Memory Subsystem Simulation in Software TLM/T Models

Eric Cheung, Harry Hsieh
University of California Riverside
Riverside, California 92521
{chuncheung,harry}@cs.ucr.edu

Felice Balarin
Cadence Design Systems
San Jose, California 95134
felice@cadence.com

Abstract— Design of Multiprocessor System-on-a-Chips requires efficient and accurate simulation of every component. Since the memory subsystem accounts for up to 50% of the performance and energy expenditures, it has to be considered in system-level design space exploration. In this paper, we present a novel technique to simulate memory accesses in software TLM/T models. We use a compiler to automatically expose all memory accesses in software and annotate them onto efficient TLM/T models. A reverse address map provides target memory addresses for accurate cache and memory simulation. Simulating at more than 10MHz, our models allow realistic architectural design space explorations on memory subsystems. We demonstrate our approach with a design exploration case study of an industrial-strength MPEG-2 decoder.

I. Introduction

Multiprocessor System-on-a-Chips (MPSoC) are highly configurable. The efficiency with which the designers can explore system-level design space directly impacts the quality of the implementation. Architectural decisions, such as distribution of available die area for caches and memories, have significant consequences in terms of total performance and energy. An efficient simulation for system-level *design space explorations* (DSE) of MPSoC architectures is crucial for a successful implementation.

The most important requirement of the simulation is the ability to provide a fast and accurate feedback on performance and energy characteristics of a specific implementation. The memory subsystem, which accounts for up to 50% of performance and energy expenses, is one of the most important architectural decisions to be explored. If the performance or energy consumption of an implementation is not satisfactory, designers need to be able to quickly try out other implementations. The simulation must satisfy the following requirements in order to be efficient in system-level DSE:

- Procedure to generate simulation models for an implementation must be automatic.
- Architectural components, including the memory subsystem, must be accurately evaluated for their impacts on the implementation.
- Simulation speed must be sufficiently fast to allow quick feedback.

None of the current techniques for cache simulation in MPSoC satisfies all three requirements. In behavioral-level simulation [1, 5, 8, 12, 13, 15, 21], memory accesses that are generated when software is compiled are not visible in a high-level programs. In [6, 7, 9, 14, 17, 20], memory subsystem is not accurate simulated. In instruction-by-instruction [2, 3, 10, 16, 19, 22], simulation is inefficient and does not provide sufficient speed for system-level DSE.

In this paper, we introduce a novel technique to simulate MPSoC implementations with caches and memories that satisfies all three requirements. 1. We use a compiler to automatically generate TLM/T models of software that exposes all memory accesses. 2. A reverse address map generates target memory addresses for accurate memory subsystem simulation. 3. simulation runs natively in the host machine in transaction-level for efficient simulation. The main contributions of this paper include a practical use of a compiler-annotation technique, and a novel technique to accurately simulate memory subsystems in software TLM/T models.

The rest of this paper is organized as follow. In Section II, we describe some related works. In Section III, we describe software TLM/T models in SystemC and their limitations. In Section IV, we discuss how we generate software TLM/T models automatically and simulate caches and memories accurately. In Section V, we conduct a set of experiments to evaluate our memory subsystem simulation. We conclude the paper in Section VI.

II. Related Work

A number of behavioral-level simulation platforms are developed to make simulation more efficient. Metropolis [1], Simulink [8], Sesame [15], MESH [13] and SystemC *transition-level modeling* (TLM) [12] provide behavioral-level simulation for MPSoC. Simulation runs faster by running software natively on the machine machine. However, the simulation models often hide memory access details such that the memory subsystem is not accurately considered. Memory accesses in the simulation models are omitted, annotated manually or based on execution trace from ISS. Manual annotation is not automatic, not accurate, and does not consider implicit accesses generated in compilation. And execution trace annotation, which requires an execution in ISS for every cache configuration or design modification, is inefficient.

Cache exploration has been a common research topic for single-processor systems. Cache exploration is usually based on simulation with a pre-generated memory address trace [6]. However, simulation using traces does not work for MPSoC because there are multiple interacting programs. Address traces change with a different implementation of the memory subsystem, especially for implementations with caches that are accessible by more than one processors (i.e. coherent caches).

Analytical and statistical methods have been used to estimate cache performance in MPSoC. A histogram based estimation [14] provides statistical estimations on memory subsystems. However, their estimations, with up to more than 300% error, are not reliable for DSE. Analytical method in [20] only works on parallel algorithms with regular data access patterns and does not work on regular programs in general. Multivariate models [7] use statistics to estimate memory subsystem performance based on the results from a number of simulations. But such method has not been shown accurate. Predefined models [9] use a fixed hit/miss rate for a specific cache configura-

```
#include "systemc.h"
SC_MODULE tlm_program                  // declare the software TLM/T module
{
    sc_port<mem_if> mem_port;          // memory port
    SC_CTOR (tlm_program) {            // SystemC constructor for tlm_program
        SC_THREAD (main);              // register main() with SystemC kernel
    }
    void main () {                          // program main function
        int i, j;
        wait (sc_time(18, SC_NS));          // estimation
        for (i = 1; i < 10; i++) {
            for (var[i] = 1, j = 0; j < i; j++) {
                var[i] *= i;
                wait (sc_time(15, SC_NS));  // estimation
            }
            wait (sc_time(24, SC_NS));      // estimation
            mem_port->write(0x400,4,&var[i]); // write 4 byte data to address 0x400
        }
        wait (m_never_notify);              // program stops here
    }
    int var[10];
    sc_event m_never_notify;
};
```

Fig. 1. Software TLM/T model

```
...
unsigned var[10];
int LC2[1] = {(int)var};                          // literal
void main () {
    int a2, a3, a4, a5, a6, a7;                   // registers
    a4 = 1;
    wait(sc_time(3,SC_NS));
    mem_port->read((unsigned)LC2,4,NULL);         // read literal
    a5 = *(int *)((unsigned)LC2);                 //    from memory subsystem
    wait(sc_time(3,SC_NS));
  L2: mem_port->write(a5,4,NULL);                 // write to
    *(int *)a5 = 1;                               //    array element var[i]
    a3 = 1;
    a2 = 0;
    wait(sc_time(9,SC_NS));
  L3: a2 = a2 + 1;
    a3 = a3 * a4;
    if(a2 < a4) {wait(sc_time(15,SC_NS)); goto L3;}
    wait(sc_time(9,SC_NS));
    mem_port->write(a5,4,NULL);                   // write to
    *(int *)a5 = a3;                              //    array element var[i]
    wait(sc_time(3,SC_NS));
    mem_port->write(0x400,4,&a5);                 // explicit write to 0x400
    a4 = a4 + 1;
    a5 = a5 + 4;
    if(a4 != 10) {wait(sc_time(18,SC_NS)); goto L2;}
    wait(sc_time(12,SC_NS))
    block(never_notify_event);
}
...
```

Fig. 2. Generated Software TLM/T Model

tion. However, it does not account for the data access patterns of the programs. Cache simulator is currently the only reliable way to obtain accurate information about a particular memory subsystem implementation. Simulation that does not based on cache simulation with accurate memory address trace tends to be inaccurate.

Instruction-by-instruction simulation is the traditional way to evaluate accurately the memory subsystem in MPSoC. Memory address calculations and memory accesses are all explicit by emulating the instructions run on the target processors. In architectural [3, 10] and *instruction set simulation* (ISS) level [2, 19] simulation, target instructions are individually decoded and executed. Improved simulation techniques using interpretation [22] or compiled instruction [16] also provide instruction-by-instruction simulation of software and allow memory subsystem simulation. However, these techniques require simulation of many architectural details and are very slow and inefficient for system-level DSE.

III. Software TLM Model

A software TLM/T model is a behavioral-level description of a program annotated with execution time. The model implements *SC_MODULE* in SystemC and has one *SC_THREAD* to execute the program. Time to execute the program in the target processor is annotated onto the software TLM/T model using *wait()*. *wait()* adds timing delays to the model and synchronizes with global SystemC time. An example of a software TLM/T model is shown in Figure 1. *main()* executes once starting at the beginning of the simulation. The program calculates a set of numbers and explicitly writes to a memory address with the TLM interface. The behavior of the program is compiled into the simulation and executes directly on the host machine. TLM/T models provide more than two orders of magnitude speedup over ISS.

Simulation of software TLM/T models directly uses the host memory allocated to the instances of the models. Since the program is compiled and directly executes on the host machine, the memory required to store the variables resides on the host memory. In the example, the array *var* and the program stack for *i* and *j* store on the host memory.

Memory accesses of a program that are not explicit in the TLM/T model are not simulated. Memory accesses can be implicit or explicit in a program. Explicit accesses are specified in the TLM/T model using *read()* and *write()*. However, implicit accesses, such as reading or writing a de-referenced address, an array, a volatile variable, or a program stack due to lack of registers, are not specified in the TLM/T model.

Although some estimations allow performance of a particular private cache to be included based on a simulation trace from ISS, they are not flexible to allow memory subsystem DSE and require estimations to be generated from ISS every time the memory subsystem changes.

IV. Cache and Memory Simulation

Cache and memory configurations are very important in MPSoC. Different programs have different memory requirements and cache access characteristics. It is important to configure the caches and memories such that the programs run efficiently under the constraints in performance, area and power. Analyzing cache and memory configurations for MPSoC is difficult because programs on different processors complexly interact. Memory address traces differ dramatically with a minor change in the implementation. Therefore, the address traces generated by ISS cannot be reused for DSE. To efficiently evaluate different cache and memory configurations and explore the memory subsystem design space, our generated software TLM/T models provide dynamic target memory addresses within simulation such that caches and memories are accurately simulated.

A. Generated Software TLM/T Model

Since memory accesses can be explicit or implicit, a compiler is utilized to expose *all* memory accesses in the programs. We develop a SystemC backend for the compiler to generate TLM/T models in SystemC for software [4]. C programs are run through the compiler. Explicit memory addresses are specified to be unmodified. The program is portable that it does not assume the memory addresses of its data or the relationships between the addresses of different data. The compiler first converts a program into an optimized low level *intermediate representation* (IR). Then the backend of the compiler generates SystemC TLM/T codes based on performance and energy estimations in the IR [23]. Procedures to generate software

TLM/T models with annotations are completely automatic and transparent to the designers.

Simulating caches and memories is possible because all memory accesses are accurately visible in the IR. Explicit accesses are specified by the designers. Implicit accesses generated by the compiler are converted into load and store instructions, which represent accesses to the memory. The number and locations of the implicit accesses depend on the optimization levels in the compiler. Therefore, as shown in Figure 2, the memory accesses are annotated onto the TLM/T model using *read()* and *write()* by the compiler backend. Note that the implicit accesses only simulate the accesses of the addresses but do not actually read or write (specified by *NULL*) because these accesses are internal to the programs. Without using the compiler to analyze the program, implicit memory accesses cannot be exposed accurately. Instruction accesses, on the other hand, can be annotated based on [17].

With *read()* and *write()* annotated onto the TLM/T models, memory accesses can be simulated for performance and energy estimations of different cache and memory configurations. To our knowledge, no other system-level design platform is able to simulate caches alongside behavioral-level models (annotated or otherwise) with the same level of accuracy and efficiency. We consider this a great deficiency when the memory subsystem typically accounts for at least 20% of the energy and performance degradation. We will show that it is possible to simulate caches in software TLM/T models with a small loss in simulation speed.

B. Reverse Address Mapping

In addition to exposing all memory accesses in the TLM/T models, we need to determine the target memory addresses for the memory accesses to simulate the memory subsystem accurately. Data in the implementation is mapped to specific target memory addresses. The target memory addresses determine the runtime characteristics of the caches and memories. The caches behave differently with different addresses.

In general, it is impossible to determine target memory addresses for loads and stores in compile time. Statically determining target memory addresses for all accesses in compile time is a NP-hard problem, especially when pointer manipulations are involved.

B.1 Target Memory Address

The memory map of the implementation is provided to the TLM/T models to map the data into their target memory addresses. For a target executable, a linker maps data to specific memory addresses based on a linker script. The linker script specifies the memory address for each section: *bss*, *literal*, *stack*, *heap*, etc. Data can be placed to specific sections using compiler directives in the programs. To provide accurate target memory addresses for simulation, such memory map should be provided.

Without a provided memory map, which is common for early system-level DSE, one memory map can be generated. The generated memory map maintains the spatial and temporal localities of the memory accesses. Memory subsystem DSE can be done with minor loss in accuracy. If the exploration result is satisfactory, the generated memory map can be used to construct a linker script that keeps the same memory map as used in the explorations.

B.2 Address Lookup

A simulation time lookup procedure is used to determine the target memory address for each implicit memory access. For each load or store in the compiler-generated codes, the TLM/T model reads or writes to an address in the host memory where the data allocates. The host memory address indicates the data that is accessed. Therefore, using the host memory address, we can use a *reverse address mapping* procedure to determine the target memory addresses during the simulation. A similar "Address Recovery" technique [5] has been proposed to forward memory address from the target memory space to the host memory space. However, the technique does not support any pointer manipulations, which are very common in multimedia applications. Our reverse address mapping allows pointer manipulations on the host memory addresses and is able to map addresses from pointer manipulations to their correct corresponding target memory addresses.

Reverse memory map ($mmap$) is a set of tuples with three fields: *host memory address* $\in H$, *length* $\in N$ and *target memory address* $\in T$. H is the set of host memory addresses that are accessible directly from the TLM/T models. N is a natural number that represents the size of the data. T is the set of target memory addresses of the data in the implementation.

Property 1. Uniqueness of host memory addresses:

$$\nexists (h_1, l_1, t_1), (h_2, l_2, t_2) \in mmap$$
$$\text{such that } (h_1 = h_2) \wedge (t_1 \neq t_2)$$

Property 2. Non-overlapping addresses:

$$\nexists (h_1, l_1, t_1), (h_2, l_2, t_2) \in mmap$$
$$\text{such that } (h_1 < h_2) \wedge (h_1 + l_1 > h_2)$$

Since SystemC simulation is simulated in one host memory space, each data in the TLM/T models allocates a specific address in the host memory. No host memory addresses of two data are the same or overlapped. The data in software, when in scope, has unique host memory addresses. Each instance of a TLM/T model allocates a different memory address. Therefore, when a data goes into scope, $register()$ is invoked to add a new entry into $mmap$ (Algorithm 1). When the data goes out of scope, $unregister()$ is invoked to remove the entry from $mmap$. The entries in $mmap$ always obey Property 1 and 2. $mmap$ also applies to stacks and heaps as they are simply big chunks of memory allocated to the programs.

Algorithm 1: Register Reverse Address Map

```
1 $mmap = \varnothing$
2 $register(h \in H, l \in N, t \in T)$
3     check against lemma 1 and 2
4     $mmap = mmap \cup \{(h, l, t)\}$
5 $unregister(h \in H)$
6     $mmap = mmap / \{(h, *, *)\}$
```

On the other hand, target memory addresses are not unique and can overlap. Unlike host memory which is in one memory space, programs in different processors can have asymmetric memory views and access different data with the same memory address.

Property 3. In-order memory addresses:

$$(\forall (h, l, t) \in mmap) \wedge (\forall i < l)$$
$$\rightarrow lookup(h + i) = t + i$$

Memory for a data (including an array or a structure) is always contiguous. We use data-types for data of the same size. Therefore, in the program point of view, if the host memory address h with size l maps to the target memory address t, an offset added to the host memory address $(h + i)$, as long as $i < l$, maps to the same offset of the target memory address $(t + i)$. This property (Property 3) allows pointers (data addresses) to be used to lookup their corresponding target memory addresses after pointer manipulations.

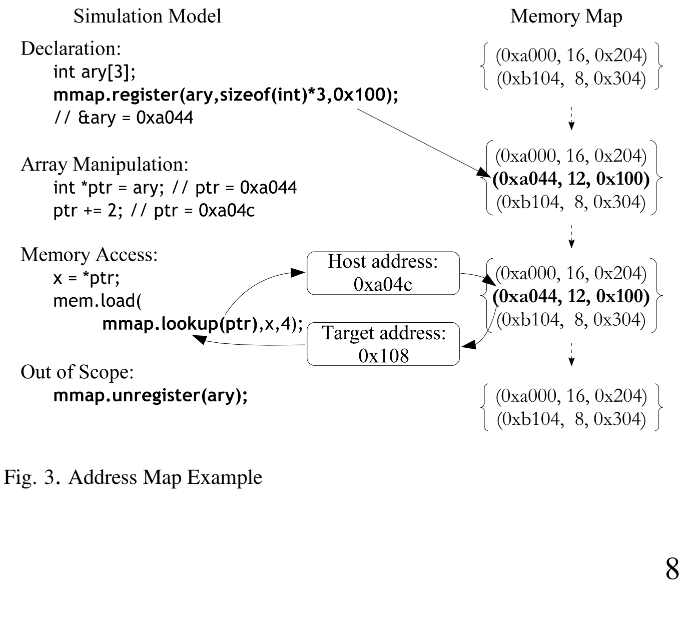
Algorithm 2: Lookup Reverse Address Map

```
1 sort $mmap$ with increasing order of h
2 $lookup(s \in H)$
3     use binary search for last $(h, l, t) \in mmap$ such that $h < s$
4     if $h + l < s$ then
5         return $ERROR$
6     else
7         return $(t + (s - h))$
8     end
```

The reverse address map allows the target memory addresses to be generated dynamically during simulation regardless of pointer manipulations in the programs. The function $lookup : H \rightarrow T$ (Algorithm 2) is used to determine the target memory addresses based on the host memory addresses. The complexity of such lookup is $O(lnN)$, which N is the number of entries in $mmap$. Hence, such lookup is scalable for big designs. During the simulation, pointer manipulations are applied directly on the host memory addresses, and the resulting addresses are used to lookup the target memory addresses. A valid pointer manipulation always ends up in a memory address of a data that is properly declared in the host memory and registered in the reverse address map. Unless the programs try to access memory that is not properly declared and registered, we can handle pointer manipulations without difficulty.

B.3 Address Map Example

An example of a reverse address mapping is shown in Figure 3. First, when a data is allocated, *register()* adds an entry to $mmap$. In the example, the array ary of total size 12 bytes resides in the host memory starting at $0\text{x}a044$. The host memory address of the array may be different every time the simulation runs, however it always map to the same target memory address. The target memory address for the array ary is $0\text{x}100$, provided by a memory map. In *register()*, the entry $(0\text{x}a044, 12, 0\text{x}100)$ is added to $mmap$. Second, the address ptr is calculated by the pointer manipulations. As a result ptr points to $0\text{x}a04c$ in the host memory, the third element in ary. Third, ptr is then used in a load instruction, where it is used in *lookup()* for the target memory address. An offset of 8 is



```
int ary[3];
mmap.register(ary,sizeof(int)*3,0x100);
// &ary = 0xa044

int *ptr = ary; // ptr = 0xa044
ptr += 2; // ptr = 0xa04c

x = *ptr;
mem.load(
      mmap.lookup(ptr),x,4);

mmap.unregister(ary);
```

Fig. 3. Address Map Example

then applied to the address 0x100, which results in the target memory address 0x108. The address corresponds to the third element of ary in the target memory. In general, for a legal load or store, the host memory address must point to a memory space that is properly declared, hence the address can always map to a target memory address. Last, when ary goes out of scope, the entry in $mmap$ is deleted with *unregister()*. Entries in $mmap$ are dynamically added and deleted when variables and arrays go in and out of scope during simulation. This dynamic operation is necessary since the same host memory space can be reused by the simulator for multiple data where their life times do not overlap

C. Memory Subsystem Modeling

A transaction-level memory subsystem modeling is used to simulate caches and memories in the target MPSoC system. Caches and memories are modeled in SystemC TLM/T [22] to provide fast efficient simulation. The cache model is highly configurable. It can be configured with any combination of valid cache size, associativity, block size and replacement policy. Cache coherency can also be implemented to match the target memory subsystem implementation.

We integrate CACTI library [18] in our cache and memory models to accurately estimate the access times and the energy consumptions of caches and memories with different configurations. CACTI is an integrated model of cache access time, area, aspect ratio, and power based on the capability of accurately calculated wire capacitances and resistances of address and data routings.

V. Experiments & Results

In this section, we present a set of experiments for MPSoC memory subsystem simulation evaluations and design space explorations. We implement our SystemC backend in LLVM compiler infrastructure [11], which utilizes the GCC C/C++ frontend and a set of common global and inter-procedural optimizations. The backend creates software TLM/T models with performance and energy estimations.

We use Tensilica's Xtensa LX2 processors [19] in our MPSoC architecture. We use the typical configuration generated using Xtensa Processor Generator without instruction extension. All experiments run on a Pentium 4 3.3GHz machine with 1GB of memory. Xtensa tools run on Linux and SystemC simulation is compiled using GCC and OSCI SystemC library. All timing, energy and area estimations are based on 0.12u technology [24].

A. Simulation Time Evaluation

We demonstrate our memory subsystem simulation and design space explorations using an industrial-strength MPEG-2 decoder design (Figure 4). The design is written in Kahn Process Network. It is composed of nine programs that are partitioned and mapped into processors. MPEG-2 decoder is a stream-based application that is suitable to be implemented in MPSoC. Global accesses from the programs using read and write calls are explicitly defined in the design.

The target implementation architecture is shown in Figure 5. Multiple processor subsystems, each with a processor, a private instruction memory and a private data cache, are connected using a crossbar. For energy efficiency, cache coherency is not used because it generally keeps the caches and the crossbar busy. Instead shared objects are allocated in memory and are not cached in the private cache. A shared buffer is used to increase efficiency of the memory by caching the memory accesses.

Memory accesses from each program are transactionally correct. Because memory accesses are exactly represented in the software TLM/T models and the reverse address mapping maps the accesses correctly into target memory addresses with a provided memory map, memory accesses from each program have no error. Since the accuracy of the overall simulation also depends on other factors, such as the accuracy of the timing estimation of the software models and the modeling of interconnects, overall accuracy results are not representative here. In our experiments, we obtain an overall simulation performance within 10% from the results of ISS, including the inaccuracies from the software timing estimation and interconnect modeling.

	Sim. Time	Speed·Proc	Speedup
ISS	2 hrs	100 kHz	1
Interpretation	7 min	1.5 MHz	15
TLM/T	19 sec	37 Mhz	370
TLM/T & Cache	41 sec	16 MHz	160

TABLE I
SIMULATION SPEED COMPARISON

Our cache and memory simulation in software TLM/T models provides efficient simulation for memory subsystem DSE. Table I shows the simulation speeds of TLM/T models with and without memory subsystem simulation. We compare the simulation speeds to the results of *ISS* and *interpretation*, two instruction-by-instruction simulation techniques that allow memory subsystem simulation reported in literatures [19, 22]. Although simulating caches and memories reduces the simulation speed because of the overhead for additional timing delay calls, reverse address lookups and memory subsystem simulation, our simulation still simulates in excess of 10MHz. Our TLM/T models with memory subsystem simulation simulate two orders of magnitude faster than ISS and one order of magnitude faster than interpretation. Such simulation speed allows complete implementation developments and DSE with realistic inputs. Hence the designers can choose the best implementation based on performance and energy characteristics.

B. Design Point Analysis

The strength of our simulation is the ability to explore various implementations efficiently for MPSoC. Here we explore the number of processors and memory subsystem configurations.

B.1 Processor and Cache Allocation

In MPSoC, the die area is shared by both processors and caches. It is the designers' choice to allocate area for more processors or for bigger caches. This problem is previous ignored because DSE platforms based on ISS are too slow and behavioral simulations do not support memory subsystem simulation.

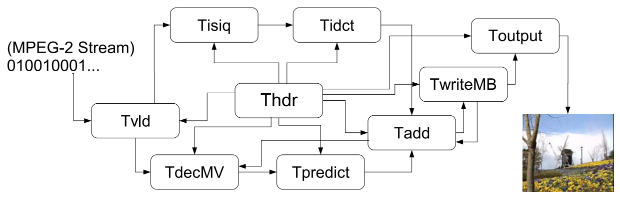


Fig. 4. MPEG-2 Decoder Design

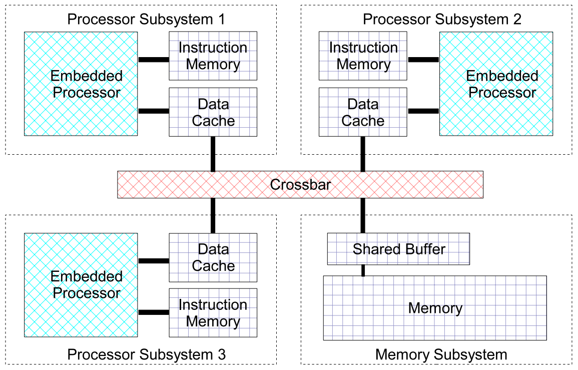


Fig. 5. MPSoC Architecture

	Mapping
1 proc	{Tvld,Tisiq,Thdr,Tidct,TdecMV, Tpredict,Tadd,TwriteMB,Toutput}
2 proc	{Tvld,Thdr,Tidct,TdecMV,Tadd}, {Tisiq,Tpredict,TwriteMB,Toutput}
3 proc	{Tvld,Thdr,Tidct,TdecMV}, {Tisiq,Tadd,TwriteMB},{Tpredict,Toutput}
4 proc	{Tvld,Thdr,TdecMV,TwriteMB},{Tisiq,Tidct}, {Tadd,Toutput},{Tpredict}
5 proc	{Thdr,TdecMV,Tadd},{Tisiq,TwriteMB}, {Tvld,Toutput},{Tpredict},{Tidct}
6 proc	{Tvld,Thdr,TdecMV},{Tisiq,Toutput}, {TwriteMB},{Tadd},{Tidct},{Tpredict}

TABLE II
PROGRAM MAPPINGS

We use one to six processors in our target implementation architecture and homogeneous configurations for all private data caches. Program mapping is based on load balancing and is shown in Table II. Total area for all processors and caches is limited to 5 mm^2.

Figure 6 shows the result of the processor/cache exploration. Power consumption increases with the number of processors because the same area used for processors consumes more power than caches. However, if too little area is devoted to computation (i.e. 1 and 2 processors), the execution time lengthens and it results in higher energy consumption. From one to five processors, performance increases with the number of processors. With six processors, however, performance gets worse as the caches become too small and memory accesses regularly go to the main memory. Therefore, we get

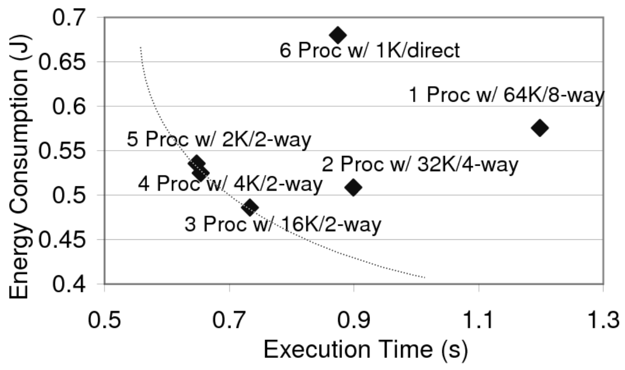


Fig. 6. Processor and Cache Allocation

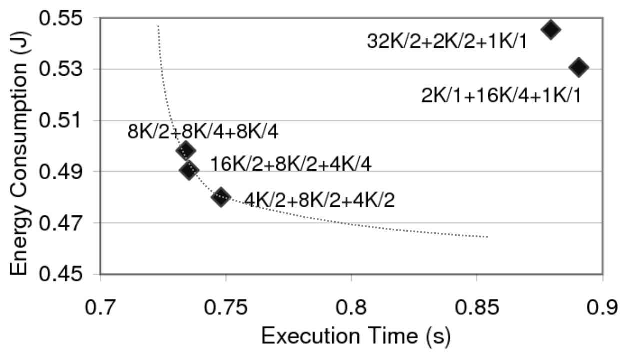


Fig. 7. Cache Configuration

the best performance with five processors and the best energy consumption with three processors.

B.2 Cache Configuration

Memory subsystem design space in an MPSoC implementation is very large. Each cache can be configured with more than a hundred configurations. In our experiment, we use three processors in our target architecture with three private data caches. Total area for all processors and caches is limited to 5 mm^2.

Figure 7 shows the result of the heterogeneous cache exploration. In our experiment, a bad cache configuration can take 20% more execution time and energy compared to a good configuration. Good configurations are heterogeneous. Each private data cache is configured according to the memory access characteristics of the programs that are mapped to the processors. Hence it is important to explore memory subsystem design space to achieve a good implementation. We get the best performance with 8K/2, 8K/4, 8K/4 for the three caches respectively and the best energy consumption with 2K/2, 4K/2, 16K/2 respectively.

As shown in our analysis, the memory subsystem has a huge impact on the performance and energy characteristics of an implementation. Designers need to be able to explore different cache and memory configurations. It requires fast and automatic system-level simulation models that allow memory subsystem simulation. Our software TLM/T models provide accurate simulation for memory accesses and allow designers to explore system-level design space like never before.

VI. Conclusion

In this paper, we introduce a technique to simulate memory subsystems in software TLM/T models. We present an automatic annotation for getting quantitative memory accesses into efficient software TLM/T models. And we present a reverse address map to make it possible to accurately simulate caches alongside TLM/T models. Our technique is able to efficiently explore complex architectural issues such as distribution of available die area for caches and memories. Our approach opens up design space explorations that is not possible before.

References

[1] F. Balarin, Y. Watanabe, et al. Metropolis: An integrated electronic system design environment. *Computer*, 36(4):45–52, 2003.

[2] L. Benini, D. Bertozzi, et al. Mparm: Exploring the multiprocessor soc design space with systemc. *J. VLSI Signal Process. Syst.*, 41(2):169–182, 2005.

[3] D. Brooks, V. Tiwari, et al. Wattch: a framework for architectural-level power analysis and optimizations. In *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, pages 83–94, New York, NY, USA, 2000. ACM.

[4] E. Cheung, H. Hsieh, et al. Framework for fast and accurate performance simulation of multiprocessor systems, Nov. 2007.

[5] L. Gao, K. Karuri, et al. Multiprocessor performance estimation using hybrid simulation. In *DAC '08: Proceedings of the 45th annual conference on Design automation*, pages 325–330, New York, NY, USA, 2008. ACM.

[6] A. Ghosh and T. Givargis. Analytical design space exploration of caches for embedded systems. In *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, page 10650, Washington, DC, USA, 2003. IEEE Computer Society.

[7] I. Gluhovsky and B. O'Krafka. Comprehensive multiprocessor cache miss rate generation using multivariate models. *ACM Trans. Comput. Syst.*, 23(2):111–145, 2005.

[8] K. Huang, S. il Han, et al. Simulink-based mpsoc design flow: case study of motion-jpeg and h.264. In *DAC '07: Proceedings of the 44th annual conference on Design automation*, pages 39–42, New York, NY, USA, 2007. ACM Press.

[9] Y. Hwang, S. Abdi, et al. Cycle-approximate retargetable performance estimation at the transaction level. In *Design, Automation and Test in Europe, 2008. DATE '08*, pages 3–8, Munich, Germany,, Mar. 2008.

[10] S. Lee, S. Das, et al. Circuit-aware architectural simulation. In *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 305–310, New York, NY, USA, 2004. ACM.

[11] LLVM Compiler Infrastructure Project. http://www.llvm.org.

[12] Open SystemC Initiative. http://www.systemc.org.

[13] J. M. Paul, A. Bobrek, et al. Schedulers as model-based design elements in programmable heterogeneous multiprocessors. In *DAC '03: Proceedings of the 40th conference on Design automation*, pages 408–411, New York, NY, USA, 2003. ACM.

[14] J. J. Pieper, A. Mellan, et al. High level cache simulation for heterogeneous multiprocessors. In *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 287–292, New York, NY, USA, 2004. ACM.

[15] S. Polstra. A systematic approach to exploring embedded system architectures at multiple abstraction levels. *IEEE Transations on Computers*, 55(2):99–112, 2006. Member-Andy D. Pimentel and Student Member-Cagkan Erbas.

[16] M. Reshadi, P. Mishra, et al. Instruction set compiled simulation: a technique for fast and flexible instruction set simulation. In *DAC '03: Proceedings of the 40th conference on Design automation*, pages 758–763, New York, NY, USA, 2003. ACM.

[17] J. Schnerr, O. Bringmann, et al. High-performance timing simulation of embedded software. In *DAC '08: Proceedings of the 45th annual conference on Design automation*, pages 290–295, New York, NY, USA, 2008. ACM.

[18] P. Shivakumar and N. P. Jouppi. Cacti 3.0: An integrated cache timing, power, and area model.

[19] Tensilica Xtensa processors. http://www.tensilica.com.

[20] J. Tsai and A. Agarwal. Analyzing multiprocessor cache behavior through data reference modeling. In *SIGMETRICS '93: Proceedings of the 1993 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 236–247, New York, NY, USA, 1993. ACM.

[21] P. van der Wolf, E. de Kock, et al. Design and programming of embedded multiprocessors: an interface-centric approach. In *CODES+ISSS '04: Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 206–217, New York, NY, USA, 2004. ACM Press.

[22] E. Viaud, F. Pêcheux, et al. An efficient tlm/t modeling and simulation environment based on conservative parallel discrete event principles. In *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, pages 94–99, Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[23] T. Simunić, L. Benini, et al. Cycle-accurate simulation of energy consumption in embedded systems. In *DAC '99: Proceedings of the 36th ACM/IEEE conference on Design automation*, pages 867–872, New York, NY, USA, 1999. ACM.

[24] T. Wolf and J. S. Turner. Design issues for high-performance active routers. *IEEE Journal on Selected Areas in Communications*, 19(3):404–409, Mar. 2001.