IPSJ Transactions on System LSI Design Methodology Vol.8 38–50 (Feb. 2015)

[DOI: 10.2197/ipsjtsldm.8.38]

Regular Paper

Efficient Design Exploration Framework of SW/HW Systems Based on Tightly-coupled Thread Model

Arif Ullah Khan[1,a)] Tsuyoshi Isshiki[2,b)] Dongju Li[2,c)] Hiroaki Kunieda[2,d)]

Received: June 2, 2014, Revised: August 22, 2014,
Accepted: October 15, 2014, Released: February 12, 2015

Abstract: In order to meet the increased computational requirement of today's consumer portable devices, heterogeneous multiprocessor system-on-chip (MPSoC) architectures have become widespread. These MPSoCs include not only multiple processors but also multiple dedicated hardware accelerators. Due to the increase complexity of the MPSoC, fast and accurate design space exploration (DSE) for best system performance at early stage of the design process is desired. Any DSE solution is desired to provide best system partitioning scheme for best performance with efficient area utilization. In this paper we propose a design space exploration framework for heterogeneous MPSoC based on tightly-coupled thread (TCT) parallel programing model which can handles system partition exploration and HW synthesis exploration. The proposed framework drastically reduces the exponential size design space into near-linear size by utilizing the accurate HW timing models as the indicator for system bottleneck and guiding the enumeration process of HW version combinations. Experimental results shows the accuracy of the proposed method with an average estimation error of 1.38% for HW timing of each thread, and 2.80% estimation error for the system-level simulation, where the simulation speedup factor was in the order of 5,000 times. Currently the proposed framework partially depends on a high level synthesis (HLS) tool eXCite, but other HLS tools can be easily integrated into the proposed framework.

Keywords: MPSoC, TCT Model, system performance estimation, design space exploration

1. Introduction

The number of processing engines in consumer embedded and portable devices is increasing rapidly in order to meet the computational performance requirement of these devices. With the growing complexity of devices and the improvements in process technology heterogeneous MPSoC architectures, which combines embedded processors, multiple dedicated hardware accelerators and application specific instruction set processors (ASIPs) connected to a network-on-chip (NoC) to provide a complete integrated system [1], are widely adopted.

Now in the era of heterogeneous MPSoC, where an application will be divided into several software and hardware processing nodes, new design methodologies are required to address different aspects [2] of MPSoC design, which includes software programming model, communication and synchronization between various components, HW accelerator design, integration, system level performance estimation and design space exploration.

Lack of a viable new programming model has held back several multi-processor core architectures, like the PlayStation 3's Cell, from mainstream, because years later the application programmers have barely been able to comprehend how to write applications for it. If the tasks and processors in MPSoCs are not synchronized effectively, will leads to potentially inconsistent results during run-time. In a typical MPSoC programming, insertion of API calls for explicit inter-thread communication and synchronization is a time consuming, error prone and laborious job.

As we look to the future in MPSoC design, we witness two opposing forces that will have a profound impact. On one hand increase in software content is driving up data processing requirements of SoC platforms whilst on the other hand we see embedded processor clock frequencies being held in check as we struggle to stay within stringent power envelopes. By replacing computation intensive software part of the code with hardware implementation will not only enhance the performance [3] of the system but will also reduce overall power consumption of a system [4]. Manual design and integration of hardware accelerators is not straight forward and very time consuming especially when it involves multiple software and hardware processing nodes. Several High-Level Synthesis (HLS) tools have emerged in the past decade that attempt to address this challenge. By using HLS tools dedicated hardware accelerators can be generated from software code, written in high-level languages like 'C'. Integrating these hardware accelerators into a system is complex operation because, both on system level and signal level, the functionality and interfaces of those hardware accelerators differ from rest of the system.

One of other key aspect in design of a heterogeneous MPSoC is design space exploration, which is a process of exploring different system architecture for best performance under different

1 Graduate School of Information Science and Technology, Osaka University, Suita, Osaka 565–0871, Japan
2 Graduate School of Science and Engineering, Tokyo Institute of Technology, Meguro, Tokyo 152–8550, Japan
a) khan@ist.osaka-u.ac.jp
b) isshiki@vlsi.ce.titech.ac.jp
c) dongju@vlsi.ce.titech.ac.jp
d) kunieda@vlsi.ce.titech.ac.jp

© 2015 Information Processing Society of Japan

design constraints while preserving the overall functionality of the system. Design space exploration is one of the obstacles for a quick and efficient design of a heterogeneous MPSoC [5]. Now in the era of heterogeneous MPSoCs in which systems include multiple hardware and software processing nodes, design space exploration poses a new set of requirement on the design tools and methodologies. In past different methods are employed for design space exploration of MPSoCs. Traditionally detailed simulation of a system is used for design space exploration, which includes instruction accurate or cycle accurate instruction-set-simulator (ISS). Hardware-Software co-simulation is used when systems consists of hardware and software components. All these techniques are impractical to evaluate a large number of design choices due to prohibitively long simulation time. Use of statistical (analytical) workload models, which can evaluate large design space exploration efficiently but they need application specific methodologies for workload generation. There is a need for fast exploration of functionally similar design alternatives of a system with accurate estimations of performance, area and power, in order to adjust the MPSoC architecture at an early stage of the design process.

In this paper, we propose a new design space exploration framework for SW/HW systems which attempts to solve many of the design challenges stated above. First is the system partitioning methodology for quickly exploring various SW/HW configurations from the sequential application based on our *Tightly-Coupled Thread* programming model [6]. Second is the HW module generation exploration using HLS tool where a variety of synthesis options (resource allocation constraints) are given by our framework to the HLS tool in order to provide a wide selection of area-time tradeoffs for each HW components. These two dimensions of design space, system partition space and HW synthesis space, create a very large design space to explore. For this, we propose two orthogonal techniques to shorten the design space exploration time. First is the trace-driven workload simulation technology [7], [8] for fast evaluation of system performance of each design points. Second is the design space pruning technique which drastically reduces the design search space by the use of HW timing models extracted by the trace-driven workload model.

Rest of the paper is organized as follows: In Section 2 an overview of work which addresses various design aspects of an MPSoC will be presented. In Section 3 TCT MPSoC design space exploration framework will be explained which will be followed by area-time pareto-optimal design space evaluation in Section 4. Experimental results will be presented in Section 5, which will be followed by discussion and conclusion in Section 6.

2. Related Works

Designing of an MPSoC from conception to realization is a complex process which involves several steps. Extensive research has been performed to address the difficulties designers are facing when going through these steps. Due to increase in processing nodes in embedded systems, software design becomes more complex. In current practice of parallel programming with MPI [9] or OpenMP [10], the programmer should manually optimize the parallel code for each target processing node to ensure error and deadlock free communication and synchronization. This is a time consuming and laborious job. In Ref. [11] Paulin et al. proposes a MultiFlex, a multi-processor SoC programming environment, targeted at symmetrical multiprocessing with distributed memory architectures and distributed system object component (DSOC). In their programing environment a programmer is responsible to consider target architecture and related tuning of the application. In Ref. [12] N. Pazos et al presented a framework and a set of guidelines for transforming existing uni-processor software for multimedia applications into parallel models that can be used for design space exploration of MPSoC platforms. They proposed general guidelines for architecture independent MPSoC programing but they did not tackle the problem of communication and synchronization between various processing element and how to ensure deadlock free system.

Another key aspect in an MPSoC design is to replace software blocks with a dedicated hardware in order to meet real time performance goals of a system. In recent years advancement in HLS tools has made it easy to generate standalone hardware blocks but there is still lack of seamless methodologies for generating and integrating hardware accelerators in a system that can offload workload from software processing nodes and enhance performance of MPSoC running software written in a high level language like C. In Ref. [13] Y. Ando et al. presented design space exploration tool called System Builder. Their tool uses a HLS tool "eXCite" [14] for generation of RTL description of hardware processes. In their methodology, communication APIs between software processes and hardware bound processes are written explicitly with channel models. In contrast, our TCT model eliminates the burden of writing explicit communication and synchronization code.

In Ref. [15] Ian Page et al. proposes a hardware generation methodology which uses a concurrent programming language "occam" for expressing hardware accelerator and then generating hardware blocks. Their proposed methodology can be used to generate hardware accelerators but it did not address the problem of communication and synchronization between hardware and software. Other methodologies which uses C++ or C [16] emerges during past decade and all these methodologies focus on standalone generation of hardware accelerator. The hardware accelerator generation framework proposed by David Lau [17] generates a tightly coupled hardware accelerator and their framework also addresses the issues of hardware-software integration. Their proposed framework is based on single threaded software model and do not cover MPSoC. In Ref. [18] A. Samahi proposes a methodology for hardware-software communication synthesis for MPSoC and target at streaming applications. In their methodology the communication between hardware and software is defined by the designer.

In Ref. [19] Takuya et. al presented TECSCE, where software components are utilized for HW/SW co-design framework. While their framework provides automatic generation of communications and hardware components, it relies on the designer's manual descriptions of interface functions for communications between software components.

The TCT MPSoC programming model offers following advan-

© 2015 Information Processing Society of Japan

tages:
- The TCT MPSoC programing model allows the decomposition of sequential programs written in C into concurrent processes. System partitioning can be done directly on a reference C code. Thus very early in the design stage, system partitioning and performance estimation analysis can be performed. This will help the designer to quickly achieve the desired performance goals for the MPSoC implementation.
- TCT compiler automatically generates inter-thread communication and synchronization codes for each of the concurrent threads to create the system-level concurrent execution model. These communication/synchronization code insertion is achieved by the data dependence analysis of the entire application which frees the designer from dealing with time-consuming and error-prone task of designing the inter-thread communication codes. TCT compiler guarantees the identical behavior of the original sequential code and the parallelized code, automatically avoiding race conditions and deadlocks.

Early design space exploration is another key factor in cost and performance efficient design of a heterogeneous MPSoC. In Ref. [5] M. Gries presented a detail overview of different methods used for Design Space Exploration at system and micro-architecture levels. In Ref. [20] P. van Stralen has proposed a scenario based design space exploration methodology for MPSoCs. They argue that since the number of workload scenario is too large for exhaustive evaluation of all of the design alternatives. Their work is based on workload aware design space exploration by dynamically limiting the workload scenario. In Ref. [21] T. Wild propose TAPES framework based on trace simulation for fast design space exploration of complex SoC architecture. They rely on *manual specifications* of the traces for each SoC components, especially the timing behavior of CPUs and HW accelerators.

In contrast to the previous works on design space exploration mentioned above, our system performance estimation technique based on trace-driven workload simulation technology [7], the generation of application execution traces and accurate timing models for both SW and HW components are carried out automatically, and near cycle accurate performance estimation for each design instance can be achieved with significantly shorter amount of time compared to conventional system-level simulation technologies.

3. TCT MPSoC Design Space Exploration Framework

Authors have previously proposed an application development framework for an MPSoC based on the Tightly-Coupled Thread (TCT) model [6], [22]. These works were further extended in Ref. [8] where high-level synthesis tool was utilized to generate HW accelerators for several SW threads defined by the TCT programming model, and a unified timing modeling scheme was proposed for both SW threads and HW accelerators for fast and accurate performance estimation of mixed SW/HW components.

In this paper, we turn our focus on the design exploration methodology of mixed SW/HW components, where authors' previous works are integrated with new features to establish a complete tool chain to enable a thorough design exploration of various design decisions at each design phases. **Figure 1** shows the flow and various steps of the proposed design space exploration framework.

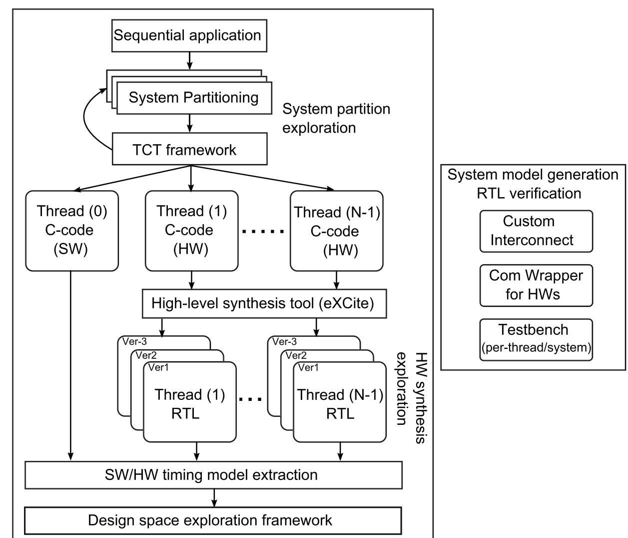


Fig. 1 TCT MPSoC design space exploration flow.

Below summarizes the contribution of this paper.

A System partition exploration: We utilize the "thread scope" semantics of the TCT model proposed in Ref. [6] to allow the designers to express a variety of different system partition instances on a single C-source. Here, we introduce a new effective macro-based coding style to control the thread scope insertions to reflect the system partitioning strategy (Section 3.1).

B System model generation: For each system partition instance, a complete set of component models are generated for RTL integration and functional verification. RTL generation using HLS tool eXCite and communication wrapper RTL generation was presented previously in Ref. [8]. Here, we additionally propose an automatic generation of customized partial crossbar interconnect which has high scalability with number of threads (Section 3.2.2). We have also refined the systematic test environment generation for the entire SW/HW systems as well as single-unit test models (Section 3.2.4).

C HW thread resource allocation exploration: For each partitioned behavioral C-codes for HW synthesis, we propose a new resource allocation constraint generation scheme for the HLS tool to control the area-time tradeoff exploration of the HW synthesis design space (Section 3.3).

D Area-time Pareto-optimal design point evaluation: Output of our proposed design exploration of mixed SW/HW system is the enumeration of Pareto-optimal design points. By utilizing our previous work on unified timing model for SW threads and HW accelerators [8], we propose a new design space pruning heuristic algorithm to obtain a near linear-order number of instances of Pareto-optimal design candidates from an exponential size exhaustive design space (Section 4.3). Finally, our previous trace-driven workload simu-

© 2015 Information Processing Society of Japan

lation technique [7], [8] is applied to evaluate the total number of clock cycles for each design candidates in order to obtain the area-time Pareto-optimal design curve generation from all system partitioning instances (Sections 5.2, 5.3).

3.1 System Partition Exploration

In TCT design space exploration framework a designer starts from a pure sequential C description of the application. The sequential software is analyzed by TCT tools and feedback about computation and communication intensive part of the code is provided by the tool. In the proposed framework the system partitioning is done directly on the C programs by the designer.

In TCT MPSoC design framework system partitioning is carried out by declaring *thread scopes*. A thread scope indicates a separate parallel process, which we simply refer to as threads, to be executed on a (separate) software processing node or as a dedicated hardware block. Thread scope's statement syntax is given as:

THREAD(name) { statements }

Any C statement (including function calls and nested thread-scopes) can be included inside the thread-scope region as long as the thread-scope forms a Single-Entry Single-Exit (SESE) region. Thread annotations can be inserted manually or through MAPS framework [23]. MAPS framework assists the designer with rich program analysis capabilities to emit thread annotated code semi-automatically.

For the demonstration of design space exploration, JPEG encoder [24] is used throughout this paper, because it is a real-world multimedia processing problem, its moderate complexity is well suited for the verification of the proposed design space exploration framework. JPEG encoder application has the following functional blocks:

- RGB: This function perform color space conversion from RGB to YC_bC_r space
- BUF: line-buffers for 8x8 block processing, down-sampling of chroma components
- BLK: 8x8 block processing controlling 4 Y-blocks and C_b/C_r blocks
- DCT: The discrete cosine transform function convert 8x8 pixel block of YC_bC_r to frequency domain
- Q: DCT coefficient quantization is performed for reducing the high frequency components
- E: Quantized coefficients are compressed using Huffman coding.
- OUT: Each byte output data is stored.

Figure 2 shows the core part of the JPEG encoder. In the TCT framework, *root* thread is defined as the code region which is not enclosed by any thread scope.

Except for the explicit thread scope in emit_byte, other thread scope insertions are specified through `TH_XX` macros whose definition is linked to the system partition strategy specified by THREAD_COUNT as below:

```
#define THREAD_COUNT spc // spc=3,4,5,7,8,12,16
#if ENABLE_XX // if (ENABLE_XX==1), then THREAD(XX) is inserted
#define TH_XX  THREAD(XX) // ex. #define TH_BLK THREAD(BLK)
```

```
void JPEGtop(){
for(i = 0; i < sizeYPadding; ){ // Loop L0
  for(ii = 0; ii < 8; ii ++){ // Loop L1
    ReadOneLine(i ++); // RGB->Y0/Y1, Cb,Cr
    ReadOneLine(i ++); // RGB->Y0/Y1, Cb,Cr
  } // loop L1
  TH_BLK {
    int nR = (i - 0 >= sizeY);
    for(j = 0; j < sizeX; j += 16){//Loop L2
      int nC = (j + 8 >= sizeX);
      BLK8x8(&Y0[j], 0, 0, 0);
      BLK8x8(&Y0[j + 8], 0, 0, nC);
      TH_BLK_Y1 {
        BLK8x8(&Y1[j], 0, 0, nR);
        BLK8x8(&Y1[j+DCTSIZE],0,0, nC+nR);
      } // TH_BLK_Y1
      TH_BLK_C {
        BLK8x8(&Cb[j >> 1], 1, 1, 0);
        BLK8x8(&Cr[j >> 1], 1, 2, 0);
      } // TH_BLK_C
    } // loop L2
  } // TH_BLK
} // loop L0
}
```

```
void ReadOneLine(int i){
  TH_RGB { // RGB->YCbCr conversion
   ...
    TH_BUF{...}//line-buffer,CbCr downsample
  }
}
```

```
void BLK8x8(UCHAR * pcomp, int cID, int sID,
            int isDummy){
  CopyBlock(pcomp, coef, compID, isDummy);
  TH_DCT { DCTcore(coef, coef2);}
  TH_Q {
    Quant(coef2, qcoef, cID);
    if(!isDummy){ UpdateDC(cID, sID, qcoef);}
  }
  TH_E {
   EncHuff(&put_buf,&put_bits,&buf,qcoef,cID);
   WriteBits(&put_buf, &put_bits, &buf);
  }
}
```

```
void emit_byte(int b_data)
{
  THREAD(OUT){ ... } // store byte out
}
```

Fig. 2 JPEG encoder program with thread scopes for the top function.

```
#else
#define TH_XX // ex. #define TH_BLK
#endif
```

Examples of thread scope insertion macro enable conditions (`ENABLE_XX`) are shown below.

```
#define ENABLE_BLK (THREAD_COUNT >=3)//inserted on spc=3,4,5,7,8,12,16
#define ENABLE_RGB (THREAD_COUNT>=4)//inserted on spc=4,5,7,8,12,16
#define ENABLE_E (THREAD_COUNT>=5)//inserted on spc=5,7,8,12,16
#define ENABLE_DCT (THREAD_COUNT>=7)//inserted on spc=7,8,12,16
#define ENABLE_Q (THREAD_COUNT>=7)//inserted on spc=7,8,12,16
#define ENABLE_BUF (THREAD_COUNT>=8)//inserted on spc=8,12,16
#define ENABLE_BLK_Y1 (THREAD_COUNT>=12)//inserted on spc=12,16
#define ENABLE_BLK_C (THREAD_COUNT>=16)//inserted on spc=16
```

First `#define` enables thread scope insertion of XX in that location, whereas the second `#define` converts `TH_XX` into null-string which as no effect on the source code. A desired set of thread scopes can be inserted by changing the condition before the macro definitions. Thus, thread scope semantic of our TCT programming model allows the designer to test different system partition strategy on the same source code by simply changing the definitions of these macros. In this work, we first explore the system partition instances described below. Here, root thread and THREAD(OUT) are included on all system partition instances, where root thread is assumed to be executed on the processor, and THREAD(OUT) as external device storing the output data, where both threads will not be the target for HLS HW generation.

- SP3: root, BLK, OUT
- SP4: root, RGB, BLK, OUT
- SP5: root, RGB, BLK, E, OUT
- SP7: root, RGB, BLK, DCT, Q, E, OUT
- SP8: root, RGB, BUF, BLK, DCT, Q, E, OUT

TCT thread scope semantic offers another interesting feature for system partitioning when thread scopes are inserted in the locations TH_BLK_Y1 and TH_BLK_C. Function BLK8x8 called from these locations contains DCT, Q and E threads (when enabled by the `TH_XX` macro definition). When multiple thread scopes (BLK, BLK_Y1, BLK_C) calls this function, DCT, Q and E threads will be cloned at each caller threads. This hierarchical partition structure allows the designer to create a large number of partitioned threads efficiently in the C code. By enabling thread scopes for BLK_Y1 and BLK_C, we add the below set of system partition instances in the exploration space:

- SP12: root, RGB, BUF, BLK, DCT, Q, E, BLK_Y1, DCT2,

© 2015 Information Processing Society of Japan

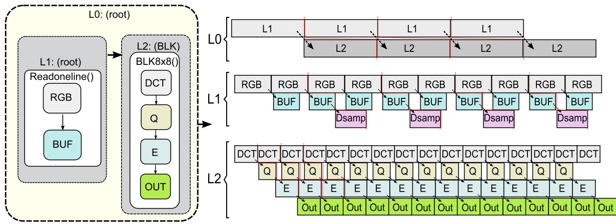


Fig. 3 Functional pipelines formed by concurrent threads in JPEG.

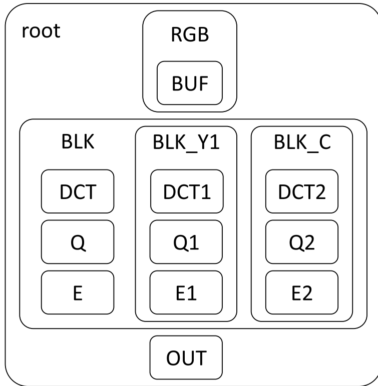


Fig. 4 System partition structure "SP16" of JPEG encoder (all other system partitions can be derived by disabling some of these thread scopes).

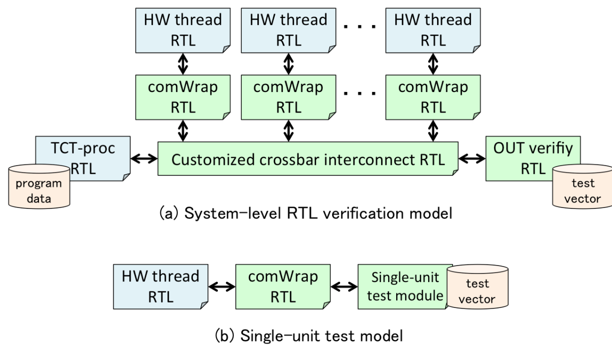


Fig. 5 System verification RTL models generated by our design framework.

Q2, E2, OUT

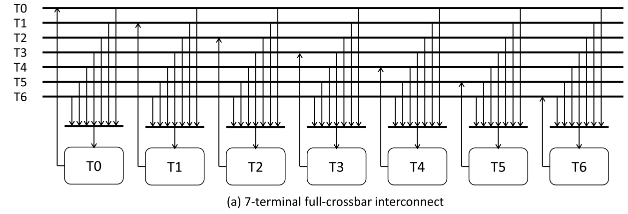
- SP16: root, RGB, BUF, BLK, DCT, Q, E, BLK_Y1, DCT2, Q2, E2, BLK_C, DCT3, Q3, E3, OUT

Figure 3 shows the core loop structures of system partition SP8 and its timing behavior, where directed edges in the left part of the figure denotes the dependencies between threads. As this figure shows, the concurrent execution model created by TCT compiler operates in hierarchical functional pipeline fashion, where the iteration throughput of each pipeline is different for each loop structure (loops L0, L1 and L2 in Fig. 2).

Figure 4 shows the hierarchical system partition structure of SP16. Here, loop structures are dependency edges are omitted for simplicity, where vertically placed threads indicate dependencies and horizontally placed threads indicate parallelisms.

3.2 System Model Generation

For each system partition instance, a complete set of component models are generated for RTL integration and functional verification. **Figure 5** shows the system-level RTL verification models and single-unit test model automatically generated by our framework.



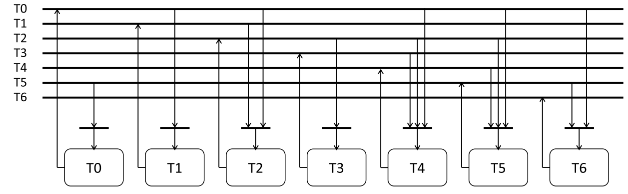


Fig. 6 Crossbar interconnect structure (a) full-crossbar (b) customized partial-crossbar according to the communication requirement of the partitioned application.

3.2.1 RTL Generation of SW/HW Threads

Separate C-source file is generated for each "thread scope" region, which can be implemented either as SW thread executed on a processor or a HW thread synthesized by HLS tool. These partitioned behavioral C-codes contain communication API calls that are responsible for inter-thread communications (thread activation, data transfer and data receive).

In this paper, we assume that the main thread (code regions outside any thread scopes) is implemented as SW thread only, and all others are implemented as either SW or HW threads. For SW thread, we use the RTL model of our TCT processor [22] which contains inter-processor communication module for fast message passing. Also, as explained in Section 3.1, OUT thread is assumed to be an external device storing the output data, where RTL verification module is generate as shown in Fig. 5 (a).

3.2.2 Customized Partial Crossbar Interconnect RTL Model Generation

Our original efficient interconnect infrastructure [22] consisting of communication module inside the processor and a fast crossbar switch is automatically generated as RTL model. Here, instead of providing a full point-to-point connectivity with a full crossbar connection required for multi-core systems which may run different applications, we optimize the connection according to the connectivity requirements of the given system partition instance which results in drastic decrease in interconnect circuit area.

In **Fig. 6** (a), a full crossbar connection is shown, whereas in Fig. 6 (b) a partial crossbar connection which reflects the connectivity requirements of the 7-thread design instance (SP7) given in Section 3.1 as shown below:

T0 → {T1, T2, T4, T5, T6}, T1 → {T2}
T2 → {T3, T4, T5}, T3 → {T4}
T4 → {T5}, T5 → {T0, T6}, T6 → {empty}

Since $N : 1$ multiplexer can be implemented as a tree of $N - 1$ 2-input multiplexers, the total number of 2-input multiplexers reduces from 42 multiplexers in original full-crossbar interconnect to only 6 multiplexers in the customized partial crossbar interconnect.

Figure 7 shows the comparison of full-crossbar complexity against customized partial-crossbar. Here, the connectivity re-

© 2015 Information Processing Society of Japan

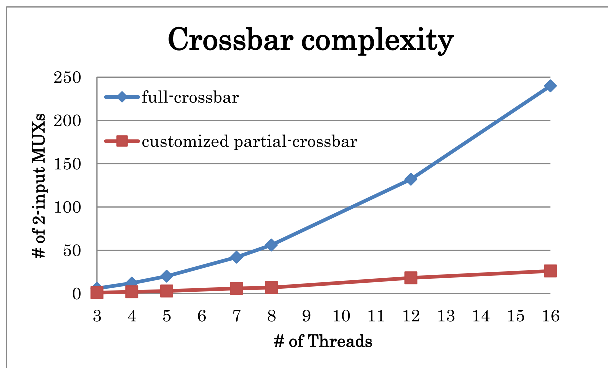


Fig. 7 Comparison of crossbar complexity: full-crossbar vs. customized partial-crossbar.

quirements correspond to SP3, SP4, . . . SP12, SP16 in Section 3.1 for JPEG encoder case.

As we can see in the figure, while the number of 2-input multiplexers on N-terminal full-crossbar increases in the order of $O(N^2)$, the customized partial-crossbar increases at a much lower rate, which presents a good scalability for large number of terminals.

3.2.3 Communication Wrapper Module Generation for HW Threads

Due to the different communication protocol supported by the HLS tool eXCite and our TCT interconnect, we automatically generate the communication wrapper module for the protocol conversion as reported in Ref. [8]. This communication wrapper needs to be custom-generated for individual HW threads since the configuration of communication channels (data size, input/output port IDs, source/destination information) differs.

3.2.4 Test Vector Generation for SW/HW Threads

From the original C-codes containing thread scope descriptions, code instrumentation is performed to generate the test vector at the interconnect boundaries. Input vectors and output vectors feeding into and out of each SW/HW thread is generated in separate files which are used for individual SW/HW component functional verification with RTL simulation as well as the verification of the entire SW/HW system. For the system-level verification, output data received by THREAD(OUT) module are checked against the output data obtained this test vector generation framework.

For single-unit test model shown in Fig. 5 (b), single-unit test module containing the input/output vectors for a particular thread drives the RTL simulation for the HW thread via communication wrapper module for thread-level functional verification. Also, this single-unit test model is used to verify the HW timing extraction method described in the later section.

3.3 HW Synthesis Exploration

Here, we describe the tool flow for generating a set of RTL descriptions for each HW component (partitioned C-code) with different resource allocation strategies using HLS tool. The objective here is to generate a variety of HW versions having area-time tradeoffs so as to provide the designer with wide spectrum of design choices on each system partitioning instance. On the particular HLS tool "eXCite" we utilize for this work, we use the following synthesis options to create multiple HW versions:

- "Best performance with maximum component sharing": Sufficient number of datapath components are allocated to maximize performance (minimizing the number of control steps) while components are shared as much as possible.
- "Maximum component sharing": In this synthesis option, the user can additionally specify the datapath resource allocations (number of datapath components with particular bitwidth for each component type). If no resource allocation is specified, the tool will allocate the minimum (one at most) for each datapath component type.

Table 1 Resource allocation of ADD components on DCT HW-thread (numbers in bracket indicates bit-widths, and ADD (all) is the total component count).

HW version #	1	2	3	4
ADD(7)	12	4	0	0
ADD(9)	1	1	0	0
ADD(10)	3	3	0	0
ADD(17)	1	1	1	0
ADD(18)	3	3	3	0
ADD(27)	2	2	2	0
ADD(32)	2	2	2	1
ADD(all)	24	16	8	1

In order to automate the generation of J multiple HW component versions, we developed a tcl script that controls the eXCite tool to perform the following tasks:

(1) Initial synthesis: Synthesize the target C-code with "Best performance with maximum component sharing" option.

(2) Synthesis of "middle" versions: Parse the generated RTL (Verilog) code from Step 1 to extract the resource allocation information, and then generate $J-2$ varieties of resource allocation constraints for each component type to produce these $J-2$ "middle" versions. Here, we take a simple strategy of linearly sweeping the initial "best performance" resource allocation.

(3) Synthesis of "maximum component sharing" version: Finally, synthesize the target C-code with no resource allocations to generate the most compact RTL.

In **Table 1**, resource allocations of adders with various bitwidths on $J = 4$ HW versions are shown for DCT HW-thread. To deal with the resource allocation of various bit-width, we first create a resource allocation list in the increasing order of bit-widths, and gradually decrease the resource allocation from the smallest available bit-width. In the example shown in Table 1, there are initially 24 ADD components with bit-widths ranging from 7 to 32. Using the linear sweep strategy among J versions with M components at version 1 ("Best Performance"), the number of components at version j ($j = 1, \ldots, J$) is set as $\left\lceil \frac{M-1}{J-1}(J-j)+1 \right\rceil$.

4. Area-time Pareto-optimal Design Point Evaluation

In this section, we explain the overall design space exploration methodology to present the designers with area-time Pareto-optimal design curves. One of the important challenges in design space exploration is to provide a search space pruning technique to drastically reduce the search space size. Another key issue is how to reduce the evaluation time of each design points,

© 2015 Information Processing Society of Japan

where the conventional means was via time consuming RTL simulations. We will first outline our previous work on the timing extraction technique for HW components generated with HLS tool, which becomes an essential means to realize the design search space pruning process as well as reducing the overall design evaluation time.

4.1 HW Timing Extraction Technique

Our timing extraction technique for HW components is based on the novel trace-driven workload simulation technology [7]. This technology is composed of the following key concepts:

- Branch bitstream is a compact encoding scheme for representing the program execution trace as a sequence of branch condition bits. It is generated through source-level instrumentation for emitting branch condition bits in the execution order.
- Program trace graph (PTG) is an abstract workload model of the application that contains accurate cycle counts for individual instruction streams. PTG-node consists of function-start, function-end, branch, call, as well as communication instructions (CT, DT, DS) inserted by the TCT compiler. PTG-edge corresponds to the code segment bordered by PTG-node instructions and is annotated with accurate cycle count information.

Figure 8 (a) shows an example of PTG, where a PTG-node consists of function-start, function-end, branch and call. PTG-edge corresponds to the code segment without conditional jumps, and the timing information is annotated on the PTG-edge. Also in Fig. 8 (a), branch bitstream of 1011001 represents the program execution trace that can be decoded into the PTG-edge sequence (e_0 e_1 e_7 e_9 e_2 e_3 e_3 e_4 e_5 e_7 e_8 e_6). This trace decoding is accomplished by traversing the PTG and reading the branch bitstream one bit at a time upon reaching a branch node to determine which branch path to continue traversing.

In similar way we construct a PTG for each thread called the thread program trace graph (T-PTG) that corresponds to the PTG that is enclosed by the SESE thread-scope region as shown in Fig. 8 (b). Each T-PTG is terminated by thread-start and thread-end nodes instead of function-start and function-end nodes in the normal PTG. Each TCT thread includes TCT communication instructions (CT, DT, DS) that interact with other threads through MPSoC interconnect.

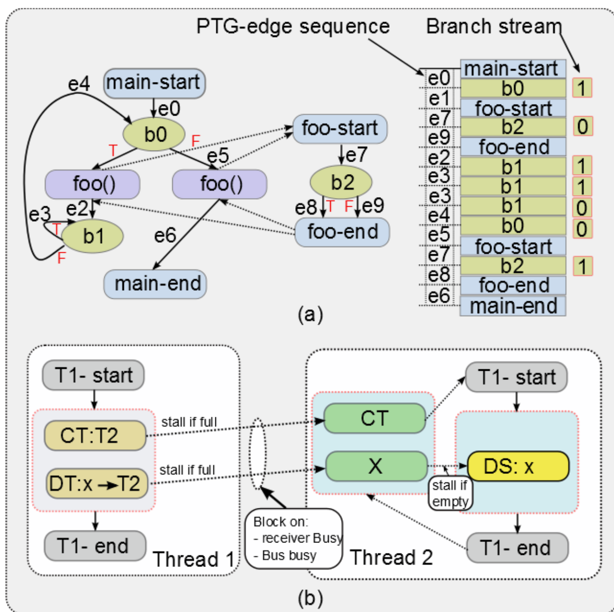


Fig. 8 (a) Program trace graph (PTG), (b) Thread-PTG-sync-nodes for modeling communication.

Based on this technology, we have previously extended the PTG representation to model the algorithmic state machine of the HW components generated by HLS tool eXCite [8]. It consists of the following steps:

(1) State transition graph extraction: RTL code generated by eXCite is parsed and the state transition information is extracted to construct the state transition graph (STG).

(2) STG reduction: Unconditional STG-edges are collapsed and annotated as the state machine cycles between adjacent conditional states. Also, a group of conditional STG-edges corresponding to the handshaking protocol of the message channel accesses of the HW component are identified and replaced with communication state nodes of the matching TCT communication instructions (CT, DT, DS).

(3) Graph matching between reduced-STG and PTG: After STG reduction, the reduced-STG reflects the same abstract computational model as PTG of the original SW code. Graph matching is performed to identify the corresponding edge pairs of reduced-STG and PTG. The state machine cycles on the STG-edges are then back-annotated to the corresponding PTG-edges to reflect the HW timing.

4.2 Timing Characterization of HW Threads

Timing behavior of HW components depends on two factors. First is the state machine cycles annotated to the PTG-edge that characterizes the individual fine-grain computational workload of the HW's algorithmic state machine. Second is the input-dependent program execution behavior encoded as branch bitstream and then decoded back as PTG-edge sequence. In the case of JPEG encoder, input data set includes the input image as well as the compression rate setting where HW timing behavior can change drastically depending on the nature of input image and the compression rate. In case of THREAD(Q), DCT coefficient quantization contains conditional check to determine whether the costly division is necessary. In the case of THREAD(E), Huffman encoding behavior depends largely on the sequence of quantized DCT coefficients. For these reasons, it is very important that the design space exploration is carried out on a variety of input data instead of a single input data. We will use the below two notations to characterize the timing behavior of HW components, where j is the HW version index and k is the index of the input data set used in the application execution:

- PTG-edge cycle count $c_j(e_i)$: The timing information annotated to PTG-edge e_i of the jth HW version characterizes the fine-grain workload of the corresponding set of computations.
- PTG-edge occurrence $n_k(e_i)$: On the kth input data set, branch bitstream is generated and decoded as PTG-edge sequence as described previously. PTG-edge occurrence

© 2015 Information Processing Society of Japan

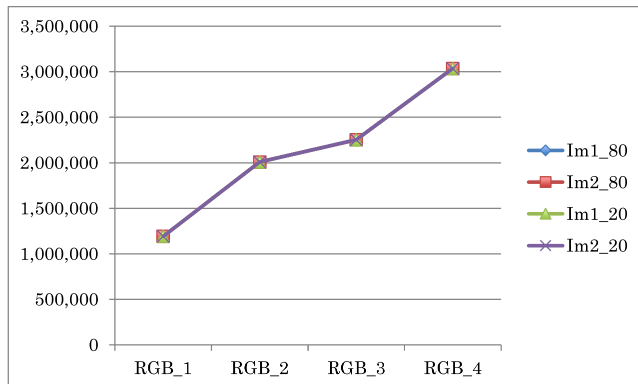
$n_k(e_i)$ is the number of occurrence of PTG-edge e_i in the kth PTG-edge sequence. As reported in Ref. [8], PTG-edge sequence decoding takes roughly the same amount of computation time compared to the application's execution time, and where the enumeration of PTG-edge occurrence can be done during the PTG-edge sequence decoding.

From the above two annotated parameters $c_j(e_i)$ and $n_k(e_i)$ on PTG-edge e_i, HW cycle count $T(j,k)$ of jth HW version on k^{th} input data set is simply calculated as the sum-of-products of the two parameters:

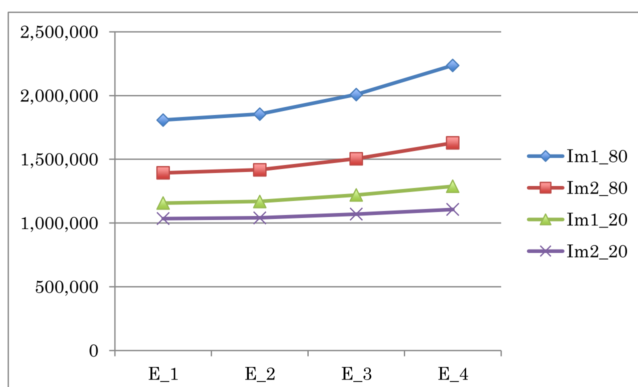
$$T(j,k) = \sum c_j(e_i) \cdot n_k(e_i) \quad (1)$$

The significance of Eq. (1) is that once the PTG-edge cycle counts on J sets of HW versions are extracted, using our HW timing extraction technique, and K sets of PTG-edge occurrences are enumerated through PTG-edge sequence decoding, then $J \times K$ combinations of HW cycle counts $T(j,k)$ can be computed essentially *on-the-fly*. This fact is a large contrast to the conventional HW timing characterization that requires $J \times K$ sets of RTL simulation runs. These HW cycle counts represent the overall internal workload of the algorithmic state machine inside each HW thread, but excludes other timing factors induced by the interconnect and inter-thread communications, which are to be handled later by our trace-driven workload simulation framework.

Figure 9 shows the HW cycle counts of two HW threads on various HW versions and input data sets. Here, "ImX_Y" in the graphs is the input data set (X: input image data, Y: compression setting) where the two images have the same size of 300x400 pixels. HW cycle counts of THREAD(RGB) shown in Fig. 9 (a) are identical on all input data sets, since its computation workload is independent on the image data and compression setting. On the other hand, HW cycle counts of THREAD(E) shown in Fig. 9 (b) present high dependency on the input data sets.



(a) HW cycle counts of THREAD RGB



(b) HW cycle counts of THREAD E

Fig. 9 HW cycle counts (Y axis) on various HW versions (X axis) and input data sets (each curve). (a) THREAD (RGB), (b) THREAD (E).

4.3 Pareto-Optimal Design Candidate Selection

Here, design search space pruning technique utilizing the HW cycle counts is explained. In generating area-time Pareto-optimal design curves of the SW/HW system, there is one issue of how to take into account the difference in the program execution behavior on various input data sets in the design search space. In this work, we choose to perform the design exploration separately on each input data set, that is, for K sets of input data used for program execution, K separate Pareto-optimal design curves will be generated. Therefore, in this part, we focus on enumerating the Pareto-optimal design candidates for a single program execution trace on certain input data set.

4.3.1 HW Version Set Pruning

The first step in design space pruning is a trivial process of eliminating HW versions that are not area-time Pareto-optimal. As described in the previous section, various HW versions are generated by linearly sweeping the resource allocations for each component type, in the expectation that smaller resource allocation leads to smaller circuit area. While this expectation holds true for many cases, there are some cases where the circuit overhead for inserting multiplexers on shared datapath components exceeds the resource allocation reduction thereby increasing the overall circuit area.

Also, SW threads are treated in the same manner as HW threads, where SW cycle counts are obtained from the original PTG-edge cycle counts that corresponds to the SW execution on the processor, and SW thread area corresponds to the processor area. HW version index $j = 0$ is allocated for SW thread, and $j = 1, 2, \ldots, J$ corresponds to HW threads. **Figure 10** shows the area-time plots of HW/SW threads on system partitions SP3 and SP4. While RGB thread is same for SP3 and SP4, BLK thread in SP3 is split into BLK and E threads in SP4. Here, dotted red circles denote non-Pareto-optimal HW versions. The circuit area data were obtained from Synopsys Design Compiler on 0.18 um CMOS library, where the area unit is measured in um^2. For system partitions with more number of threads (SP5, SP7, SP8, SP12, SP16), all synthesized HW threads had smaller area and smaller cycle counts compared to that of the processor (SW thread), and thus SW threads were all excluded from the area-time Pareto-optimal HW version sets on these system partitions.

Table 2 shows the area-time Pareto-optimal HW version set and the total design space size for each system partition instance. Total design space size is given as the product of the number of HW versions for all HW threads, corresponding to an exhaustive search where all combinations of HW versions at each HW thread are evaluated. Even after the HW version pruning, we can observe the exponential growth of the design space size on the number of HW threads.

© 2015 Information Processing Society of Japan

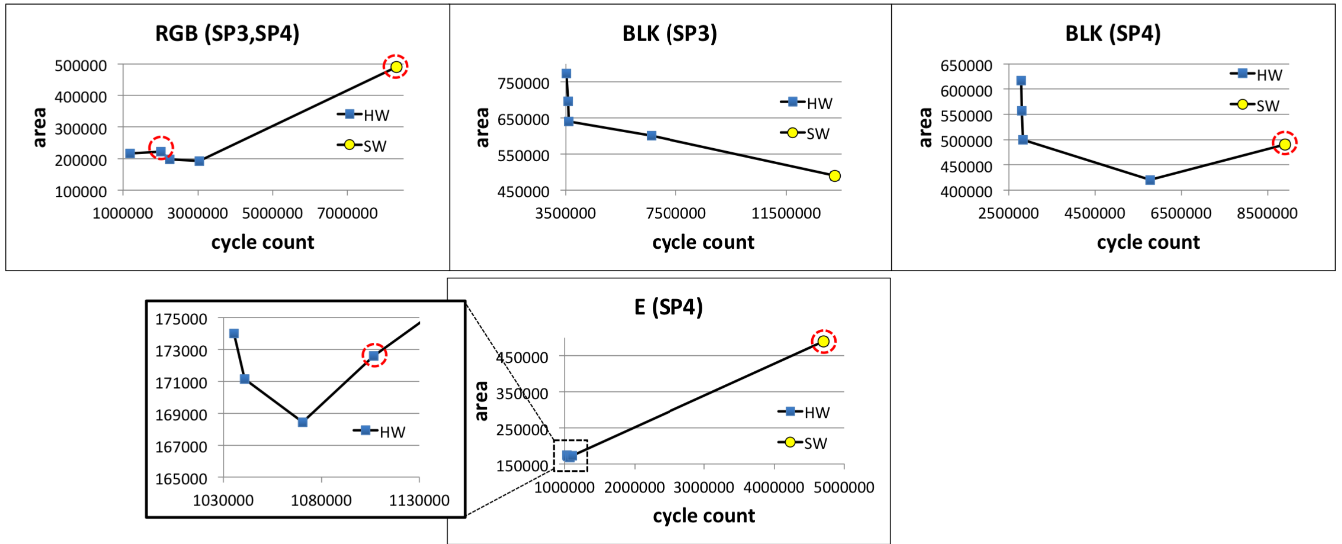


Fig. 10 Area-time plots of HW/SW threads on SP3 and SP4 (dotted red circles denote non-Pareto-optimal HW versions).

Table 2 Area-time Pareto-optimal HW version set and the total design space sizes for each system partition instance.

System partitioning	Pareto-optimal HW version sets (numbers represent HW version indices)	#design points
SP3	BLK{ 0, 1, 2, 3, 4 }	5
SP4	RGB{ 1, 3, 4}, BLK{ 0, 1, 2, 3, 4}	15
SP5	RGB{1, 3, 4}, BLK{1, 2, 3, 4}, E{1, 2, 3}	48
SP7	RGB{1, 3, 4}, BLK{1, 2, 3, 4}, DCT{1, 3, 4}, Q{1}, E{1, 2, 3}	108
SP8	RGB{1, 3}, BUF{1, 2, 3, 4}, BLK{1, 2, 3, 4}, DCT{1, 3, 4}, Q{1}, E{1, 2, 3}	288
SP12	RGB{1, 3}, BUF{1, 2, 3, 4}, BLK{1}, DCT{1, 3, 4},Q{1},E{1, 2, 3}, BLK_Y1{1, 4}, DCT2{1, 3, 4}, Q2{1}, E2{1, 2, 3, 4}	1,728
SP16	RGB{1, 3}, BUF{1, 2, 3, 4}, BLK{3, 4}, DCT{1, 3, 4}, Q{1}, E{1, 2, 3}, BLK_Y1{1, 4}, DCT2{1, 3, 4}, Q2{1}, E2{1, 2, 3, 4}, BLK_C{2, 3}, DCT3{1, 3, 4}, Q3{1},E3{1, 2, 3}	62,208

4.3.2 HW Design Point Enumeration

As the final preparation phase for the actual design space exploration, we describe the algorithm for reducing the design space size by enumerating a small set of design points that are expected to be Pareto-optimal. As illustrated in Fig. 3, TCT Model generates a system partition where threads execute in pipeline fashion. The key insight for deriving the HW design point enumeration algorithm is the fact that pipeline throughput is determined by the critical (slowest) pipeline stage, and therefore speeding up this critical pipeline stage will lead to increase in the overall pipeline throughput.

Each design point within the design space is denoted as $D_N(i_1, i_2, \ldots, i_N)$ where N is the number of HW threads in the system partition, and i_n is the HW version index on n^{th} HW thread $(n = 1, 2, \ldots, N)$. Due to the HW version pruning process, each HW version set is sorted by the decreasing order of area and increasing order of cycle counts. Also, the HW cycle count notation $T(j, k)$ in Eq. (1) is modified to $T_n\ (i_n)$ on HW version i_n on n^{th} HW thread. Here, input data set index k is ignored since the design space evaluation is separated for each input data set. Next, we define two operators for retrieving the HW version index i_n:

DESIGN_SPACE_ENUM(N):

(1) $DP_{set} \leftarrow \phi,\ i_n \leftarrow IDX_AO(n)\ \ (n = 1,2,\cdots,N)$

(2) $c \leftarrow 0,\ T_{max} \leftarrow 0,\ DP_{set} \leftarrow DP_{set} \cup \{D_N(i_1, i_2, \cdots, i_N)\}$

(3) FOR $(n = 1,2,\cdots,N)$

  IF $(IDX_NEXT(n, i_n) \neq 0\ \&\&\ T_{max} < T_n(i_n))$

   $c \leftarrow n, T_{max} \leftarrow T_n(i_n)$

(4) IF $(c \geq 1)$

  $i_c \leftarrow IDX_NEXT(c, i_c)$, GOTO (2)

(5) END

Fig. 11 HW design point enumeration algorithm.

Table 3 HW design point enumeration on system partition SP7 (RGB{1, 3, 4}, BLK{1, 2, 3, 4}, DCT{1, 3, 4}, Q{1}, E{1, 2, 3}).

HW version index HW cycle counts				
RGB	BLK	DCT	Q	E
(4) 3,035,797	(4) 1,937,734	(4) 2,396,884	(1) 1,718,477	(3) 1,504,628
(3) 2,252,744	(4) 1,937,734	(4) 2,396,884	(1) 1,718,477	(3) 1,504,628
(3) 2,252,744	(4) 1,937,734	(3) 1,051,684	(1) 1,718,477	(3) 1,504,628
(1) 1,192,972	(4) 1,937,734	(3) 1,051,684	(1) 1,718,477	(3) 1,504,628
(1) 1,192,972	(3) 943,084	(3) 1,051,684	(1) 1,718,477	(3) 1,504,628
(1) 1,192,972	(3) 943,084	(3) 1,051,684	(1) 1,718,477	(2) 1,418,078
(1) 1,192,972	(3) 943,084	(3) 1,051,684	(1) 1,718,477	(1) 1,393,839
(1) 1,192,972	(3) 943,084	(1) 983,284	(1) 1,718,477	(1) 1,393,839
(1) 1,192,972	(2) 917,434	(1) 983,284	(1) 1,718,477	(1) 1,393,839
(1) 1,192,972	(1) 914,584	(1) 983,284	(1) 1,718,477	(1) 1,393,839

- $IDX_AO(n)$: This operator returns the highest HW version index (area-optimal HW version index) of nth HW thread.
- $IDX_NEXT(n, i_n)$: This operator returns the highest HW version index after i_n (that is, the index immediately smaller than i_n). If i_n is the smallest HW version index, it returns 0.

The HW design point enumeration algorithm is described in **Fig. 11**. DP_{set} is the enumerated design point set. In (1), the algorithm starts from the area-optimal design point. In (3), the critical HW cycle count $T_{\max}$ and the corresponding HW thread index c is evaluated. In (4), HW version index i_c of the critical HW thread is updated to $IDX_NEXT(c, i_c)$ which is the HW version with less HW cycle count.

Table 3 shows the HW design point enumeration on 5 HW threads (SP7). In the table, shaded cells represent the critical HW version at each design point, where they are replaced by smaller

© 2015 Information Processing Society of Japan

Table 4 Comparison of design space search sizes.

# System Partitioning	# HW versions at each thread	$SIZE_{ENUM}$	$SIZE_{TOTAL}$
SP3	5	5	5
SP4	3, 5	7	15
SP5	3, 4, 3	8	36
SP7	3, 4, 3, 1, 3	10	108
SP8	2, 4, 4, 3, 1, 3	12	288
SP12	2, 4, 1, 3, 1, 3, 2, 3, 1, 4	15	1728
SP16	2, 4, 2, 3, 1, 3, 2, 3, 1, 4, 2, 3, 1, 3	21	62,208

HW version indices on the succeeding design point.

Next, let us compare the total number of enumerated design points obtained by our algorithm against exhaustive search space size. Let S_n be the number of HW versions on n^{th} HW thread. Then the number of enumerated design points $SIZE_{ENUM}$ and the exhaustive search space size $SIZE_{TOTAL}$ are given as :

$$SIZE_{ENUM} = \sum_{n=1}^{N} S_n - N + 1$$

$$SIZE_{TOTAL} = \prod_{n=1}^{N} S_n \quad (2)$$

Table 4 shows the design space size comparison on each system partition instance.

5. Experimental Results

In this section, we first show the experimental results on the accuracy of our HW timing characterization technique and trace-driven workload simulation on the overall SW/HW system automatically synthesized from our proposed design space exploration framework. We then show the design space coverage of the proposed HW design point enumeration algorithm compared against the exhaustive design search space. Finally, the overall design space exploration results as a set of area-time Pareto-optimal curves on various system partitions and input data sets are shown.

For the experiments in this section, following machine environments were used to run the RTL simulation and our design space exploration framework:

(1) RTL simulation: Synopsys VCS on Linux server machine (2.53 GHz Intel Xeon E5630 16-core CPU with 16 GB memory)

(2) Design space exploration framework (including trace-driven workload simulation): Windows 7 machine (3.4 GHz Intel Core i7-2600 CPU with 8 GB memory)

5.1 Comparison of Trace-Driven Workload Simulation and RTL Simulation

Here, timing accuracy of our HW thread timing characterization technique described in Section 4.2 is examined. **Table 5** shows the comparison of cycle counts obtained by trace-driven workload model in Eq. (1) against actual RTL simulation on a given input data set. Here, RTL simulation is performed separately on each HW thread on the single HW unit test environment explained in Section 3.2.4. We can see that the HW cycle counts estimated by our model have high accuracy with 1.38% average estimation error on 5 HW (SP7) version set.

In **Table 6**, we show the cycle count comparison of the

Table 5 Comparison of HW cycles count of each HW version on RTL simulation and trace-driven workload model (TW) cycle counts on system partition SP7.

HW versions	RTL cycle count	RTL sim time (sec)	TW cycle count	estimation error
RGB_1	1,137,495	9.05	1,192,972	4.65 %
RGB_3	1,952,248	14.38	2,008,636	2.80 %
RGB_4	2,979,713	16.04	3,035,797	1.84 %
BLK_1	919,309	8.98	914,584	0.51 %
BLK_2	922,178	9.35	917,434	0.51 %
BLK_3	947,828	8.44	943,084	0.50 %
BLK_4	1,942,478	12.48	1,937,734	0.24 %
DCT_1	970,424	7.03	983,284	1.30 %
DCT_3	1,041,674	7.37	1,051,684	0.95 %
DCT_4	2,386,874	10.91	2,396,884	0.41 %
Q_1	1,769,608	5.45	1,718,477	2.97 %
E_1	1,388,782	8.50	1,393,839	0.36 %
E_2	1,415,871	8.36	1,418,078	0.15 %
E_3	1,472,933	7.92	1,504,628	2.10 %

Table 6 Comparison of SW/HW system cycles count on RTL simulation and trace-driven workload simulation on system partition SP7 (entire trace-driven workload simulation time is 0.348 sec).

HW design points	RTL cycle count	RTL sim time (sec)	TW cycle count	estimation error
{4,4,4,1,3}	3,264,084	142.53	3,226,879	1.13 %
{3,4,4,1,3}	2,692,388	139.31	2,693,684	0.04 %
{3,4,3,1,3}	2,452,460	132.81	2,435,549	0.68 %
{1,4,3,1,3}	2,280,539	124.90	2,255,805	1.08 %
{1,3,3,1,3}	2,011,140	117.70	1,996,898	0.70 %
{1,3,3,1,2}	2,011,002	117.57	1,954,135	2.82 %
{1,3,3,1,1}	2,011,757	117.50	1,943,311	3.40 %
{1,3,1,1,1}	2,013,908	116.66	1,942,467	3.54 %
{1,2,1,1,1}	2,012,802	118.09	1,942,086	3.51 %
{1,1,1,1,1}	2,012,217	117.83	1,942,067	3.48 %

SW/HW system-level behavior of trace-driven workload simulation against RTL simulation. As we have explained in Section 4.1, our trace-driven workload simulation framework models the dynamic timing behavior of the interconnect, such as data transfer stalls due to request collision on the same destination, and inter-thread communication, such as buffer-full stalls at the senders and buffer-empty stalls at the receivers. As we can see in the table, our trace-driven workload simulation exhibits high cycle accuracy having 2.80% average estimation error on the system-level simulation. Comparing the simulation time, 10 RTL simulation runs take 1,245 seconds, whereas the trace-driven workload simulator requires only 0.348 seconds, which translates to a 5,000 times speedup. Here, the trace-driven workload simulator takes PTG information and branch bitstream data as inputs, and performs PTG-edge sequence decoding during the workload simulation to obtain the overall cycle counts reported in Table 6.

5.2 Pareto-Optimality Evaluation of Enumerated HW Design Points

We have already shown in Eq. (2) that the design point enumeration algorithm can drastically reduce the design space search size, from exponential size to near-linear size with respect to the number of HW threads. What we need to verify is the quality of the enumerated design points, that is, how close they are located to the actual Pareto-optimal curve. In **Fig. 12**, we compare the Pareto-optimal design points obtained by exhaustive search against the one obtained by our design point enumeration algorithm on the 4 input data sets on 10 HW-thread partition. For the

© 2015 Information Processing Society of Japan

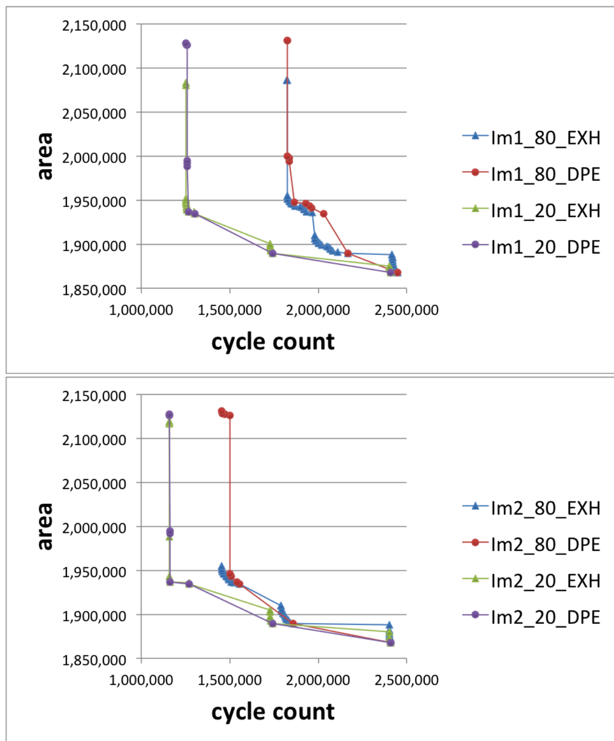


Fig. 12 Pareto-optimal curves obtained by exhaustive search (EXH) and our design point enumeration method (DPE) on system partition SP12.

inputs with high compression setting (Im1_20 and Im2_20), our enumeration algorithm can find many of the Pareto-optimal design points. For the inputs with low compression setting (Im1_80 and Im2_80), however, the shape of Pareto-optimal curve becomes more complex in which our algorithm fails to enumerate many of those finely sampled Pareto-optimal design points. This limitation of our enumeration algorithm mainly comes from the fact that the criticality of each HW threads on the overall system performance is approximated simply by the total HW cycle count, that is, the throughput of each HW thread is estimated by the average workload. For HW threads whose workload is highly dependent on the input data (such as THREAD(Q) and THREAD(E)), there is a large temporal fluctuation in the individual workload (cycle counts of processing a single 8x8 block data) that can have larger impact on the system throughput. Improvement of our design point enumeration algorithm will be addressed in our future works. Despite its limitations on several cases, we believe that our design point enumeration algorithm provides an effective means to prune out the huge design space that is essential in carrying out the design space exploration.

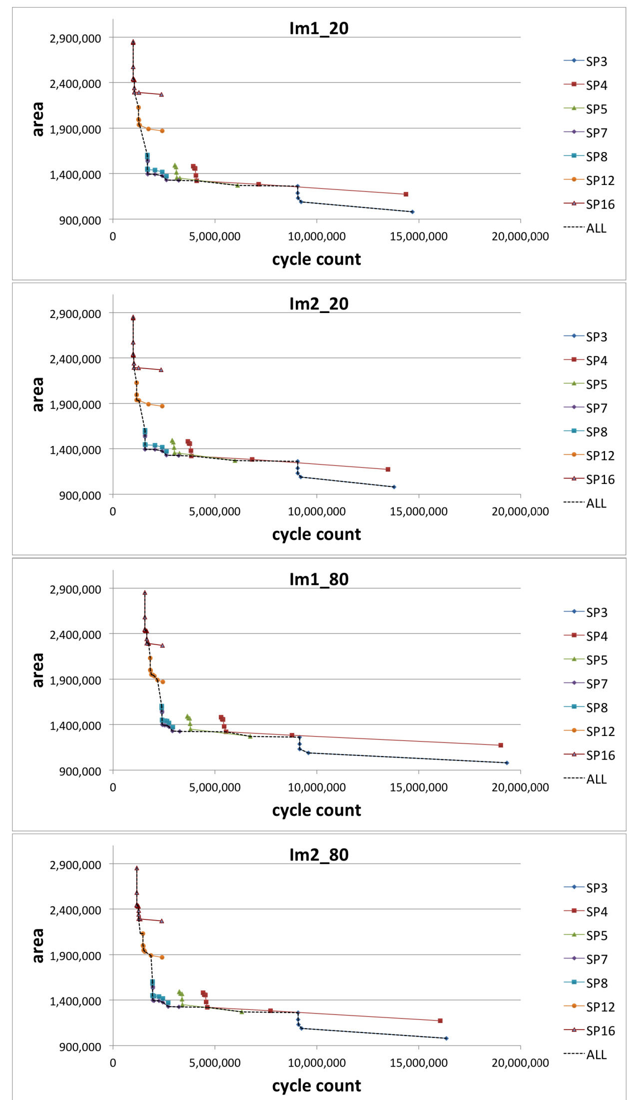


Fig. 13 Pareto-optimal curves obtained by our design point enumeration method on 7 system partition instances.

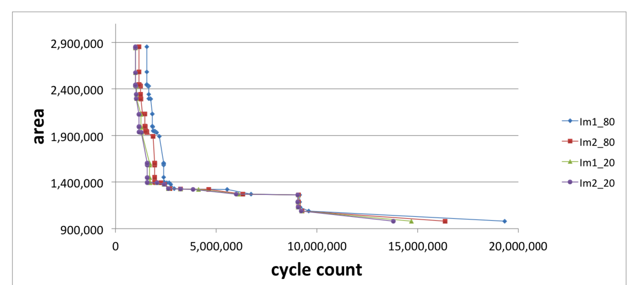


Fig. 14 Overlay of Pareto-optimal design curves on the 4 input data sets.

5.3 Pareto-Optimal Design Curves of All System Partition Instances on Various Input Data Sets

Figure 13 shows the overall Pareto-optimal design points approximated by our design point enumeration algorithm of all 7 system partition instances explained in Section 3.1 on the 4 input data sets. As we can see in these figures, by the combination of multiple system partition instances together with multiple HW version combinations within each system partition instance, we are able to cover a very wide range of system performance and circuit area for the entire design space. Dotted lines in the figures show the Pareto-optimal points of the combined system partition instances. **Figure 14** shows the overlayed Pareto-optimal curves of the 4 input data sets. It can be clearly seen from the figure that system performance can vary with different input data sets, demonstrating that it is essential to evaluate the system performance on a wide range of input data sets in order to validate whether the system satisfies the given performance requirements.

6. Conclusions

In this paper, we presented a new design space exploration

© 2015 Information Processing Society of Japan

framework for SW/HW systems where two dimensions of design explorations, system partition exploration and HW synthesis exploration, are tackled with techniques based on TCT programming model and trace-driven workload simulation technology. We proposed a design space pruning method which drastically reduced the exponential size design space into near-linear size by utilizing the accurate HW timing models as the indicator for system bottleneck and guiding the enumeration process of HW version combinations. In the experiments, we showed detailed results on the accuracy of our trace-driven workload simulation technology compared to RTL simulation, which revealed an average estimation error of 1.38% for HW timing of each thread, and 2.80% estimation error for the system-level simulation, where the simulation speedup factor was in the order of 5,000 times. We also explained the system model generation features where all RTL components of the complete system are automatically generated (except for the processor element which was pre-designed) including customized partial crossbar interconnect, communication wrapper for HW threads for protocol conversion. In addition to above a complete system-level functional test environment is generated also as single-unit test environment to guarantee functional equivalency between the synthesized SW/HW system and the original sequential application. We believe that our design space exploration framework for SW/HW systems, even though the current implementation partially depends on the HLS tool eXCite, can be applied to other HLS tools by porting the communication protocol converters for RTL integration to our efficient interconnect and adapting the HW timing model extraction features.

Acknowledgments This work is supported by VLSI Design and Education Center (VDEC), the University of Tokyo, Synopsys, Inc., and Y Explorations, Inc.

References

[1] Wolf, W.: The future of multiprocessor systems-on-chips, *Design Automation Conference*, pp.681–685 (2004).

[2] Martin, G.: Overview of the MPSoC Design Challenge, *Design Automation Conference*, pp.274–279 (2006).

[3] Kanbara, H., Nakatani, T., Umehara, N., Ishihara, N. and Tomiyama, H.: Speed Improvement of AES Encryption using hardware accelerators synthesized by C Compatible Architecture Prototyper (CCAP), *Workshop on Synthesis And System Integration of Mixed Information Technologies* (*SASIMI 2007*), pp.130–134 (2007).

[4] Adding Hardware Accelerators to Reduce Power in Embedded Systems, available from ⟨http://www.altera.com/literature/wp/wp-01112-hw-reduce-power.pdf⟩

[5] Gries, M.: Methods for evaluating and covering the design space during early design development, *Integration, the VLSI Journal*, Vol.38, No.2, pp.131–183 (2004).

[6] Urifianto, M.Z., Isshiki, T., Khan, A.U., Li, D. and Kunieda, H.: Decomposition of task-level concurrency on C programs applied to the design of multiprocessor SoC, *IEICE Trans. Fundamentals*, Vol.E91-A, No.7, pp.1748–1756 (2008).

[7] Isshiki, T., Li, D., Kunieda, H., Isomura, T. and Satou, K.: Trace-Driven Workload Simulation Method for Multiprocessor System-On-Chips, *Design Automation Conference*, pp.232–237 (2009).

[8] Khan, A.U., Isshiki, T., Li, D. and Kunieda, H.: A Unified Performance Estimation Method for Hardware and Software Components in Multiprocessor System-On-Chips, *IPSJ Trans. System LSI Design Methodology*, Vol.3, pp.194–206 (2010).

[9] The Message Passing Interface (MPI) Standard, available from ⟨http://www-unix.mcs.anl.gov/mpi/⟩

[10] OpenMP, available from ⟨http://www.openmp.org⟩

[11] Paulin, P.G., Pilikington, C., Langevin, M., Bensoudane, E. and Nicolescu, G.: Parallel Programming Models for a Multi-Processor SoC Platform Applied to High-Speed Traffic Management, *Proc. International Conference on Hardware Software Codesign*, pp.48–53 (2004).

[12] Pazos, N., Ienne, P., Leblebici, Y. and Maxiaguine, A.: Parallel Modelling Paradigm in Multimedia Applications: Mapping and Scheduling onto a Multi-Processor System-on-Chip Platform, *The Global Signal Processing Conference* (*GSPx*) (2004).

[13] Ando, Y., Shibata, S., Honda, S., Tomiyama, H. and Takada, H.: Automatic Communication Synthesis with Hardware Sharing for Design Space Exploration, *IEEE International Symposium on Circuits and Systems* (*ISCAS*), pp.1863–1866 (2010).

[14] Y Exploration Inc., available from ⟨http://www.yxi.com/⟩

[15] Page, I. and Luk, W.: Compiling Occam into Field-Programmable Gate Arrays, *W. Moore and W. Luk, FPGAs, Oxford Workshop on Field Programmable Logic and Applications*, Abingdon EE and CS Books Abingdon UK, pp.271–283 (1991).

[16] Martin, G. and Smith, G.: High-Level Synthesis: Past, Present, and Future, *IEEE Design & Test of Computers*, Vol.26, No.4, pp.18–25 (2009).

[17] Lau, D., Pritchard, O. and Molson, P.: Automated Generation of Hardware Accelerators with Direct Memory Access from ANSI/ISO Standard C Functions, *IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM06*, pp.45–56 (2006).

[18] Samahi, A. and Bourennane, E.: Automated Integration and Communication Synthesis of Reconfigurable MPSoC Platform, *Second NASA/ESA Conference on Adaptive Hardware and Systems*, pp.379–385 (2007).

[19] Azumi, T., Samei Syahkal, Y., Hara-Azumi, Y., Oyama, H. and Domer, R.: TECSCE: HW/SW Codesign Framework for Data Parallelism Based on Software Component, *Embedded Systems: Design, Analysis and Verification 2013*, Vol.403, pp.1–13.

[20] van Stralen, P. and Pimentel, A.: Scenario-Based Design Space Exploration of MPSoCs, *IEEE International Conference on Computer Design* (*ICCD*), pp.305–312 (2010).

[21] Wild, T., Herkersdorf, A. and Lee, G.: TAPES-Trace-based architecture performance evaluation with SystemC, *Design Automation for Embedded Systems*, Vol.10, No.2-3, pp.157–179 (2005).

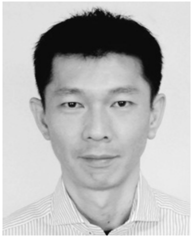
[22] Urifianto, M.Z., Isshiki, T., Khan, A.U., Li, D. and Kunieda, H.: A Multiprocessor SoC Architecture with Efficient Communication Infrastructure and Advanced Compiler Support for Easy Application Development, *IEICE Trans. Fundamentals*, Vol.E91-A, No.4, pp.1185–1196 (2008).

[23] Ceng, J., Castrillon, J., Sheng, W., Scharwachter, H., Leupers, R., Ascheid, G., Meyr, H., Isshiki, T. and Kunieda, H.: MAPS: An Integrated Framework for MPSoC Application Parallelization, *Design Automation Conference*, pp.754–759 (2008).

[24] Independent JPEG group, available from ⟨http://www.ijg.org⟩

Arif Ullah KHAN received his B.Sc. degree in Electrical Engineering from NWFP University of Engineering and Technology Pakistan in 2001 and M.Sc. degree in Information and Communication Engineering from Karlsruhe Institute of Technology, Germany in 2004. From 2008 to 2011 he worked as a researcher at Department of Communication and Integrated Systems, Tokyo Institute of Technology. Currently he is working as a specially appointed researcher at Department of Information Systems Engineering, Osaka University. His primary interests include various aspects of MPSoC design including ASIP design, hardware-software integration and network-on-chip. He is also interested in "System in Package" design and 3D NoC design.

© 2015 Information Processing Society of Japan



Tsuyoshi ISSHIKI received his B.E. and M.E. degrees in Electrical and Electronics Engineering from Tokyo Institute of Technology in 1990 and 1992, respectively. He received his Ph.D. degree in Computer Engineering from University of California at Santa Cruz in 1996. He is currently an Associate Professor at Department of Communications and Integrated Systems in Tokyo Institute of Technology. His research interests include MPSoC programming framework, high-level design methodology for configurable systems, bit-serial synthesis, FPGA architecture, image processing, fingerprint authentication algorithms, computer graphics, and speech synthesis. Prof. Isshiki is a member of IEEE CAS, IPSJ and IEICE.

Dongju Li received her B.S. degree from LiaoNing University and M.E. degree from Harbin Institute of Technology, China, in 1984 and 1987, respectively. She worked as an IC design engineer in VLSI Design Laboratory of Northeast Micro-electronics Institute, Electronic Industry Burean, China, from 1987–1993. She is currently a Research Associate at Department of Communications and Integrated Systems in Tokyo Institute of Technology. Her current research interests are in embedded fingerprint authentication algorithms, VLSI Architecture and Design Methodology. System on Chip design for multimedia processing including video CODEC. Dr. Li is a member of IEEE CAS and IEICE.

Hiroaki KUNIEDA was born in Yokohama in 1951. He received his B.E., M.E. and Dr. Eng. degrees from Tokyo Institute of Technology in 1973, 1975 and 1978, respectively. He was a Research Associate in 1978 and an Associate Professor in 1985, at Tokyo Institute of Technology. He is currently Professor at Department of Communications and Integrated Systems in Tokyo Institute of Technology. He has been engaged in researches on Distributed Circuits, Switched Capacitor Circuits, IC Circuit Simulation, VLSI CAD, VLSI Signal Processing and VLSI Design. His current research focuses on fingerprint authentication algorithms, VLSI Multimedia Processing including Video CODEC, Design for System On Chip, VLSI Signal Processing, VLSI Architecture including Reconfigurable Architecture, and VLSI CAD. Prof. Kunieda is a member of IEEE CAS, SP society, IPSJ and IEICE.

(Recommended by Associate Editor: *Kiyoharu Hamaguchi*)

© 2015 Information Processing Society of Japan