PSM: An Object-Oriented Synthesis Approach to Multiprocessor System Design

Pao-Ann Hsiung, Sao-Jie Chen, *Member, IEEE,* Tsung-Chien Hu, and Shih-Chiang Wang

***Abstract*— Although multiprocessor systems are becoming a trend today, yet few synthesis tools currently available can actually automate the design of multiprocessor systems. Performance synthesis methodology (PSM) is an object-oriented system-level synthesis approach to multiprocessor system design. Since PSM was designed specifically for the synthesis of multiprocessor systems, it is not only much more efficient when synthesizing parallel systems, but also produces better parallel systems than currently available uniprocessor system-level synthesis tools. Colored Petri nets used in modeling system components and object modeling technique used in the design process have both contributed to the shortening of system development time and to the reduction of design cost. First, user specification consisting of functional models and performance constraints is translated into architecture models. Then, the system is configured by selecting the method of control, the memory organization, the type of processor, and the type of system interconnection. Finally, a heuristic design space exploration algorithm is used to generate several near-optimal design alternatives. The best architecture is chosen by evaluating the design alternatives using a flexible performance estimation formula that mainly considers system level design features, such as system throughput, utilization, reliability, scalability, fault-tolerance, and cost. Several systems were successfully synthesized using this top-down object-oriented PSM, thus showing its feasibility as a design automation tool for parallel systems.**

***Index Terms*— Colored Petri nets, functional model, heuristic design-space exploration, multiprocessor synthesis, object-oriented hardware design, parallel architecture model, performance synthesis methodology, system-level synthesis.**

I. Introduction

NOWADAYS, as computer-aided design tools become more and more intelligent, they can accumulate experience and techniques of design experts and enable computers to do automatically what only the human designers have been able to do in the past. An example is the use of silicon compilation to automate the design of VLSI chips [1], [2]. Continuous researches have been conducted in this field and considerable progress has been made to change the level of synthesis from a very low (detailed) level in the past to the high-level and system-level syntheses today [3]–[8]. However, practically very little research has been devoted to the synthesis of parallel computer systems. Although the theory behind system-level synthesis is not mature enough, nonetheless an efficient and flexible methodology for the system-level design of parallel computer systems is presented here. This methodology brings together the notions of 1) parallel computer system modeling, such as object-oriented modeling and Petri-net modeling, 2) performance modeling which considers not only throughput, utilization, and cost but also scalability, reliability, and fault-tolerance, and 3) system-level synthesis of parallel computer systems. This new design methodology, called performance synthesis methodology (PSM), assists computer system hardware engineers in developing their designs more easily, thus increasing their productivity, as well as, shortening the time-to-market of a complete system. Section II discusses the motivation of this paper and related previous work. Problem specification and system inputs are described in Section III. Different phases of PSM are explained in Section IV. Section V illustrates the application of the methodology by going through the synthesis of several example systems. The conclusion and some suggested future work are presented in Section VI.

Manuscript received February 3, 1994; revised November 18, 1994 and April 26, 1995. This work was supported by the National Science Council, Taipei, Taiwan, Republic of China, under Grants NSC-83-0408-E002-129 and NSC-83-0404-E002-059.

The authors are with the Department of Electrical Engineering, National Taiwan University, Taipei, Taiwan, R.O.C.

Publisher Item Identifier S 1063-8210(96)01869-0.

II. Motivation and Previous Work

A. Multiprocessor System Design

Multiprocessor (MP) system design differs from traditional uniprocessor system design in terms of its complexity and difficulty. For example, some features exclusive to MP systems are the memory subsystem organization, the memory access conflict resolution, the number of parallel processors, and the type of system interconnection between memory and processors. Presently available synthesis systems, such as the MICON Synthesizer Version 1 (M1) [7]–[9] and the Magellan System [10] which incorporates the System Architect's Workbench [11], WOLFIE [12], and LASSIE [13], do not explicitly take the MP features into consideration during system synthesis; hence, the adoption of such systems for the design of MP systems, might not result in good parallel designs. Although there exist synthesis methodologies which do consider some of the features of MP systems, all of these methodologies have a restricted scope of application, e.g., Mabbs and Forward [14] analyzed the performance of MR-1, a clustered shared-memory MP, using a queuing model and a lost request model; Chiang and Sohi [15] evaluated design choices for Shared-Bus MP's in a throughput-oriented environment using customized mean-value analysis. There are also tools whose targets of synthesis are MP systems, but the application domain has been restricted to some special classes of MPs, for example, Rao and Kurdahi discussed hierarchical design-space exploration for digital-signal processing systems [16].

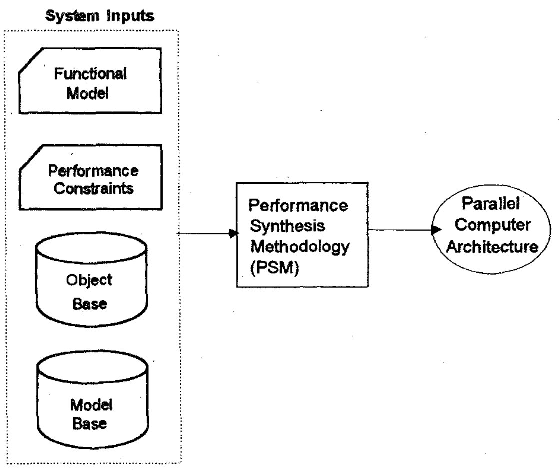


Fig. 1. Performance synthesis methodology (PSM) inputs.

PSM is a synthesis methodology which not only considers many important features of an MP system, but supports a wide range of applications since the target machine under synthesis can be either a general purpose MP system or an application-specific system. An example of a successful system-level synthesis tool is the MICON system [9], but it does not explicitly consider the MP architecture, whereas PSM spends a greater effort on MP configuration at the system level before the actual detailed synthesis. This makes PSM produce better parallel systems.

B. Object-Oriented Synthesis

Technology transfer between software and hardware design has led to the use of object-oriented techniques in hardware design [17]. Indeed, it is in the field of hardware design that the power of object-oriented techniques in modeling and design can really be exhibited. Often, two or more of the same hardware component or subsystem are used in a single system; thus, the concept of reuse in object-oriented techniques comes into play. The static and dynamic features of a hardware component make it fit exactly into the concept of an object with its data and its member functions. Kumar *et al.* [17] used a data decomposition approach to identify reusable components. The incorporation of object-oriented techniques into computer-aided synthesis has been discussed mainly in the literature [17], [18] and implemented in a few hardware description language oriented design tools [19].

PSM incorporates object-oriented techniques into the system-level synthesis of MP systems. Not only is the component library built using an object-oriented classification method, but the actual synthesis process is also object-oriented as it uses object-oriented relations, such as the *aggregation* and the *generalization* relationships, and object-oriented operations, such as the *iterator* and the *generator* to synthesize MP systems.

C. Performance Evaluation

The queuing model [14], lost-request model [14], object-oriented Petri-net model [18], [20], customized mean-value analysis model [15], Markovian model [21], and stochastic Petri-net model [22] are all examples of performance models that have been used for evaluating multiprocessor systems. Since PSM is an object-oriented methodology, the object-oriented Colored Petri-Nets (CPN) model was adopted as our performance modeling tool. For the actual execution and simulation, the SES/Workbench[1], an object-oriented simulation tool, was used. Using instruction-mixes as the simulation workload, system performance results such as throughput and utilization are obtained by performing simulations on the various design alternatives.

III. Problem Specification and Input Descriptions

A. Problem Specification

The performance synthesis problem requires that an architecture-level system be automatically synthesized such that all user-specified functional requirements and constraints on performance and cost are satisfied and the whole synthesis process is completed in a reasonable amount of time. The final architecture-level system output is the result of a heuristic search for the optimal design, with optimality being defined in terms of the best performance which is itself a combination of various system-level performance factors, such as throughput, utilization, cost, reliability, scalability, and fault-tolerance. To solve this problem, a Performance Synthesis Methodology (PSM) is proposed in this paper.

B. System Input

As shown in Fig. 1, four kinds of input information are needed in PSM: 1) functional model, 2) performance constraints, 3) object base, and 4) model base.

1) Functional Model: A functional model is used to describe the overall dynamic and functional view of a system under design. The example of functional model shown in Fig. 2 is actually a combination of the dynamic model and the functional model discussed in Rumbaugh's object modeling technique (OMT) [23]. The purpose of this combination is to eliminate the unnecessary duplication of information that might occur in Rumbaugh's Dynamic and Functional Models.

2) Performance Constraints: PSM basically considers five aspects of system level performance, namely, cost, power, reliability, scalability, and fault-tolerance. They are defined as follows:

a) **Cost:** This is simply the sum of all system component costs.
b) **Power:** It is a ratio of the system (component) throughput to the system (component) utilization. A higher throughput under the same percentage of utilization gives a greater power, while a lower utilization generating the same throughput implies more latent capacity for that particular system and hence, a greater value of power.
c) **Reliability:** The reliability of a system (or component) is the conditional probability that the system (component)

[1] SES/Workbench is a registered trademark of Scientific and Engineering Software, Inc.

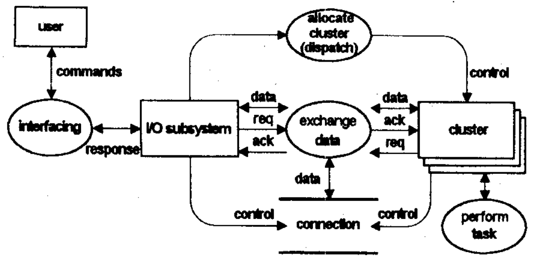


Fig. 2. Functional model of a cluster array processor.

operates correctly throughout the interval $[t_o, t]$, given that it was operating correctly at time t_o.

d) **Scalability:** Scalability is defined as a weighted sum of the percentage of possible increase in the existing hardware subsystems.

e) **Fault-Tolerance:** Fault-Tolerance is defined as a weighted sum of the maximum fraction of each subsystem that is allowed to be faulty before the system fails.

With the above system-level performance factors taken into consideration, a Performance Space (PS) is defined as follows.

3) Performance Space: Performance space is an n-dimensional real space in which each dimension represents one performance factor and n is the number of performance factors considered in the *performance estimation formula* (PEF). Thus, the coordinates of a point in this space give the performance values for some possible design alternatives.

A general form of the Performance Estimation Formula is given as follows:

$$\text{PEF} = \frac{f(x_i^{e_{x_i}})}{g(y_i^{e_{y_i}})} \tag{1}$$

where x_i and y_i are performance factors for $i = 1$ to n, e_{x_i} and e_{y_i} are the respective exponents signifying the importance of a specific performance factor, and f and g are user-defined functions that aid in integrating all the performance factors.

For illustration, a three-dimensional performance space is shown in Fig. 3 with its PEF defined as a 3-factor formula:

$$\text{PEF}_1 = (\text{cost})^2 + \left(\frac{1}{\text{power}}\right)^2 + \left(\frac{1}{\text{reliability}}\right)^2 \tag{2}$$

which is actually the square of the distance of a design point to the origin. Minimizing this geometric distance gives an optimal system in terms of cost, power, and reliability.

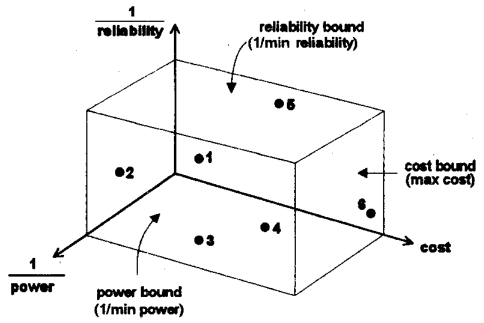
PEF could also be defined as follows:

$$\text{PEF}_2 = \frac{P^{e_P} \times R^{e_R} \times S^{e_S} \times F^{e_F}}{C^{e_C}} \tag{3}$$

where C = cost, P = power, R = reliability, S = scalability, and F = fault-tolerance.

PEF_1 consisting of only three performance factors is used mainly for illustration purpose and the 5-factor PEF_2 formula is actually used in the implementation of PSM.

4) Object Base: Here the term "object" represents a basic component of computer systems. The Object Base is primarily composed of *objects, object classification*, and *object instances*. The classification information of an object is part of


1/reliability
reliability bound (1/min reliability)
5
1
2
cost bound (max cost)
6
4
3
cost
1/power
power bound (1/min power)
Fig. 3. A 3-D performance space.

the static information about the structure of a system, which includes:

a) a unique object class name,
b) the attributes of this object class, including estimated cost and performance,
c) the operations that can be manipulated on this object class, and
d) its relationship to other object classes.

A classification diagram, as shown in Fig. 4, is represented by one or more object class nodes, by links that represent relationships among classes, and by association symbols depicting the type of relationships. For example in Fig. 5, the class `Memory Subsystem` is composed of an optional `Memory Controller` and one or more `Memory Modules`. This type of assembly-component relationship is called an *aggregation relationship*. Since the assembly class is of a higher level than the component classes, the *aggregation relationship* between them can be used to distinguish between two different levels of classes. The *aggregation relationship*, occurring in the object classification, provides necessary information on decomposing a particular component into a lower level. Traversing the classification hierarchy and following such relationship, we can easily synthesize a component by finding the subcomponents having an *aggregation relationship* with this component.

There is another kind of relationship, called *generalization*, which represents the relationship between a superclass and its one or more refined versions of subclasses. For example, the class `Memory Module` in Fig. 5 is the superclass of its subclasses `Random Access Memory` and `Read-Only Memory`. Generalization relationship is mainly useful for system configuration and design space exploration. This type of relationship provides the information that pertains to the different choices a designer has when implementing a component, (s)he can thus make a quick selection by going through each specialization subclass.

5) Model Base: The model base consists of all the models required for synthesis, namely, the system-level *architecture models* and the *object models*.

The degree of the sharing of system resources, e.g., system memory, system interconnection, etc., is much higher in parallel computer systems than in uniprocessor systems. Memory access conflicts in shared-memory parallel computer systems tend to degrade their overall performance, e.g.,

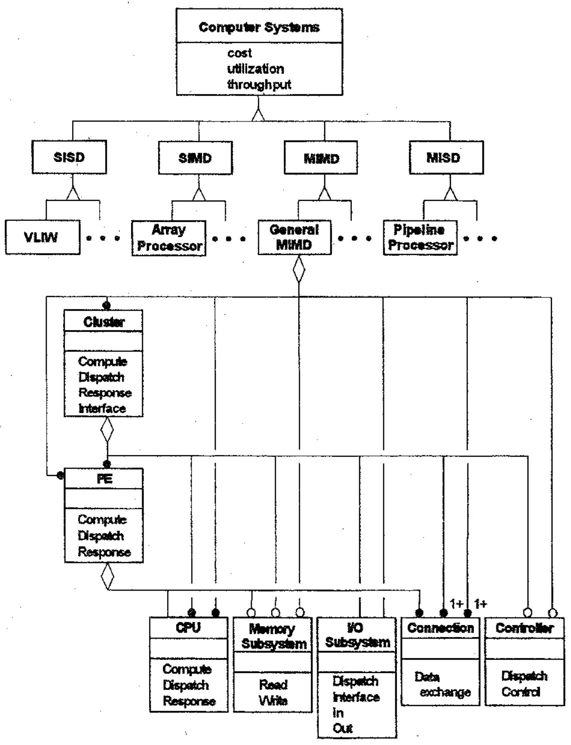


Fig. 4. Object classification.

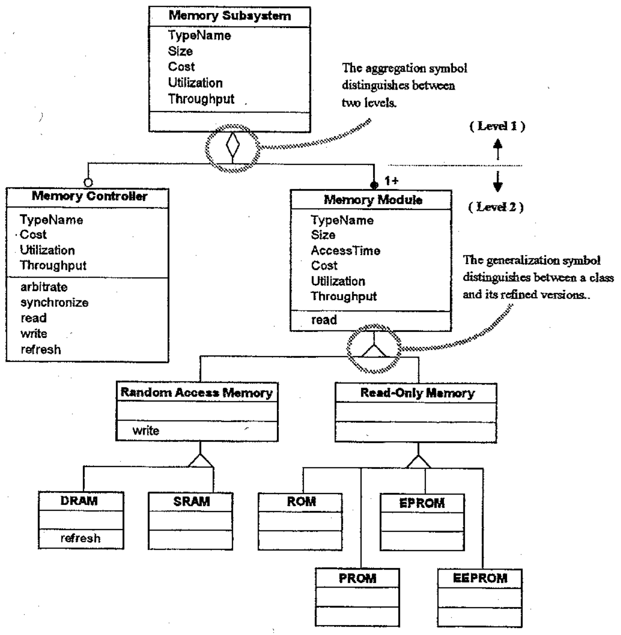


Fig. 5. Multilevel classification diagram of a memory subsystem.

throughput, response-time, power, etc., hence, PSM adopts a classification of parallel computer architectures based on different memory-access latencies, which basically consists of four kinds of system architectures: UMA (Uniform Memory Access), NUMA (Non-Uniform Memory Access), COMA (Cache-Only Memory Access), and NORMA (NO-Remote

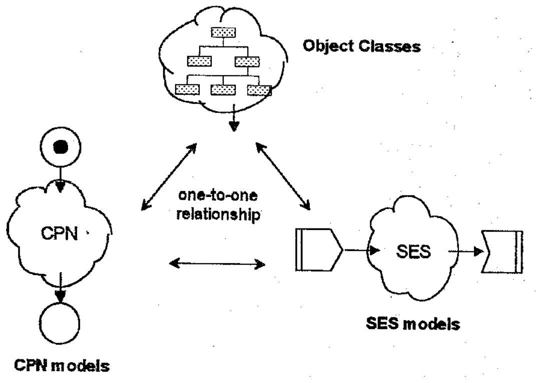


Fig. 6. Relationship between a CPN model and an SES model.

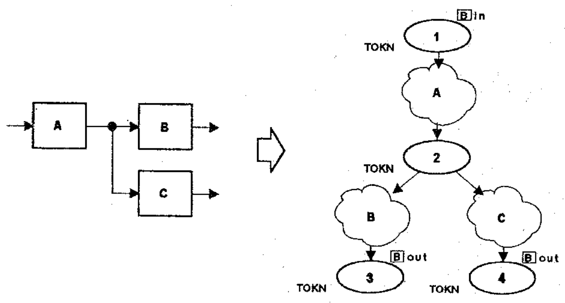


Fig. 7. Combination of CPN building blocks.

Memory Access) [24]. All information pertaining to these system architectures is stored as *architecture models* in the model base. For example, the type of system interconnections (shared bus, multistage interconnection network, crossbar, etc.), the memory placements (global or local or both), and the memory arrangements (centralized or distributed) appropriate for each system architecture are stored in each corresponding *architecture model.*

As shown in Fig. 6, each component object in the architecture model has two corresponding object models: a colored Petri-net (CPN) model and a SES/Workbench model [25], [26]. Colored Petri-net model is suitable both for users to model objects, as well as, for model analysis [27]–[29], while SES/Workbench is a convenient tool for model implementation and simulation. Both models have uniform input and output interface specifications so that two or more submodels can be combined into a larger model in a very intuitive way. Fig. 7 shows how the combination of building-blocks A, B, and C can be used to form a larger system. Figs. 8 and 9 show the CPN and SES/Workbench models of a CPU, respectively. All the information pertaining to these CPN and SES/Workbench models are stored as *object models* in the model base.

IV. Performance Synthesis Methodology

The PSM methodology consists of three major design phases: *map to architecture models phase, generate system configuration phase*, and *synthesize architecture phase*, as shown in Fig. 10. The following subsections discuss these three phases and the top-down topology used in PSM.

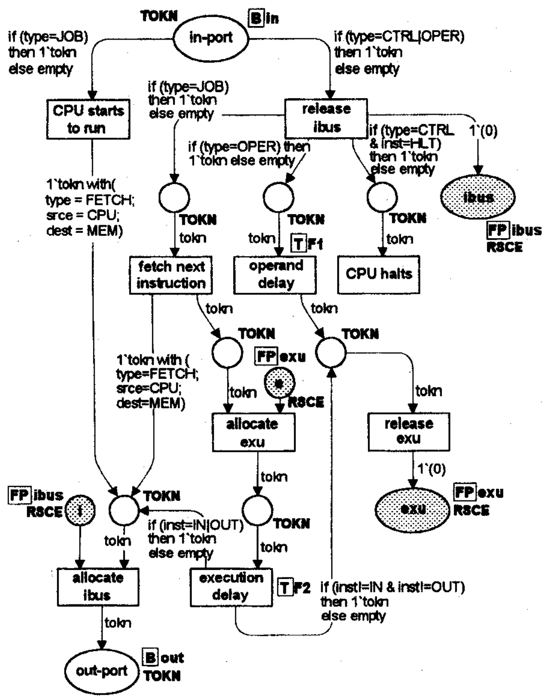


Fig. 8. CPN model of a CPU object.

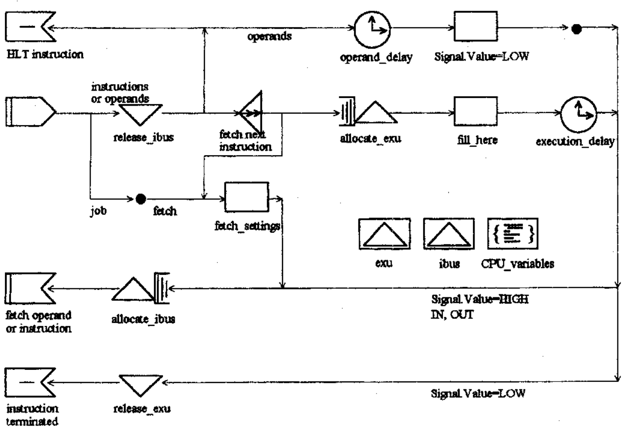


Fig. 9. SES model of a CPU object.

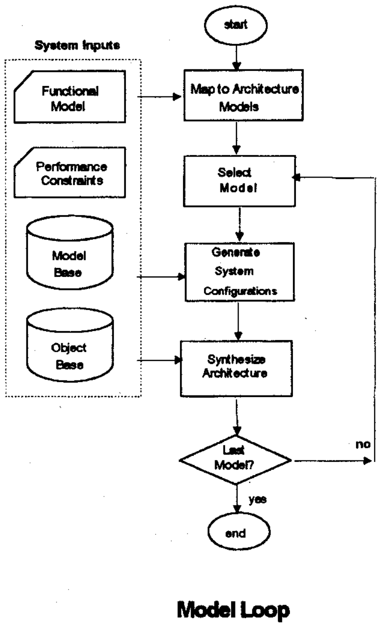


Fig. 10. Performance synthesis methodology (PSM).

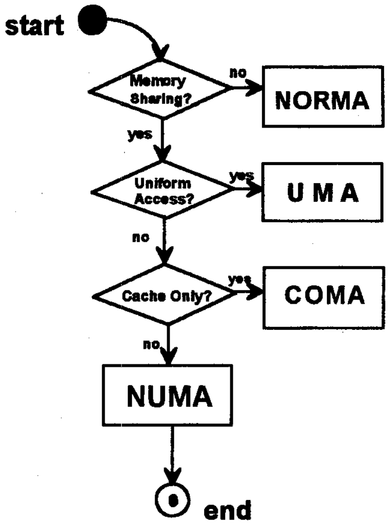


Fig. 11. Detailed flowchart of the "map of architecture models" phase.

A. Map to Architecture Models Phase

A given Functional Model input has first to be mapped onto one of the four basic *architecture models* from the Model Base, i.e., UMA, NUMA, COMA, and NORMA as shown in Fig. 11. This mapping should follow the Functional Model specification described by the user. Since UMA, COMA, and NUMA are shared-memory architecture models whereas NORMA does not allow memory sharing, the Functional Model is first checked to determine whether it is memory sharing. If there is no memory sharing at all, then the NORMA model is adopted. It can also be observed that UMA and COMA are special cases of NUMA, hence, the UMA and COMA models should be considered before the more general NUMA model. When memory sharing is required, a uniform sharing suggests the use of the UMA model, whereas a non-uniform memory access latency requires a further check to see whether the memory subsystem consists of cache. For a cache memory subsystem, the COMA model is used; otherwise, the general NUMA model is considered. The above mapping will generate many different feasible models, each of which can be subsequently selected for further system configuration generation and architecture synthesis.

B. Generate System Configurations Phase

As shown in Fig. 12, once a particular architecture model is selected, the aggregation and generalization relationships of its components stored in the Object Base are used to derive a

```
Algorithm Design_Space_Exploration( )
begin
    $SUO$ ← Generate_Useful_Objects( );          /* set of useful object classes */
    $SUM$ ← { };                                   /* set of useful modules */
    for each set of useful object classes $suo \in SUO$ do
    begin
        $m$ ← Generate_Useful_Modules($suo$);
        $SUM \leftarrow SUM \cup m$;
    end
end.
```

Algorithm 1

global design of the system. According to the classification information stored in the Object Base, a system can be configured in terms of the types of control to be implemented in the system (e.g., SIMD or MIMD type of control), the possible memory module arrangements (e.g., fully global, or partially global and partially local, or only local), the suitable processor types (e.g., fine-grained or coarse-grained processors), and the possible system interconnections (e.g., shared global bus, multistage interconnection network (MIN), or directly connected networks like hypercube).

The above process of generating system configurations is also constrained by the architecture model selected, for instance, the information provided by the model base and those pertaining to the selected architecture model are used as an initial guide for system configuration. For example, the Shared Bus would not be considered as the system interconnection if the architecture model under consideration was NORMA because there is no memory sharing involved in NORMA; alternatively, there would not be any other memory besides cache if the model selected was COMA. Besides these considerations, the performance information of each object component, such as throughput, cost, etc. which is stored in the Object Base, could be another guideline for generating system configurations.

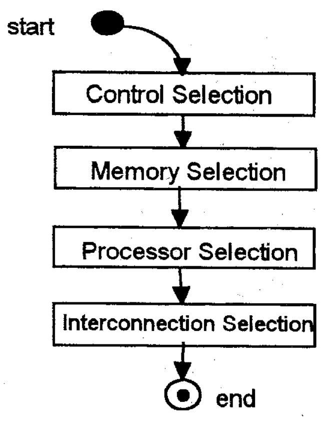


Fig. 12. Detailed flowchart of the "generate system configurations" phase.

C. Synthesize Architecture Phase

The above discussed "generate system configuration" phase was a global design of the system and the "synthesize architecture" phase, described in this subsection, is basically a detailed design of the system. This latter phase, as shown in Fig. 13, involves a loop over all the possible combinations of system configurations. After a queue is initialized, a heuristic design-space exploration (DSE) algorithm (see Fig. 14 and Algorithm 1) is used to explore a part of the design-space that is considered to be the most-optimal-possible region under the heuristic, to select a small number of feasible alternative designs, and to pick out the best among them. The use of heuristic saves considerable design time and cost by restricting the search to a part of the design-space and at the same time it ensures near-optimal results.

When a designer describes a computer system using the Functional Model, three types of nodes are specified: *processes* (functions), *actors* (active objects), and *data stores* (passive objects). With this knowledge in mind, some abbreviations are introduced as follows.

FNC: the set of functions defined by *processes* in the functional model.

OBJ: the set of object classes defined by *actors* and *data stores* in the functional model.

SOF(fnc): the set of object classes selected from the Object Base that can be used to support a function $fnc \in FNC$.

SCO: the set of critical object classes in *OBJ*, where a critical object class is the object class supporting a function $fnc \in FNC$ that cannot be supported by any other object class in *OBJ*.

SNCO: the set of non-critical object classes in *OBJ*.

1) Generate Useful Objects: The first subphase of DSE known as "generate useful objects" is shown in Fig. 14 and Algorithm 2. The "set of useful object classes" (*SUO*) is formed as a result of this subphase. *SUO* is a set whose elements are collections of object classes. Repetition of object classes in an element of *SUO* is not permitted. Each element of *SUO, suo*, is also a set; this set is a collection of object classes from the object base, containing the components of an alternative design and having the following properties:

a) *Feasibility:* For each function $fnc \in FNC$, there exists an object class $obj \in suo$ such that *fnc* is supported by *obj*, i.e., $obj \in SOF(fnc)$.
b) *Optimality:* Two object classes, supporting similar functions specified in *FNC* but having different performance

values, should not belong to the same *suo.* The class with the poorer performance should be deleted from *suo.*

c) *Acceptability:* For an acceptable design, the total cost of each *suo* must be lower than the user-specified Cost Constraint, its estimated power must be greater than or equal to the Power Constraint, and all the other performance constraints, if any, must also be satisfied.

Using Algorithm 2, an *SUO* is generated as follows: First, for each function *fnc* in the Functional Model, an *SOF(fnc)* is formed, where object classes with equal functions are compared and those having poorer performance are deleted. Then, *OBJ* which is initialized to contain all object classes that implement the *actors* and *data stores* in the functional model, is partitioned into a critical set *SCO* and a noncritical set *SNCO.* Finally, elements of an *SUO* are formed by the union of the *SCO* and some elements of *SNCO,* each of which should support all the functions required in *FNC.*

For example, an SUO may look like this: $SUO = \{\{\mathtt{A,B,C}\},\{\mathtt{A,B,D,E}\}\}$.

```
Algorithm Generate_Useful_Objects( )
begin
    /* Step 1: Initialize $FNC$ and $OBJ$ */
    $FNC \leftarrow$ the set of functions defined by processes in the Functional Model;
    $OBJ \leftarrow$ the set of object classes defined by actors and data stores in the Functional Model;
    /* Step 2: check all object classes in $OBJ$ to pick out improper ones */
    for each function $fnc \in FNC$ do
    begin
        $SOF(fnc) \leftarrow$ the set of object classes defined in the Object Base that can be used to support the function
                 $fnc$;
        for each $obj \in SOF(fnc)$ do
            if (current level cannot be composed of $obj$) then $SOF(fnc) \leftarrow SOF(fnc) - \{obj\}$;
        for each pair $(obj_1, obj_2) \in \{(x,y) | x,y \in SOF(fnc), x \neq y\}$ do
        begin
            $fnc_1 \leftarrow$ the set of functions supported by object class $obj_1$;
            $fnc_2 \leftarrow$ the set of functions supported by object class $obj_2$;
            if $((fnc_1 \cap FNC) = (fnc_2 \cap FNC))$ then
                if (Performance($obj_1$) $\geq$ Performance($obj_2$))
                then $SOF(fnc) \leftarrow SOF(fnc) - \{obj_2\}$          /* $obj_1$ is better */
                else $SOF(fnc) \leftarrow SOF(fnc) - \{obj_1\}$;         /* $obj_2$ is better */
        end
        $OBJ \leftarrow OBJ \cup SOF(fnc)$;
    end
    /* Step 3: Initialize $SUO, SCO$ and $SNCO$ */
    $SUO$   $\leftarrow \{\ \}$;  /* set of useful object classes */
    $SCO$   $\leftarrow$ the set of critical object classes in $OBJ$;
    $SNCO$  $\leftarrow$ the set of non-critical object classes in $OBJ$;
    $PWR$   $\leftarrow$ the power set of $SNCO$;
    /* Step 4: combine $SCO$ and each subset of $SNCO$ */
    for each $pwr \in PWR$ do
    begin
        $TMP \leftarrow SCO \cup pwr$;
        if (($\forall fnc \in FNC, \exists$ object class $obj \in TMP$ such that $fnc$ can be implemented by $obj$)
           and (Performance($TMP$) $\geq$ Performance_Bound)) then $SUO \leftarrow SUO \cup \{TMP\}$;
    end
    /* Step 5: return the $SUO$ */
    return($SUO$);
end.

Performance (module)
begin
    return$\left(\frac{\text{module-power} \times \text{module-reliability} \times \text{module-scalability} \times \text{module-fault-tolerance}}{\text{module-cost}}\right)$;
end
```

Algorithm 2

This means that the system is composed of object classes A, B, and C, or of object classes A, B, D, and E. Both combinations match the user specifications, i.e., they support all functions described by the user in the functional model. Object classes A through E may be any computer components, such as CPU's or memory modules. Note that replication of classes is not allowed here. This is different from the concept of useful modules described in the following subsection.

2) Generate Useful Modules: Each element $suo \in SUO$ contains enough object classes to implement all the function specifications of a given system. However, designs using components from *SUO* can only be developed into serial ones (from a hardware point of view), because classes in *suo* are single ones without repetition. To explore the possible multiplicity of design components, we must permit duplication when using these classes. Therefore, in the second subphase of DSE, we use a `Generate_Useful_Modules` algorithm (see Fig. 14 and Algorithm 3), to derive the "set of useful modules" (*SUM*) from *SUO*. Since repetition of object classes is allowed in *SUM*, its elements are no more "sets" in the restrictive sense.

Algorithm 3 is based on the following idea: recalling that the five performance factors we defined were cost, power, reliability, scalability, and fault-tolerance, one can observe an interesting phenomenon in hardware synthesis: *the closer the cost of design is to the cost bound, the better is the performance results in terms of power and reliability (and other performance measures, if any).* Furthermore, an exhaustive search through the entire design-space is time-consuming and cost-wasting. Thus, why not explore only a small part of the design-space

```
Algorithm Generate_Useful_Modules ($suo$)
begin
    $n \leftarrow$ dimension($suo$);        /* the number of elements in set of useful object $suo$ */
    $m \leftarrow$ the set of solutions to the problem [■] stated below;
return ($m$);
end.
```

[■] Problem Statement:

To find $2 \cdot n$ points closest to the intersection point P of a hyperplane S and a normal line L where

$$S\colon c_1x_1 + c_2x_2 + \cdots + c_nx_n = \text{Cost Bound}$$

and the normal line L of S passes through the origin point O,

$$L\colon \frac{x_1}{c_1} = \frac{x_2}{c_2} = \cdots = \frac{x_n}{c_n}$$

Each of the above $2 \cdot n$ points stands for an alternative module design and thus must agree with the following constraints:

1. the values of $x_1, x_2, \cdots, x_n$ are integers.

$$2.\ c_1x_1 + c_2x_2 + \cdots + c_nx_n \le \text{Cost Bound}$$

where c_a: the cost of object class obj_a per unit, and
x_a: the multiplicity of object class obj_a, for $a = 1$ to n.

$$3.\ \frac{\min(t_ax_a)}{\left(\frac{u_1}{\beta}x_1 + \frac{u_2}{\beta}x_2 + \cdots + \frac{u_n}{\beta}\right)} \ge \text{Power Bound}$$

where u_a: the utilization of object class obj_a per unit for $a = 1$ to n,
$\beta = \sum_{a=1}^{n} x_a$
t_a: the throughput of object class obj_a per unit for $a = 1$ to n; and
the function $\min(t_ax_a)$ returns the minimal value among $t_1x_1, t_2x_2, \cdots, t_nx_n$.

$$4.\ \Pi(1 - (1 - R_a(t))^{x_a}) \ge \text{Reliability Bound}$$

where R_a: the reliability of object class obj_a, for $a = 1$ to n.

$$5.\ \sum_{a=1}^{n} f_a \times \frac{d_a}{x_a} \ge \text{Fault-Tolerance Bound}$$

where d_i = number of maximum allowable faulty object class obj_i
f_a = importance factor for the object class obj_a

$$6.\ \sum_{a=1}^{n} f_as_a \ge \text{Scalability Bound}$$

where s_a = scalability of object class obj_a, and
f_a = importance factor for object class obj_a.

Algorithm 3

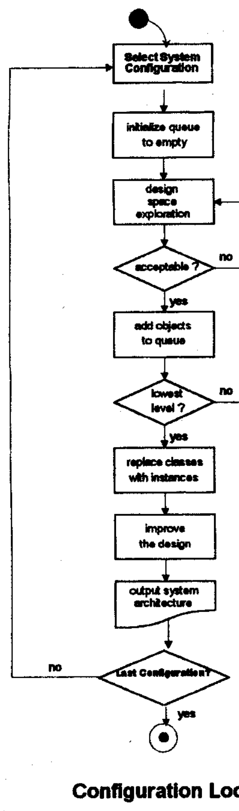


Fig. 13. Detailed flowchart of the "synthesize architecture" phase.

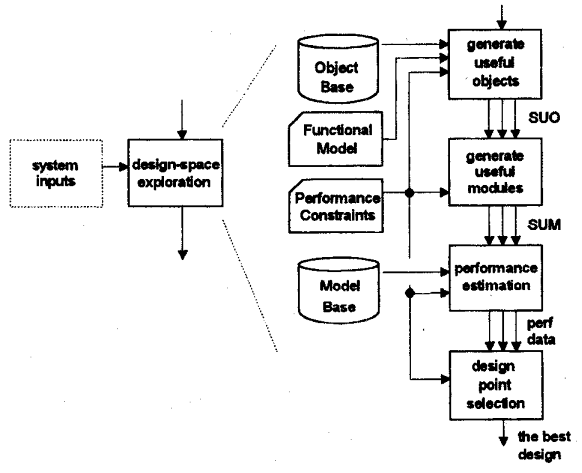


Fig. 14. Detailed flowchart of the "design space exploration" in Fig. 13.

that is close to the cost bound? This strategy should give us several feasible and acceptable designs with good performance and near-optimality.

The above idea is formalized in the statement [■] of Algorithm 3. Design points close to the intersection point of the Cost Bound plane and its normal passing through the origin are considered to be near-optimal design candidates. Consider again the example given in the first subphase:

$$SUO = \{\{\mathtt{A},\mathtt{B},\mathtt{C}\},\{\mathtt{A},\mathtt{B},\mathtt{D},\mathtt{E}\}\}.$$

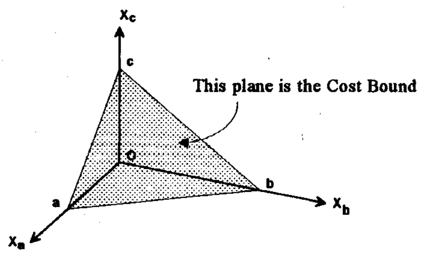


Fig. 15. Feasible design-space for $suo_1 = \{\mathtt{A}, \mathtt{B}, \mathtt{C}\}$.

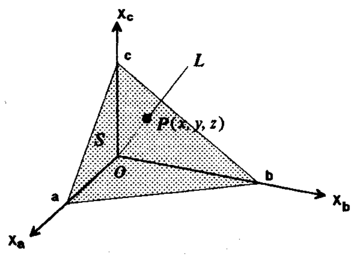


Fig. 16. Intersection point.

For the first element of *SUO*, $suo_1 = \{\mathtt{A},\mathtt{B},\mathtt{C}\}$, we can formulate an equation:

$$c_a x_a + c_b x_b + c_c x_c \leq \text{Cost Bound} \tag{4}$$

where

c_n: the cost of object class obj_n (constant),
x_n: the multiplicity of object class obj_n (variable), and
Cost Bound is a constant value given by the user.

Graphical representation of (4) is a three-dimensional coordinate space as shown in Fig. 15 where $a = \text{Cost Bound}/c_a$, $b = \text{Cost Bound}/c_b$, and $c = \text{Cost Bound}/c_c$.

As far as cost is concerned, all points interior to the pyramid $\mathtt{Oabc}$ with positive integer coordinates are feasible solutions, with each point representing an alternative design. For example, if the point (1, 2, 2) is inside the pyramid $\mathtt{Oabc}$, this point will represent a design composed of one object class $\mathtt{A}$, two object classes $\mathtt{B}$'s, and two object classes $\mathtt{C}$'s. The pyramid $\mathtt{Oabc}$ is in fact the entire design space.

As mentioned before, an exhaustive search for the optimal design through the entire design-space is a very time-consuming and tedious effort. Therefore, the search is restricted to a portion of the design-space by looking at only those integer points close to the intersection point $\boldsymbol{P}(x, y, z)$ of the cost bound plane $\boldsymbol{S}$ and the normal line $\boldsymbol{L}$ of $\boldsymbol{S}$ which passes through the origin $\boldsymbol{O}$, as shown in Fig. 16.

Let the normal vector of plane $\boldsymbol{S}$ be (c_a, c_b, c_c) and since $\boldsymbol{P}(x, y, z)$ is a point on the normal line, we then get:

$$(c_a, c_b, c_c) = k \cdot (x, y, z) \tag{5}$$

where k is a constant,

$$\Rightarrow c_a \cdot \frac{c_a}{k} + c_b \cdot \frac{c_b}{k} + c_c \cdot \frac{c_c}{k} = \text{Cost Bound} \tag{6}$$

$$\Rightarrow k = \frac{c_a^2 + c_b^2 + c_c^2}{\text{Cost Bound}} \tag{7}$$

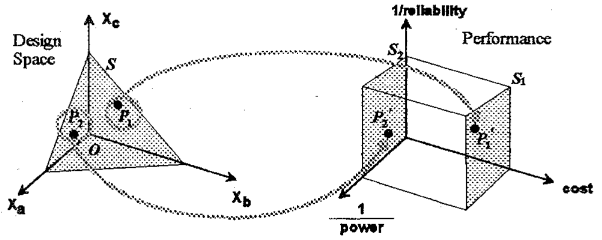


Fig. 17. Relationship between design and performance space.

$$\Rightarrow (x,y,z) = \left(\frac{c_a \cdot (\text{Cost Bound})}{c_a^2+c_b^2+c_c^2}, \frac{c_b \cdot (\text{Cost Bound})}{c_a^2+c_b^2+c_c^2}, \frac{c_c \cdot (\text{Cost Bound})}{c_a^2+c_b^2+c_c^2}\right). \quad (8)$$

Thus, the coordinates of point P can be calculated using (8). Some of the points with integer coordinates close to P are what we are looking for.

3) Performance Estimation: Recall that each alternative design has a representing point in the three-dimensional performance space of Fig. 3. Combining this with the concepts of Figs. 15 and 16, we have Fig. 17 and the following conclusions:

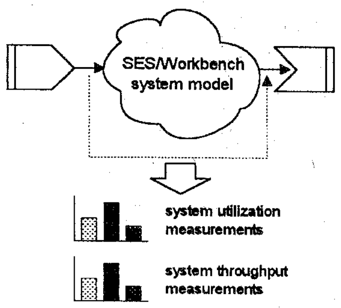
a) the design points close to the cost bound plane S in a design space are equivalent to the points close to the cost bound plane S_1 in a performance space (e.g., P_1 and P'_1);
b) the design points close to the origin O in a design space are equivalent to the points close to zero-cost plane S_2 in a performance space (e.g., P_2 and P'_2).

Each element of *SUM* generated in Algorithm 3 represents an alternative design. For each design alternative $sum \in SUM$, an executable model is built using the SES/Workbench simulation models in the model base. As shown in Fig. 18, each design is simulated to generate two kinds of performance data: system utilization and system throughput whose ratio gives the system power. The calculation of other performance factors, such as system reliability, scalability, and fault-tolerance, has also been shown in Algorithm 3.

4) Design Point Selection: After performance estimation, five pieces of performance data were obtained for each alternative design: cost, power, reliability, scalability, and fault-tolerance. Using this data, it is possible to plot each design as a point in a five-dimensional performance space using the performance evaluation formula (3) as defined in Section III-B2), and pick the point closest to the origin as the best design.

D. Top-Down Topology

As mentioned before, PSM starts with higher level descriptions and ends with lower level architecture schematics. This high-level-to-low-level synthesis topology is termed "top-down" topology. The three design phases of PSM, described in the previous three subsections, systematically synthesize a multiprocessor system from a user-given *functional model*



SES/Workbench system model
system utilization measurements
system throughput measurements

Fig. 18. Performance estimation by simulation.

specification. The *map to architecture models phase* maps a *functional model* into an architecture model which is then consolidated by generating system configurations in the *generate system configuration phase.* The global design obtained in the second phase is further manifested, in the three subphases of the *synthesize architecture phase*, into an actual multiprocessor architecture. The full design scheme is top-down and this is observed particularly in the last phase of PSM where the desired system is iteratively synthesized using a heuristic design space exploration algorithm. In what follows, we will discuss this topology concept by giving an example, the hierarchy of which is limited to three levels for ease of discussion.

Assume a simple computer system is to be synthesized. It is the aggregation type of relationship that first comes into play during this top-down synthesis. Since all aggregation relationships between objects classes have been stored in the Object Base, one can immediately obtain the lower level subcomponents by just following the aggregation relationships in the object classification. This operation is performed by the operator known as "iterator" [30]. Similarly, corresponding to the *generalization relationship*, there is the "generator" operator, that is used during design-space exploration, to select one class from several alternative and refined subclasses of a particular object class. During the synthesis process, a queue is used to hold all information that pertain to the building-blocks waiting to be synthesized into more basic components. As shown in Fig. 19, in Step 1, there is a level-1 block representing the computer system itself, and the queue is initialized to empty. By referring to the object classification in the Object Base and applying the iterator operator, the computer system is synthesized into four components: the CPU, the bus, the I/O subsystem, and the memory subsystem. These level-2 components are appended to the queue waiting for further synthesis. In Step 3, the CPU is popped from the queue, and synthesized to consist of the ALU, the internal bus, the registers, and the timing controller. At the same time, the queue is updated to include these new components. The bus need not be further synthesized because it is already a basic component. Next the I/O and memory subsystems are synthesized in Steps 4 and 5, respectively. After Step 5, all building-blocks contained in the queue are basic components, therefore the synthesis process stops. The three diagrams as shown in Steps 3, 4, and 5 of Fig. 19 form a complete architectural design in the lowest level (level-3).

V. Application Examples

In this section, some PSM applications are described by going through the synthesis process of several large example systems. Three different examples are given. The first system synthesized is the *cluster array processor (CAP) system* which is a multiprocessor digital signal processing system. This example gives an overview of the multiple levels in the synthesis procedure. The second *sorting* example mainly illustrates the design choices made during the design-space exploration phase and the simulation of different design alternatives. The third example summarizes the four *multiprocessor systems* that PSM has successfully synthesized.

A. Cluster Array Processor

Cluster array processor (CAP) is a typical and simple multiprocessor system with a hierarchical cluster architecture. It has been used as an example for performance modeling and fault modeling [31]. Hence, we choose CAP as our first illustration.

1) User Specification: Assume that a user or a system designer has specified his/her requirements by drawing Functional Model diagrams as shown in Fig. 20.

2) Performance Constraints: Total Cost Bound = \$200.

3) Map to Architecture Models: Since there is no memory sharing in the given functional model [Fig. 20(a)], a NORMA model is adopted.

4) Generate System Configurations: Based on the NORMA model, either an SIMD or an MIMD control method could be adopted.

5) Synthesize Architecture: The hierarchy is limited to three levels in order to simplify the discussion. Let the cost of a Cluster be \$80. This implies that at most two Clusters can be used. The few processors and the low cost bound also imply that the use of at most two shared buses for system interconnection is enough. At level-1 of the synthesis process, on applying the Design Space Exploration algorithm (Algorithm 1) to the above system, we have the following results as shown at the bottom of the page.

The resulting *SUM* generated from *SUO* by Algorithm 3 is given below:

$$
\begin{aligned}
SUM = \{sum_1 &= \{\text{connection, I/O subsystem, clustercluster, cluster}\},\\
sum_2 &= \{\text{connection, I/O subsystem, cluster}\},\\
sum_3 &= \{\text{connection, connection, I/O subsystem, cluster, cluster}\},\\
sum_4 &= \{\text{connection, I/O subsystem, cluster, cluster, controller}\},\\
sum_5 &= \{\text{connection, I/O subsystem, cluster, controller}\},\\
sum_6 &= \{\text{connection, connection, I/O subsystem, cluster, cluster, controller}\},
\end{aligned}
$$

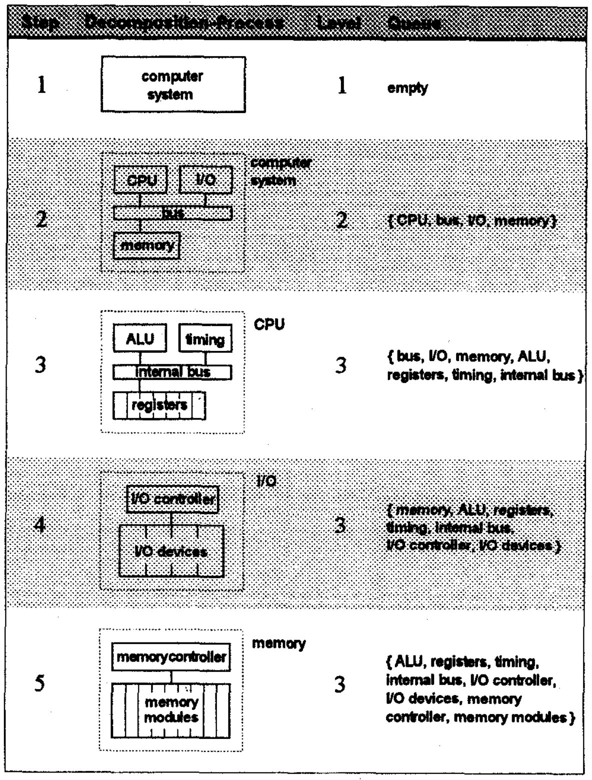

Step
Decomposition Process
Level
Queue
1
computer system
1
empty
2
CPU
I/O
bus
memory
computer system
2
{CPU, bus, I/O, memory}
3
ALU
timing
internal bus
registers
CPU
3
{bus, I/O, memory, ALU, registers, timing, internal bus}
4
I/O controller
I/O devices
I/O
3
{memory, ALU, registers, timing, internal bus, I/O controller, I/O devices}
5
memory controller
memory modules
memory
3
{ALU, registers, timing, internal bus, I/O controller, I/O devices, memory controller, memory modules}

Fig. 19. Top-down decomposition.

$$
\begin{aligned}
&FNC && = \{\text{interfacing, exchange data, dispatch, perform task}\}\\
&OBJ && = \{\text{I/O subsystem, connection, cluster}\}\\
&SOF\ (\text{interfacing}) && = \{\text{I/O subsystem, cluster}\}\\
&SOF\ (\text{exchange data}) && = \{\text{connection}\}\\
&SOF\ (\text{dispatch}) && = \{\text{I/O subsystem, cluster, controller, PE, CPU}\}\\
&SOF\ (\text{perform task}) && = \{\text{cluster, PE, CPU}\}\\
\Rightarrow\ &OBJ && = \{\text{I/O subsystem, connection, cluster, controller, PE, CPU}\}\\
&SCO && = \{\text{connection, I/O subsystem, cluster}\}\\
&SNCO && = \{\text{controller, PE, CPU}\}\\
\Rightarrow\ &SUO && = \{suo_1 = \{\text{connection, I/O subsystem, cluster}\},\\
& && \quad suo_2 = \{\text{connection, I/O subsystem, cluster, controller}\}\}
\end{aligned}
$$

where $\Rightarrow$ represents different execution sub-phases of Algorithm 2.

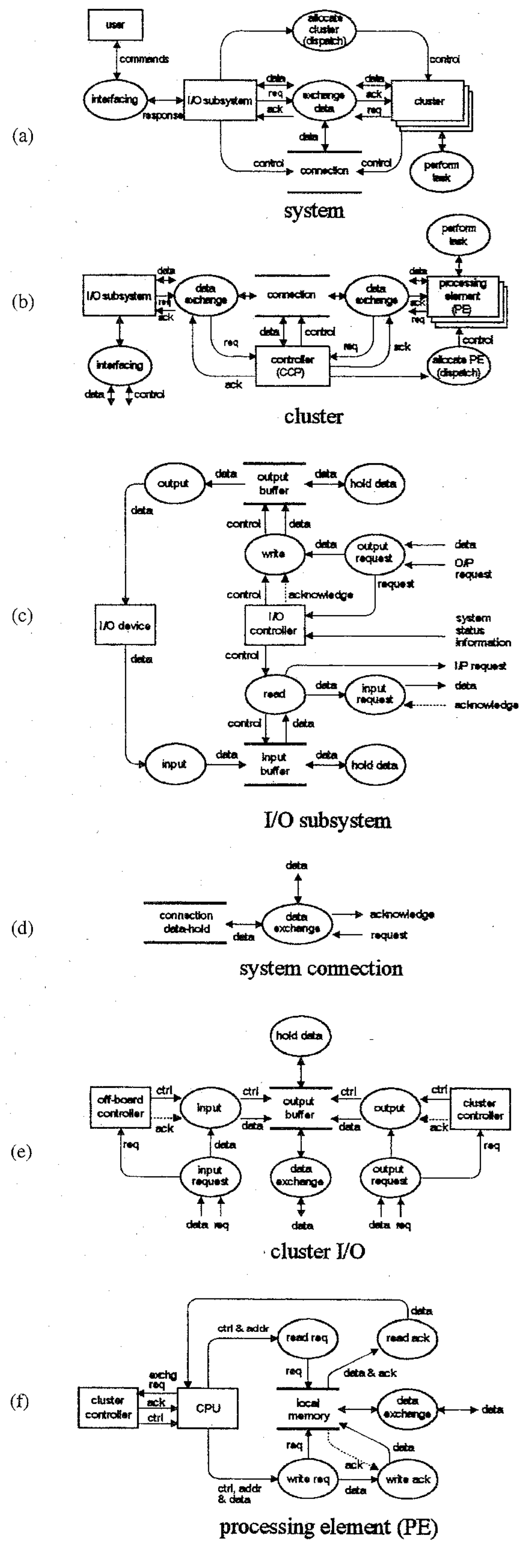


Fig. 20. Functional models describing the components being synthesized.

$$sum_7 = \{\text{connection, connection, I/O subsystem, cluster, controller}\}\}.$$

TABLE I
PERFORMANCE EVALUATION RESULTS

sum #	Cost (C)	Throughput (T)	Utilization (U)	Power (P=T/U)	Reliability (R)	Fault Tolerance (F)	Scalability (S)	Distance (1)	Distance (2)
sum_1	175	0.1	0.41	0.24	0.89	0.33	0.28	3.0132	2.91
sum_2	95	0.09	0.58	0.16	0.85	0.17	0.44	3.53	2.82
sum_3	180	0.1	0.39	0.25	0.98	0.5	0.22	2.98	3.99
sum_4	185	0.1	0.39	0.25	0.84	0.33	0.25	3.14	2.49
sum_5	105	0.1	0.57	0.17	0.8	0.17	0.44	3.2398	2.47
sum_6	190	0.1	0.4	0.24	0.93	0.5	0.17	3.3022	2.59
sum_7	110	0.1	0.58	0.17	0.88	0.17	0.33	3.2349	1.93

Then SES/Workbench simulation models of all the above design alternatives are executed and their performance results are recorded in Table I. Each design alternative is evaluated using the following two different performance metrics:

$$\text{distance}_1 = \sqrt{C^2 + P^{-2} + R^{-2} + F^{-2} + S^{-2}},$$
$$\text{distance}_2 = \frac{P \times R \times S \times F}{C}$$

where P = performance, R = reliability, S = scalability, F = fault-tolerance, and C = cost.

The first distance is in fact a geometric distance which, when minimized, will give the point closest to the origin and hence the best design. The second distance is a direct product of the various performance factors, and maximizing this distance will give the best design point.

It is concluded from Table I, that sum_3 is the best choice. The component queue status is now {cluster, cluster, I/O subsystem, connection, controller} [Fig. 21(a)], Moving on to level-2, the next component to be synthesized is the cluster. Algorithm 1 is executed again with the Functional Model of a cluster as input [Fig. 20(b)]. The synthesized architecture of this cluster is shown in Fig. 21(b).

The above procedure is repeated for the *I/O subsystem* and the *system connection* in level-2 and for the *cluster I/O*, and the *processing element* (*PE*) in level-3. The respective Functional Models are shown in Fig. 20 (c), (d), (e), and (f), and their synthesized architectures in Fig. 21 (c), (d), (e), and (f). At this point, the synthesis process stops as the hierarchy is limited to three levels, thus all the components left in the component queue are considered to be basic physical components. The above synthesis procedure is repeated for each system configuration. A final evaluation of all synthesized architectures gives the best overall architecture.

B. Sorting Example

In this example, we assume that the user has specified a sorting problem in the Functional Model input.

1) Problem: Use shared-memory to sort 8×10^9 records. Each record has two randomly generated keys: a primary and a secondary key.

2) Performance Constraints: CPU benchmark: SPECint92 at least 80 and SPECfp92 at least 90, Total memory = 128 MB and Total cost bound = $100 000.

3) Map to Architecture Models: Since memory sharing is required and there is no nonuniformity in memory access, an UMA model is adopted.

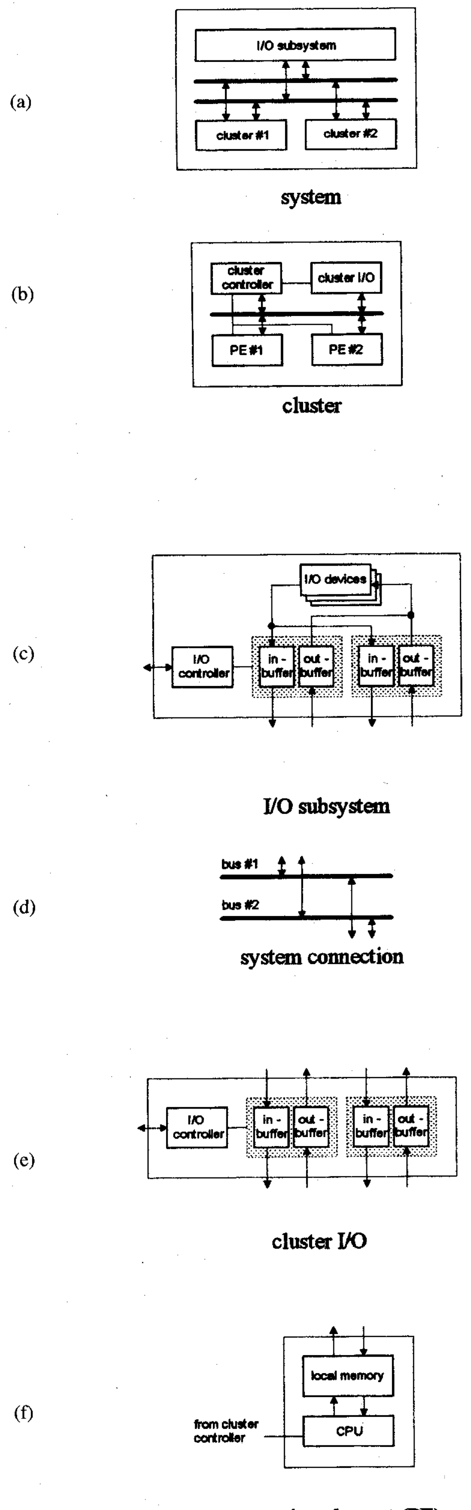


Fig. 21. Synthesized architecture schemes.

4) Generate System Configurations: Since sorting can be decomposed into a collection of similar tasks, i.e., subsequence sorting, the SIMD method of control is chosen. System interconnection can be either shared-bus or MIN.

TABLE II
PROCESSOR FAMILIES AND RELATED INFORMATION (SOURCE: IEEE SPECTRUM DEC. 1993 AND IEEE COMPUTER JUNE 1994)

• Control circuitry and other costs = $700.

Company	Intel Corp.		Sun Micro. Corp. & Texas Instr. Inc.	Hewlett-Packard Co.	MIPS Technology Inc.	IBM Corp. & Motorola Inc.		Digital Equipment Corp.	
Features/CPU	Pentium		Super SPARC	Precision Arch.	MIPS	PowerPC		Alpha Chip	
Model	561	735	—	PA-7100	R4400SC	601	603	21,064	21,066
Type	CISC		RISC	RISC	RISC	RISC		RISC	
Clock (MHz)	66	90	60	100	150	80	80	200	166
SPECint92	67.4	90.1	80	81	88	85	75	130	70
SPECfp92	63.6	72.7	100	150	97	105	85	184	105
Technology	0.8 um BiCMOS		0.7 um BiCMOS	0.8 um CMOS	0.6 um CMOS	0.65 um CMOS		0.68 um CMOS	
US $/1000	898	NA	999	NA	1,100	557	NA	505	NA

TABLE III
SYSTEM CONFIGURATIONS FOR EXAMPLE 2

CPU types & Interconnection Networks	Number of CPUs with			Total system cost	
	1 Shared Bus	2 Shared Bus	MIN	1 Shared Bus	2 Shared Bus or MIN
Super SPARC	93	91	91	98,407	96,409
PA-7100	93	91	91	98,500	96,500
MIPS R4400SC	84	83	83	97,900	96,800
PowerPC-601	165	160	160	97,405	94,620
Alpha-21064	181	176	176	96,905	94,380

5) Synthesize Architecture: We make some assumptions as follows:

- 1 shared-bus costs $150 (capacity = 10);
- 8 × 8 MIN costs $240;
- 1 bank of 4 MB RAM costs $150;
- control circuitry and other costs = $700.

Currently there are six processor families in our object base and model base. Using the information given in Table II, it can be observed that five different processor models meet the SPEC benchmark requirements, the design-space is then explored at each level in a similar way as given in the previous CAP example. Table III shows the obtained feasible configurations.

By plotting the simulation results of the above five design alternatives in Fig. 22, we observe that design 5 is the best choice as far as throughput and utilization are concerned. The performance metric distance$_2$, mentioned in the previous example, was used and the plotting of this distance measure for the above five alternatives is shown in Fig. 23. Finally, it can be concluded that a multiprocessor system with 176 Alpha-21064 CPUs, a 128 MB global memory, and a multistage interconnection network is the best design choice for the sorting example.

C. Other Examples

Table IV shows four representative multiprocessor system designs created by PSM. Due to space consideration, we do not present the Functional Models and the detailed synthesis processes. However, from Table IV, it can be observed that a wide range of systems were synthesized using PSM, which include a massively parallel system (Design #1), a system

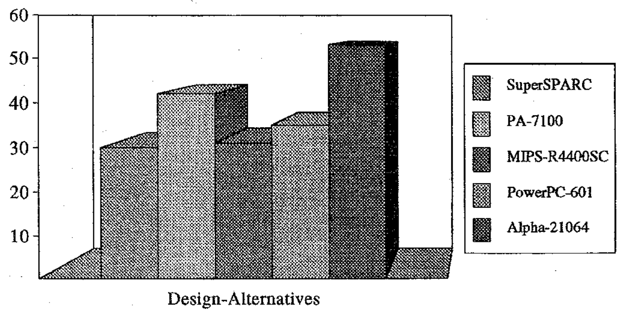


Fig. 22. Throughput/utilization of the design alternatives in Example 2.

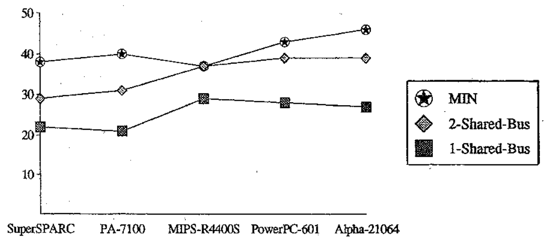


Fig. 23. Distance metric of each design alternative with different interconnection methods.

TABLE IV
REPRESENTATIVE DESIGNS CREATED BY PSM

	Partial Design Specification			Synthesis Results					
Design No.	Functional Model Summary	Cost Bound ($)	Power (=T/U) (sec^{-1})	Method of Control	No. of processors	Synthesis Levels	Physical Objects	Run-time (sec)	Estimated GFlops
1	A fine-grained, highly parallel job.	11,500,000	100	SIMD	10,240	5	32	605	10.5
2	Engineering computation requiring large memory space.	1,100,000	70	SIMD	1,024	4	26	519	5.4
3	Scientific computation requiring high processing speed	1,750,000	200	MIMD	1,024	4	29	580	128
4	A message-passing environment.	600,000	80	MIMD	512	3	20	472	2

requiring large memory space (Design #2), a system requiring high speed (Design #3), and a message-passing system (Design #4). The run-time given here includes only the DSE and simulation times.

Design #1 is comparable to the fine-grain massively parallel connection machine CM-2 as far as the type of parallel architecture (massively parallel), the method of control (SIMD), the number of processors, the interconnection method (hypercube interconnection), and the estimated peak performance (10 Gflops) are concerned. This also shows the feasibility of PSM as a synthesis methodology for synthesizing massively parallel systems.

VI. CONCLUSION

The following statements can be concluded about our performance synthesis methodology.

1) It provides a new synthesis methodology to automate the design of multiprocessor systems at the system level. Since PSM was designed specifically for the synthesis of MP systems, it can synthesize parallel computer systems more efficiently and produce better results than the currently available uniprocessor synthesis tools.
2) The incorporation of object-oriented techniques into hardware design makes the synthesis methodology more efficient and easier to maintain. Therefore, the system development time can be shortened and the design cost reduced. Since time and cost are two very important factors in the market, if PSM is successfully applied in the commercial field, it will certainly be helpful in assisting a company to win a bigger market share.
3) Each phase and subphase of PSM can be controlled by the designer. This feature makes the methodology flexible and easily adaptable to different design environments. It also provides designers with a chance to discover any architecture defect in the early phases of the design process.
4) PSM provides a flexible cost-to-performance tradeoff by generating several design alternatives. This allows a designer to choose the architecture that best satisfies the required constraints tradeoff. For example, a designer may pick out a design that does not have the best performance but has a very low price in order to save money, or alternatively may choose a costly design to meet the demands of high performance applications.
5) Changes in technology can be easily incorporated into PSM by merely modifying or adding technology-specific information into the Object Base and the Model Base of PSM. Therefore, existing designs can be resynthesized and new MP systems can be synthesized using the new technology.

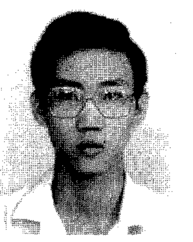
Some of our future research directions are:

1) *Virtual Machine or Simulator Generator:* Due to the complexity of today's parallel computer system, simulation before fabrication is indispensable in the design process. PSM can also be used to generate virtual machine descriptions which are executable simulators and to check whether the design satisfies user-given specifications at different design levels.
2) *Hardware–Software Codesign:* Though PSM is mainly a hardware synthesis methodology, yet the basic notion of object modeling using object-oriented technique can also be used to model software components. One of our future research directions is making PSM suitable for hardware-software cosynthesis of MP systems.

REFERENCES

[1] VLSI Design Staff, "Silicon compilers—Part 1: Drawing a blank," *VLSI Design*, vol. 5, no. 9, pp. 54–58, Sept. 1984.
[2] ——, "Silicon compilers—Part 2: Casting an image," *VLSI Design*, vol. 5, no. 10, pp. 65–68, Oct. 1984.
[3] R. Camposano, "From behavior to structure: High-level synthesis," *IEEE Design Test Comput.*, vol. 7, pp. 8–19, Oct. 1990.
[4] R. Dutta, J. Roy, and R. Vemuri, "Distributed design-space exploration for high-level synthesis systems," in *Proc. 29th ACM/IEEE Design Automation Conf.*, 1992, pp. 644–650.
[5] J. Roy, N. Kumar, R. Dutta, and R. Vemuri, "DSS: A distributed high-level synthesis system," *IEEE Design Test Comput.*, vol. 9, pp. 18–32, June 1992.

[6] M. C. McFarland, A. C. Parker, and R. Camposano, "The high-level synthesis of digital systems," *IEEE Proc.*, vol. 78, pp. 301–318, Feb. 1990.
[7] W. P. Birmingham, A. P. Gupta, and D. P. Siewiorek, "The MICON system for computer design," in *Proc. 26th ACM/IEEE Design Automation Conf.*, 1989, pp. 135–140.
[8] ——, *Automating the Design of Computer Systems: The MICON Project*. New York: Jones and Bartlett, 1992.
[9] A. P. Gupta, W. P. Birmingham, and D. P. Siewiorek, "Automating the Design of Computer Systems," *IEEE Trans. Computer-Aided Design*, vol. 12, pp. 473–487, Apr. 1993.
[10] A. J. Gadient and D. E. Thomas, "A dynamic approach to controlling high-level synthesis CAD tools," *IEEE Trans. VLSI Syst.*, vol. 1, pp. 328–341, Sept. 1993.
[11] D. E. Thomas, E. M. Dirkes, R. A. Walker, J. V. Rajan, J. A. Nestor, and R. L. Blackburn, "The system architect's workbench," in *Proc. 25th ACM/IEEE Design Automation Conf.*, June 1988, pp. 337–343.
[12] R. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang, "Mis: A multiple level logic optimization system," *IEEE Trans. Computer-Aided Design*, vol. 6, pp. 1062–1081, Nov. 1987.
[13] M. T. Trick and S. W. Director, "Lassie: Structure to layout for behavioral synthesis tools," in *Proc. 26th. ACM/IEEE Design Automation Conf.*, June 1989, pp. 104–109.
[14] S. A. Mabbs and K. E. Forward, "Performance analysis of MR-1, A clustered shared-memory multiprocessor," *J. Parallel Distrib. Comput.*, vol. 20, pp. 158–175, Feb. 1994.
[15] M. C. Chiang and G. S. Sohi, " Evaluating design choices for shared bus multiprocessors in a throughput-oriented environment," *IEEE Trans. Comput.*, vol. 41, pp. 297–317, Mar. 1992.
[16] D. S. Rao and F. J. Kurdahi, "Hierarchical design space exploration for a class of digital systems," *IEEE Trans. VLSI Syst.*, vol. 1 pp. 282–295, Sept. 1993.
[17] S. Kumar, J. H. Aylor, B. W. Johnson, and W. A. Wulf, "Object-oriented techniques in hardware design," *Computer*, vol. 27, no. 6, pp. 64–70, June 1994.
[18] Y. K. Lee and S. J. Park, "OPNETS: An object-oriented high-level Petri-net model for real-time system modeling," *J. Syst. Software*, vol. 20, no. 1, pp. 69–86, Jan. 1993.
[19] M. J. Chung and S. Kim, "An object-oriented VHDL design environment," in *Proc. 27th ACM/IEEE Design Automation Conf.*, 1990, pp. 431–436.
[20] G. Bruno and A. Balsamo, "Petri net-based object-oriented modeling of distributed systems," in *Proc. Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Sept. 1986, pp. 284–293.
[21] G. Kemeni and J. L. Snell, *Finite Markov Chains*. Princeton, NJ: Van Nostrand, 1960.
[22] M. A. Marsan, G. Balbo, and G. Conte, "A class of generalized stochastic Petri nets for the performance evaluation of MP systems," *ACM Trans. Comput. Syst.*, vol. 2, no. 2, pp. 93–122, May 1984.
[23] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*. Englewood Cliffs, NJ: Prentice Hall, 1991.
[24] K. Hwang, *Advanced Computer Architecture*. New York: McGraw-Hill, 1993.
[25] *SES/Workbench User's Manual Release 2.1*, Scientific and Engineering Software, Inc., Feb. 1992.
[26] *SES/Workbench Reference Manual Release 2.1*, Scientific and Engineering Software, Inc., Feb. 1992.
[27] J. L. Peterson, "Petri nets," *ACM Comput. Surv.*, vol. 9, no. 3, pp. 223–252, Sept. 1977.
[28] ——, *Petri Net Theory and the Modeling of Systems*. Englewood Cliffs, NJ: Prentice-Hall, 1981.
[29] K. Jensen and G. Rozenberg, Eds., *High-Level Petri Nets: Theory and Application*. New York: Springer-Verlag, 1991.
[30] M. Shaw, W. Wulf, and R. London, "Abstraction and verification in Alphard: Iteration and generators," in *Alphard: Form and Content*. New York: Springer-Verlag, 1981.
[31] J. M. Schoen Ed., *Performance and Fault Modeling with VHDL*. Englewood Cliffs, NJ: Prentice-Hall, 1992.



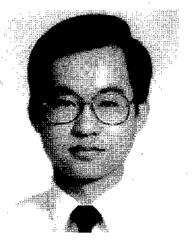
Pao-Ann Hsiung received the B.S. degree in mathematics from the National Taiwan University, Taipei, Taiwan, R.O.C., in 1991. He is currently working toward the Ph.D. degree in the Department of Electrical Engineering at the National Taiwan University.

He is a Teaching Assistant in the Department of Mathematics at the National Taiwan University. His research interests include system-level design methodologies for multiprocessor systems, hardware-software codesign, object-oriented synthesis, and multiprocessor performance modeling.

Sao-Jie Chen (S'85–M'88) received the B.S. and M.S. degrees in electrical engineering from the National Taiwan University, Taipei, Taiwan, R.O.C, in 1977 and 1982, respectively, and the Ph.D. degree in electrical engineering from the Southern Methodist University, Dallas, TX, in 1988.

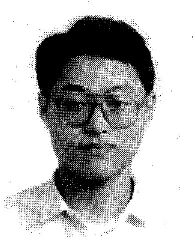
Since 1982, he has been a member of the faculty in the Department of Electrical Engineering, National Taiwan University, where he is currently an Associate Professor. From 1985 to 1988, he was on leave from the National Taiwan University and working toward the Ph.D. degree at the Southern Methodist University. During the fall of 1987, he held a visiting appointment at the Department of Electrical and Computer Engineering, University of Wisconsin, Madison. His current research interests include VLSI physical design automation, object-oriented software engineering, and supercomputer architecture design and simulation.

Dr. Chen is a member of the Chinese Institute of Engineers, the Association for Computing Machinery, and the IEEE Computer Society.



Tsung-Chien Hu received the B.S. degree in computer science and the M.S. degree in information engineering from the Tamkang University, Taipei, Taiwan, R.O.C. He is also a Ph.D. candidate at the National Taiwan University.

From 1986 to 1989, he was a Lecturer in the Department of Computer Science at the Tamkang University. Since 1989, he has been a Senior and Principal Member of Technical Staff with TeamWare Information Consulting Services Company. Currently, he is a chief leader of software products in the Vitek Communication Laboratories mainly developing object-oriented software engineering tools and visual communication technologies. His current research interests include: object-oriented software engineering, CASE technologies and concurrent object-oriented programming, methodology engineering, groupware, and hypermedia technologies.



Shih-Chiang Wang received the B.S. and M.S. degrees in electrical engineering from the National Taiwan University, Taipei, Taiwan, R.O.C., in 1991 and 1993, respectively. The title of his master thesis was "Petri Net-Based Performance Synthesis."

He is currently serving in Apple Computer Asia, Inc., Taiwan Branch, as a System Engineer. From 1991 to 1993 he participated in a project sponsored by the National Science Committee and researched performance synthesis methodology.