable's crossed statements/instructions in the code sequence or crossed states in the FSM. On the other hand, in the system design domain, the lifetime of variable is estimated in coarse-grain by system estimator such as VCC [1]. Such system level estimation never guarantees accuracy.

3. Variable size variety. In high level synthesis or compiling domains, the allocated variables/registers have the same size. However, the variables in system design domain have different sizes.

This paper proposes a novel memory size model to estimate the required memory size for variable mapping problems which maps the variables of a system behavior to the memories of a system architecture. In order to test the ability of the proposed model, we also introduce a straightforward variable mapping algorithm and apply it on the vocoder project.

The paper is organized as follows: Section II introduces related work. Section III then defines the variable mapping problem that we target at. Section IV analyzes the variable lifetime at system level. Section V introduces the memory size model. Section VI describes the proposed variable mapping algorithm. The experimental result is described in section VII. Finally, section VIII gives the conclusion.

II. Related Work

Lots of research has been done on system level design for years. Some of them took the memory issue into account.

During behavior-PE mapping, Prakash and Parker [8] adds the cost of memories determined by the amount of memories to the cost equation of design. All the variables are mapped to local memories and the processes mapped to one PE are executed sequentially. Therefore, the amount of memory required by the PE must be equal to the largest amount that is required by any of the process mapped to that PE.

Szymanke and Kuchcinshi [9] implements behavior-PE mapping by applying the similar memory model as [8]. The difference between [9] and [8] is that when two processes communicate through a variable, [9] reserves the memory for the variable in the sending process until the value of the variable is transferred to the receiving process.

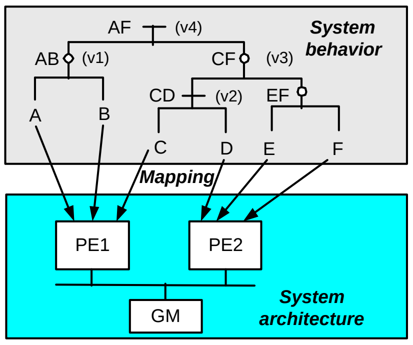


Fig. 1. Input of variable memory mapping problem

The objective of Meftali *et al.* [6] is *variable memory mapping*. They map variables to either one of the local memories of PEs or to the global memory. All the variables cannot share the memory region due to the lack of lifetime analysis.

Panda *et al.* [7] map variables into Scratch-Pad memory and off-chip DRAM accessed through data cache, in order to maximize the performance. They take the variable size variety into account.

In comparison to our work, none of above works support behavior hierarchy and behavior concurrency. [8][9][6] either simplifies or ignores lifetime analysis.

The previous work on storage requirement computation are well studied in the high level synthesis domain. Gajski *et al.* [4] surveys related scalar-based algorithms including clique partitioning, left-edge, and weighted-bipartite-matching algorithms. Some array-based algorithms are also proposed by Verbauwhede *et al.*[10], Zhao and Malik [11], and Balasa [2]. However, because of the characteristics of the system behavior described in section I, those algorithms cannot be directly applied to the system level.

III. Problem Definition

This paper follows system synthesis approaches, which contain the following three design steps. Firstly, *behavior mapping* task selects processing elements (PEs) from a PE library and maps the processes of a system behavior which represents the design functionality to the selected PEs. Secondly, *variable mapping* task selects global memories and maps variables of processes to the local memories of PEs or the global memories. Thirdly, *channel mapping* task selects the system buses, determines the bus topology, and maps the communication among the processes on different PEs to the buses. Although above three tasks are interdependent, we accomplish them independently in order to reduce the task complexity to a doable level. When implementing *variable mapping* task, because the system bus details are completely unknown due to lack of *channel mapping*, we use the total number of bytes transferred among PEs to evaluate the bus traffic and communication time.

The variable memory mapping problem has three inputs: a system behavior, a system architecture, and the process-PE mapping decisions.

We model the system behavior hierarchically using *behavior hierarchy tree* displayed at the top of Figure 1. Behavior hierarchy tree consists of two types of nodes: A leaf node represents a leaf process, and a hierarchy node represents a hierarchy process. Any two child nodes of a hierarchy node ω are executed either sequentially (denoted by "**-**" in Figure and recorded as $parallel(\omega) = 0$), or concurrently (denoted by "**o**" in Figure and recorded as $parallel(\omega) = 1$), in terms of the design functionality. For example, leaf nodes A and B are executed concurrently. Therefore $parallel(AB) = 1$.

Different variables are declared in different processes. If a variable ν is declared in a process ω, we denote that ν is ω's *associated variable*, and ω is ν's *associated process*. A variable associated with ω is used by the offspring processes of ω. In Figure 1, variable *v1* is associated with process AB and are used by processes A and B.

We model the system architecture similar to [6], which is illustrated at the bottom of Figure 1. The system architecture contains a number of PEs and a global memory (called GM) connected by system buses. A PE is either a micro-processor (SW PE) or a custom hardware (HW PE). A SW PE can have either a preemptive operating system or a non-preemptive operating system. Each PE has a local memory. Before variable-mapping, PEs have been selected and the sizes of local memories of PEs are pre-defined. Both the size of global memory and the system bus details are unknown.

The process-PE mapping decisions are also predefined before variable memory mapping. The processes in the system behavior are mapped to the allocated PEs in the system architecture, which is illustrated by arrows between the system behavior and the system architecture in Figure 1.

The objective of *variable mapping* is to minimize the required global memory size as well as to minimize the generated traffic on the system architecture, by mapping variables of processes to architecture memories .

Our variable mapping algorithm contains three tasks. First, We analyze the lifetime of variables at the system level. Second, we develop a novel memory size model which computes the required minimal memory sizes. Finally, we propose a variable-mapping algorithm based on the memory size model.

IV. Lifetime Analysis

We analyze the variable lifetime at the system level, which assumes the lifetime of any variable υ equals to the lifetime of its associated process. This approach differs from lifetime analysis at RTL (register-transfer-level) level which analyzes the lifetime of each variable independently. If necessary, before using the proposed algorithm, designers can first group variables in each process to a number of pseudo variables whose lifetime equals to its associated process's lifetime, by analyzing variable lifetime at the RTL level.

For a leaf process without calling any other processes, we define its lifetime as the duration between its start time and its end time. For a hierarchical process which calls at least one processes, we define its lifetime as the union of the lifetimes of its child processes. For example, in Figure 1,

$$lifetime(AB) = lifetime(A) \bigcup lifetime(B);$$

Based on the fact that the lifetime of sequential executing processes are disjoint and the lifetime of parallel executing processes are overlapped, we define the following three theorems to determine overlapping relations of the lifetime among processes.

Theorem1: For a process ω, if $parallel(\omega) = 0$ (sequential), then the lifetime of all its child processes is disjoint to each other.

Theorem2: If processes $\omega 1$ and $\omega 2$ are disjoint, then any offspring process of $\omega 1$ is disjoint with any offspring process of $\omega 2$.

Theorem3: If two processes do not satisfy the Theorem1 and Theorem2, then they are overlapped.

For example, in Figure1, processes AB and CF are disjoint according to *Theorem1* and processes A and C are disjoint according to *Theorem2*. Since the variable lifetime equals to the lifetime of its associated process, the overlapping relations between variables are also determined.

Rather than based on inaccurate system level estimation, the proposed approach analyzes variable lifetime based on behavior hierarchy and concurrency, which ensures the correctness of analysis.

V. Memory Size Model

A. Problem Definition for Memory Size Model

Assuming ω denotes a process in a behavior hierarchy tree. ρ denotes a local memory of a PE, or a global memory. After mapping a certain number of variables to the memory ρ, we face two problems:

Problem1: What is the required memory size of ρ?

Problem2: Whether ρ has room for the next mapped variable v?

Because different variables that are associated with different processes have different lifetime, we further refine the *Problem2* to:

Problem2b: Whether ρ has room for the next mapped variable v associated with ω?

We build the memory size model to answer above questions. We define the memory size model for pair (ω, ρ) as

$$M(\omega, \rho) = (\alpha(\omega, \rho), \beta(\omega, \rho), \lambda(\omega, \rho))$$

where α denotes the self-used memory size, β denotes the total-used memory size, and λ denotes un-used memory size.

B. Self-Used Memory Size $\alpha(\omega, \rho)$

$\alpha(\omega, \rho)$ represents the size of memory of ρ occupied by the ω's associated variables that have been mapped to ρ. $\alpha(\omega, \rho)$ is defined as:

$$\alpha(\omega, \rho) = \sum_{v_i \in S\_MV(\omega, \rho)} size(v_i, \rho)$$

where $S\_MV(\omega, \rho)$ denotes the set of ω's associated variables which have been mapped to ρ.

C. Total-used Memory Size $\beta(\omega, \rho)$

$\beta(\omega, \rho)$ not only contains $\alpha(\omega, \rho)$, but also contains the size of memory occupied by the offspring processes of ω. $\beta(\omega, \rho)$ is defined as:

If $parallel(\omega, \rho) = 1$, then

$$\beta(\omega, \rho) = \alpha(\omega, \rho) + \sum_{\theta \in S\_B(\omega)} \beta(\theta, \rho)$$

If $parallel(\omega, \rho) = 0$, then

$$\beta(\omega, \rho) = \alpha(\omega, \rho) + \max_{\theta \in S\_B(\omega)} \beta(\theta, \rho)$$

where $S\_B(\omega)$ denotes the set of ω's child processes. $parallel$ (ω, ρ) represents whether the child processes of ω are executed concurrently (=1) or sequentially (=0) in ρ. The computation of $parallel(\omega, \rho)$ will be explained in subsection E.

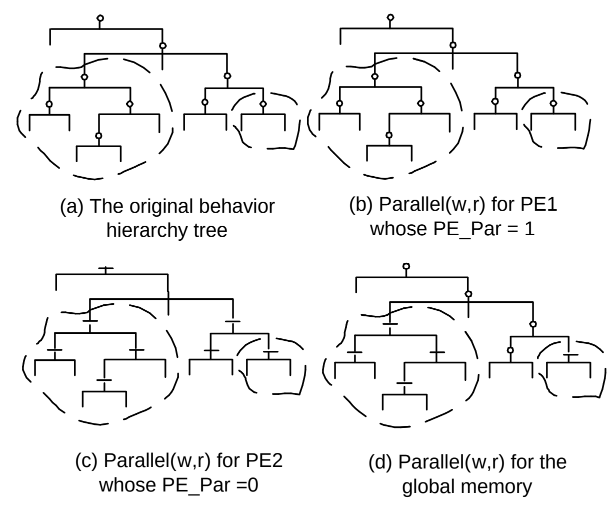
The formulation is achieved according to the following fact: if the child processes of ω are executed concurrently, then their lifetime is overlapped. Therefore, the memory required by ω's child processes equals to the sum of total-used memory sizes of ω's child processes. On the other hand, if child processes of ω are executed sequentially, then the lifetime of them is disjoint. Therefore, The child processes can share the same region of memory ρ. In this case, the memory required by ω's child processes equals to the largest total-used memory size of ω's child processes.

D. Un-used Memory Size $\lambda(\omega, \rho)$

$\lambda(\omega, \rho)$ represents the un-used memory size of ρ, which is available to store ω's associated variables that have not been mapped to any memories yet. $\lambda(\omega, \rho)$ can be computed only after the memory size of ρ called $size(\rho)$ is given. $\lambda(\omega, \rho)$ is defined as:

If ω is the root node, then
$\lambda(\omega, \rho) = size(\rho) - \beta(\omega, \rho)$
Otherwise,
if $parallel(parent(\omega), \rho) = 1$,
$\lambda(\omega, \rho) = \lambda(parent(\omega), \rho)$
if $parallel(parent(\omega), \rho) = 0$,
$\lambda(\omega, \rho) = \lambda(parent(\omega), \rho) + \beta(parent(\omega), \rho)$
$-\alpha(parent(\omega), \rho) - \beta(\omega, \rho)$

where $parent(\omega)$ denotes the parent node of ω.



(a) The original behavior hierarchy tree
(b) Parallel(w,r) for PE1 whose PE_Par = 1
(c) Parallel(w,r) for PE2 whose PE_Par =0
(d) Parallel(w,r) for the global memory

Fig. 2. Example of $parallel(\omega,\rho)$ computation

The formulation is achieved according to the following fact: if ω is the root node, then un-used memory size equals to the total memory size $size(\rho)$ subtracted by total-used memory size $\beta(\omega,\rho)$. If ω is a non-root node whose $parallel(\ parent(\omega),\rho) = 1$, then ω is concurrently executed with $parent\ (\omega)$'s other child processes. In this case, during ω's execution, ρ needs to reserve memory not only for itself, but also for $parent(\omega)$'s other child processes, Therefore $\lambda(\omega,\rho)$ equals to $\lambda(parent\ (\omega,\rho))$. On the other hand, if ω is a non-root node whose $parallel(parent(\omega,\rho),\rho) = 0$, then ω is executed sequentially with $parent(\omega)$'s other child processes. In this case, during ω's execution, ρ doesn't reserve memory for $parent(\omega)$'s other child processes. Because $parent(\omega)$ reserves memory for the largest β of its child processes ($=\max_{\theta\in S\_B(parent(\omega))}\beta(\theta,\rho)$) and ω only reserves $\beta(\omega,\rho)$, the difference between them is added on $\lambda(parent(\omega,\rho))$ to represent $\lambda(\omega,\rho)$. As shown by the second equation in subsection C, the largest β of $parent(\omega)$'s child processes equals to $\beta(parent(\omega),\rho) - \alpha(parent(\omega),\rho)$. Therefore, $\lambda(\omega,\rho) = \lambda(parent(\omega),\rho) + (\beta(parent(\omega),\rho) - \alpha(parent(\omega),\rho)) - \beta(\omega,\rho)$.

E. Memory Type Model

We compute $parallel(\omega,\rho)$ according to whether ρ represents a local memory or a global memory.

If ρ represents a local memory of a PE, we use $PE\_par(\rho) = 1/0$ to denote whether the lifetime of multiple processes executed on ρ can/cannot be overlapped, i.e. whether parallel execution of processes on ρ is allowed.

For a SW PE , if its operating system supports preemptive schedule, then $PE\_par = 1$. This is because that one process may be preempted by another, which causes the overlapping lifetime . In this case, ρ reserves local memory not only for the preempting process, but also for the preempted process. On the other hand, if SW PE's operating system doesn't support preemptive schedule, then $PE\_par = 0$.

For a HW PE , if two processes can be run concurrently on it, then $PE\_par = 1$, otherwise $PE\_par = 0$. A simple example of running two processes on one HW PE concurrently is that the HW PE contains two controllers and two datapaths.

As a result, for a local memory of a PE ρ, we define

```
if ($PE\_par(\rho) = 1$ and $parallel(\omega) = 1$)
then $parallel(\omega,\rho) = 1$;
else $parallel(\omega,\rho) = 0$;
```

Assume that ρ represents a global memory. The variable mapped to a global memory may associate with processes mapped to different PEs, which can be executed concurrently if the design functionality allows so. Therefore, if $parallel(\omega) = 1$, during ω's execution, the global memory ρ must reserve the memory for all the variables which are not only associated with ω's different offspring processes but also mapped to ρ. Furthermore, since all the processes mapped the PE whose PE_par is 0 have been sequentialized, we disable the concurrency existing in the sub-tree mapped to that PE.

Therefore, for a global memory ρ, we define

```
if ($parallel(\omega) = 0$) or
    (all the offspring processes of $\omega$ are mapped to a PE whose $PE\_par = 0$)
then $parallel(\omega,\rho) = 0$;
else $parallel(\omega,\rho) = 1$;
```

We illustrate $parallel(\omega,\rho)$ computation by Figure 2. Figure 2(a) shows the original behavior hierarchy tree reflecting the system behavior. For any node ω, "**o**" denotes $parallel(\omega) = 1$ while "**-**" denotes $parallel(\omega) = 0$. Assuming we map the processes in dotted circle to $PE2$ whose PE_par is 0 and map other processes to a PE1 whose PE_par is 1. Figure 2(b),(c), and (d) display the value of $parallel(\omega,\rho)$ in the behavior hierarchy trees, for local memory of $PE1$, local memory of $PE2$, and the global memory respectively. In these figures, "**o**" denotes $parallel(\omega,\rho) = 1$ while " **-**" denotes $parallel(\omega,\rho) = 0$.

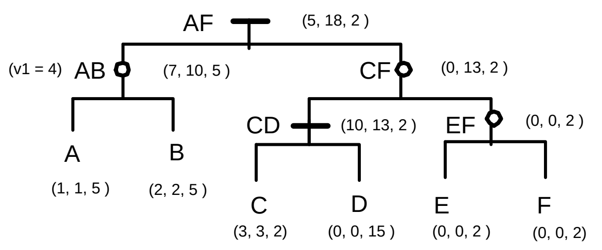
F. Answers from Memory Size Model

According to memory size model, we provide the solutions to *Problem 1* and *Problem 2b*

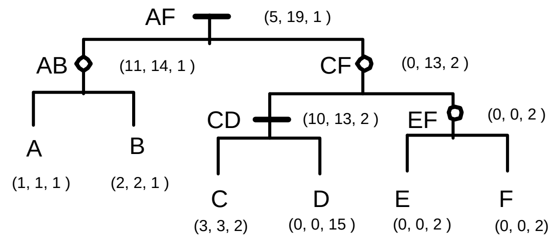
Answer1: The required memory size of ρ for process ω equals to $\beta(\omega,\rho)$. The required memory size of ρ for the entire design equals to $\beta(\varpi,\rho)$, where ϖ denotes the root node (main process) .

Answer2b: Assuming an unmapped variable v is associated with process ω. During v mapping, if $\lambda(\omega,\rho) \geq size(v,\rho)$, then v can be mapped to ρ. Otherwise, it cannot be mapped to ρ.

For example, the memory size model of PE ρ is displayed in Figure 3(a). The triple aside each node ω represents its memory size model $(\alpha(\omega,\rho),\beta(\omega,\rho),\lambda(\omega,\rho))$.



(a) Memory size model before v1 mapping



(b) Memory size model after v1 mapping (v1 = 4)

Fig. 3. Example of memory size model

The total memory size is 20 and the total un-used memory size of ρ is 2 as shown by $\lambda(AF, \rho)$. Assuming the next mapped variable is v1 which is associated with node AB. If *size(v1, ρ)* is 6, which is greater than $5(\lambda(AB, \rho))$, according to *Answer2b*, it cannot be mapped to ρ. However, if *size(v1, ρ) = 4*, according to *Answer2b*, it can be mapped to ρ. The updated memory size model after $v1$ mapping is displayed in Figure 3(b). According to *Answer1*, the required memory size for ρ equals to $\beta(AF, \rho)$, which is 19. The un-used memory sizes for processes AF, AB, A, and B are decreased while the un-used memory sizes for other processes remain the same.

VI. Variable Mapping Algorithm

We introduce a variable mapping algorithm which takes advantages of the proposed memory size model.

The algorithm first identifies global variables and local variables. If the offspring processes of variable v's associated process are mapped to different PEs, then we define v as a global variable. Otherwise, it is a local variable. At the begining, We also initialize memory size models of the local memories of PEs and the global memory.

Second, we map the code segment and all the local variables of any process ω to the local memory of the PE that ω is mapped to. After mapping, the memory size models of local memories are computed. We assume that the local memory of PE must be large enough to store all the code segment and local variables of the processes executed on it.

Third, we map global variables. There are two variable mapping alternatives. The first alternative makes local copies of the global variable inside variable's connecting PEs and synchronizes them using message passing mechanism. The connecting PEs of a variable v is defined as the PEs to which v's associated process ω and ω's offspring processes are mapped. The second alternative maps global variables to the global memory and lets processes use shared-memory mechanism to communicate.

If we map a global variable to each of its connecting PEs, its read access by process ω doesn't produce traffic on the system buses because it reads from the local memory of the PE that process ω mapped to. However, because it writes the new value to each local memory of its connecting PEs through system buses, its write access produces system traffic. On the other hand, if we map a global variable to a global memory, because processes must access global memory through system buses, both variable's read and write access produce system traffic . Since the difference between two mapping alternatives is read access, we prefer mapping variables with larger read access to the local memories.

During global variable mapping, we first order global variables in the decreasing order of read access. Then we map one global variable v at each iteration. If all of the local memories of v's connecting PEs have enough unused memory to store v according to *Answer2b*, then we map v to local memories of all of its connecting PEs . Otherwise, we map v into the global memory. After the mapping decision of v is made, the α, β, and λ of corresponding memory size models are updated. After all the variables are mapped, the required memory sizes of local memories and global memory are computed according to *Answer1*. The detailed algorithm is introduced in [3].

PE	DSP1	DSP2	HW1	HW2
Opti1(kB)	1708	1110	720	2308
Opti2(kB)	1962	1296	720	2308
N-Opti(kB)	2261	1434	723	2308

TABLE I
Computed local memory sizes for Vocoder project

VII. Experimental Result

We have implemented the proposed algorithm by programming around 3000 lines of C++ code. In this section, the experimental result on Vocoder project [5] is introduced, which shows the advantages of the proposed memory size model for a complex system design.

The Vocoder implements the voice encoding part of the GSM standard for mobile telephony encoding standard. The simplified block diagram of Vocoder is displayed at the top of Figure 4. It has 13,000 lines of code, and seven hierarchial levels. It contains 102 processes and 156 variables.

We select the system architecture containing four PEs shown at the bottom of Figure 4. $DSP1$ and $DSP2$ are Motorola DSP56600 microprocessors. $HW1$ and $HW2$ are custom hardwares whose *PE_par* equal to 0. One global memory $GMem$ is also instantiated in the system architecture. The behavior-PE mapping decision is denoted by dotted lines and matching shading styles in Figure 4.

	Input: local memory sizes(Byte)		Output: global memory sizes(Byte)			Output: traffic (KByte)		
	DSP1	DSP2	opt1	opt2	n_opti	opt1	opt2	n_opti
case1	1600	1000	145	305	434	119643	237655	483037
case2	1700	1100	57	225	345	117035	172455	296987
case3	1800	1200	0	101	265	116709	118339	205055
case4	1900	1300	0	101	145	116709	118339	119643
case5	2000	1400	0	0	57	116709	116709	117035

TABLE II
Table of required global memory sizes and generated system traffic for Vocoder project

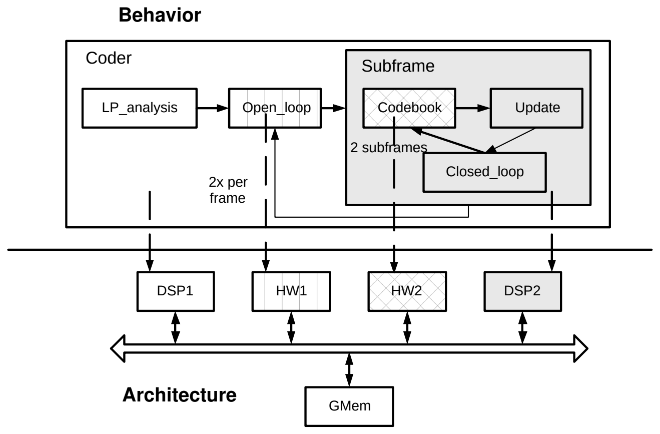


Fig. 4. System model and PE-behavior mapping of vocoder project

The first usage of our algorithm is to compute the required local memory sizes of PEs, when designers want to map all the variables to local memories in order to avoid the global memory. Table I shows the computed required local memory sizes. Raws *opti1*/*opti2* denote the required memory sizes computed by our algorithm. *Opti1* is computed when both *DSP1* and *DSP2* adopts non-preemptive OS, while *opti2* is computed when both *DSP1* and *DSP2* adopts preemptive OS. Raw *n-opti* denotes the required memory sizes computed by just adding variable sizes up, no matter what OS that *DSP1* and *DSP2* adopt. Except the memory size computation, the other parts of variable mapping algorithms of *opti1*, *opti2*, and *n-opti* are the same, which follow the proposed algorithm in section VI. For *DSP1* and *DSP2*, the memory sizes displayed in *opti1* are 24.5% and 22.6% smaller than the sizes in *n-opti*. The memory sizes in *opti2* are 13.2% and 9.6% smaller than the sizes in *n-opti*.

The second usage of our algorithm is to minimize the required global memory size and system traffic when the local memory sizes of PEs are pre-defined. Table II displays 5 different sets of predefined local memory sizes. The local memory size of *HW1* is always 800 bytes and the size of *HW2* is always 2400 bytes. We select the memory sizes of *DSP1* and *DSP2* to close to the computed required local memory sizes in Table I. The meanings of *opt(1), opt(2)* and *n_opti* are the same as in Table I. The result shows that our algorithm not only reduces required global memory size, but also reduces the traffic on the system buses. For example, in case 1, because we analyze the variable lifetime during memory size computation, the generated traffic on the system bus by *opt1* is 75% smaller than the generated traffic by *n-opti*.

VIII. Conclusion

This paper proposes a novel memory size model for variable mapping problem in system level design. By using this model, the variable memory mapping algorithms can reduce the required memory sizes in the system architecture and decrease the amount of bus traffic between PEs, such as shown in the experimental result. This is because the model allows variables with disjoint life-time and with different sizes to share the same memory region.

The direction of future work is to improve the current variable mapping algorithm based on the proposed memory size model.

References

[1] VCC[online]. Available: http://www.cadence.com/products/vcc.html.

[2] F. Balasa, F. Catthoor, and H. D. Man. Background memory area estimation for multi-dimensional signal processing systems. *IEEE Trans. on VLSI Systems*, June 1995.

[3] L. Cai and D. Gajski. Variable Mapping of System Level Design. Technical Report CECS-TR-03-03, Nov. 2002.

[4] D. Gajski, N. Dutt, S. Lin, and A. Wu. *High Level Synthesis: Introduction to Chip and System Design*. Kluwer Academic Publishers, 1992.

[5] A. Gerstlauer, S. Zhao, and D. Gajski. Design of a GSM Vocoder using SpecC Methodology. Technical Report ICS-TR-99-11, Feb. 1999.

[6] S. Meftali, F. Gharsalli, F. Rousseau, and A. Jerraya. An optimal memory allocation for apllocation-specific multiprocessor system-on-chip. In *ISSS*, 2001.

[7] P. Panda and A. N. N. Dutt. *Memory Issues in Embedded System-on-chip: Optimization and exploration*. Kluwer Academic Publishers, 1999.

[8] S. Prakash and A. Parker. Synthesis of application-specific multiprocessor systems including memory components. *IEEE Transactions on VLSI Signal Processing*, 1994.

[9] R. Szymanek and K. Kuchcinski. Design space exploration in system level synthesis under memory constraints. In *Euromicro 25*, September 1999.

[10] I. Verbauwhede, C. Scheers, and J. Rabaey. Memory estimation for high level synthesis. In *DAC*, 1994.

[11] Y. Zhao and S. Malik. Exact memory size estimation for array computation without loop unrolling. In *DAC*, 1999.