CMAPS: A Cosynthesis Methodology for Application-Oriented Parallel Systems

PAO-ANN HSIUNG
Academia Sinica

Currently, a lot of research is devoted to *system design*, and little work is done on *requirements analysis*. Besides going from specification to design, one of our main objectives is to show how an application problem can be transformed into specifications. Working from the hardware-software codesign perspective, a system is designed starting from an application problem itself, rather than the detailed behavioral specifications. Given an application problem specified as a directed acyclic graph of elementary problems, a hardware-software solution is derived such that the synthesized software, a parallel pseudoprogram, can be scheduled and executed on the synthesized hardware, a set of system-level parallel computer specifications, with heuristically optimal performance. This is known as system-level cosynthesis of application-oriented general-purpose parallel systems for which a novel methodology called *Cosynthesis Methodology for Application-Oriented Parallel Systems* (CMAPS), is presented. Since parallel programs and multiprocessor architectures are largely interdependent, CMAPS explores the relationship between hardware designs and software algorithms by interleaving the modeling phases and the synthesis phases of both hardware and software. High scalability in terms of problem complexity and easy upgradability to new technologies are achieved through modularization of the input problem specification, of the software algorithms, and of the hardware subsystem models. The work presented in this paper will be beneficial to designers of general-purpose parallel computer systems which must be oriented toward solving some user-specified problem such as the global controller of an industry automation process or a multiprocessor video server. Some application examples are given to illustrate various codesign phases of CMAPS and its feasibility.

Categories and Subject Descriptors: J.6 [**Computer Applications**]: Computer-Aided Engineering—*Computer-aided design* (CAD); C.0 [**Computer Systems Organization**]: General—*System architectures*; *Systems specification methodology*; C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors); C.5.0 [**Computer System Implementation**]: General

General Terms: Design

Additional Key Words and Phrases: Application-oriented general-purpose multiprocessors, hardware-software modeling and cosynthesis, requirements analysis

Author's address: Institute of Information Science, Academia Sinica, Sec. 2, No. 128, Academic Road, Nankang, Taipei, 115, Taiwan; email: eric@iis.sinica.edu.tw.

1. INTRODUCTION

A system is designed more often from a set of behavioral or architectural specifications than from the original requirements of a user. This is called *system design*. Before system design, a user's requirements must often be analyzed to derive system specifications. This is called *requirements analysis*. Much research has been done in developing methods, either technical or formal, to design a system from specifications. Comparatively speaking, requirements analysis has not been given equal attention. A user often has to specify in elaborate detail the behavior or architecture of the designed system. As far as design automation is concerned, it would certainly be desirable if a user's requirements could be directly input to a synthesis tool or methodology and a system designed from the requirements. This paper attempts to provide a solution within the hardware-software codesign perspective.

Synthesis is the process of automatically transforming a set of high-level system specifications to a lower-level design schematic that includes more architectural details required for the physical design of the system. Corresponding to the different levels of architecture details, synthesis can be performed at the system, the algorithm, the register-transfer, and the logic levels. Hardware synthesis has helped designers to reduce design time, effort, and cost. Several methodologies and tools have been proposed at each level of synthesis [Hsiung et al. 1996; Hsiung et al. 1998; Birmingham et al. 1993]. When software is simultaneously synthesized, it is called *cosynthesis* or *codesign* which requires system partitioning, hardware-software tradeoff refinements, and cosimulation. In particular, embedded digital systems and DSP applications are often targets of hardware-software cosynthesis [Wolf 1994; Kalavade and Lee 1993].

Parallel computer systems use more than one processor to provide supercomputing power, but at the same time designers of such systems are faced with the numerous design tradeoffs possible in such systems. Portability of software across parallel systems is also much more restrictive than across uniprocessor systems. This is due to the heavy dependence of parallel programs on the different architecture schemes in parallel systems. For instance, a parallel program with `send message` and `receive message` primitives is best executed on a message-passing parallel architecture, whereas a parallel program with shared variables should be run on a shared memory architecture for optimal performance. This interdependence between parallel software and parallel hardware motivates the codesign approach proposed in this paper.

Increasing diversity in user requirements for computer-based applications necessarily implies higher budget allocation for several different specialized systems. The overall cost expended by a user needing to run several applications can be lowered at the expense of a slight decrease in performance by using a general-purpose parallel computer system whose subsystems are appropriately configured for executing some given applications. Besides the traditional *application-specific parallel* (ASP) systems,

we also consider the codesign of *application-oriented general-purpose parallel* (AOGPP) systems, which are defined as general-purpose systems with their subsystems designed for the efficient execution of some software solution to a given problem. The reason for selecting such target systems is intuitive. On the one hand, a purely *general-purpose system* is a performance-balanced system which may not give the best performance in solving a specific problem, and on the other hand, an *application-specific system* often cannot be used to solve any problem besides the original application that it was designed for.

A typical example of AOGPP systems include the image processing workstations such as SGI's Origin2000 distributed servers coupled with cellular IRIX 6.4 OS, which optimize performance for desktop visual computing applications running on personal workstations and at the same time support normal end-user general-purpose standard software applications such as general computing, fileserver, and database applications and standard hardware such as PCI and XIO devices. A corresponding example of an ASP system would be medical imaging systems which do not support standard end-user applications such as web facilities, etc. Digital embedded systems are also ASP systems. Some other examples of AOGPP systems include Sun's Ultra-4000 Creator3D workstations optimized for demanding graphics applications such as engineering visualization and design, medical imaging and video animation; CRAY Research's CRAY Origin-2000 family including CRAY T3E, CRAY J90 and T90 are systems with very broad system scalability optimized for high-performance computing and graphics.

The following are some basic differences between the codesign of AOGPP systems and that of ASP systems:

—In AOGPP system cosynthesis, both hardware and software are needed to solve a given problem, whereas in ASP system cosynthesis a given problem can be solved technically using a complete hardware implementation or a complete software one.

—The synthesized hardware in AOGPP systems is only *optimized* for solving a given problem but, being general-purpose at the same time, it can solve any other problem, too; whereas the hardware in ASP systems generally cannot solve any other problem besides the one it was designed for.

—AOGPP system cosynthesis works at the system level, whereas most current ASP system cosyntheses work at a lower level of design.

—In ASP system cosynthesis, ASICs are synthesized along with corresponding software drivers and protocols, whereas AOGPP systems are mainly synthesized from existing off-the-shelf building blocks.

There are also many similarities between the cosyntheses of AOGPP systems and ASP systems.

—There is a heavy interdependence between hardware architectures and software algorithms in AOGPP systems, just like the hardware engines and software architectures in ASP systems.

—Similar to ASP systems, the cosynthesis of an AOGPP system must explore a vast design space consisting of both hardware and software modules.

—Hardware-software modeling phases are interleaved so as to arrive at one of the best combination of hardware and software design models.

—Cosimulation is needed to compare the various codesign alternatives.

This paper is organized as follows: Section 2 describes some previous and related work. Section 3 defines AOGPP system cosynthesis and the three repositories used in the design. Section 4 describes our cosynthesis methodology for AOGPP systems. Besides the small running example used for illustrating the design phases, two more realistic design examples are given in Section 5 to show the feasibility of the proposed methodology. A conclusion is drawn in Section 6.

2. PREVIOUS AND RELATED WORK

As far as hardware design is concerned, methodologies for the system-level synthesis of general-purpose multiprocessor systems have recently been proposed; for example, *Performance Synthesis Methodology* (PSM) [Hsiung et al. 1996] and *Intelligent Concurrent Object-Oriented Synthesis* (ICOS) methodology [Hsiung et al. 1998] are two of the most recently proposed methodologies. Some other successful methodologies for hardware design include the Micon System [Birmingham et al. 1989; Gupta et al. 1993], and the Megallan System [Gadient and Thomas 1993].

The current hardware-software codesign researches are all devoted to application-specific systems such as heterogeneous multiprocessor systems [Prakash and Parker 1992], DSP applications [Kalavade and Lee 1993], embedded digital systems [Gupta and De Micheli 1993], UNITY language programs [Barros, Rosentiel, and Xiong 1994], and distributed embedded computing systems [Wolf 1994; Yen and Wolf 1995b, a; Wolf 1996; 1997]. Application-specific systems typically require system partitioning into hardware and software parts. Therefore, current researches are typically devoted to hardware-software partitioning and tradeoffs exploration [Rozenblit and Buchenrieder 1995; Berge et al. 1997], which include strategies to move operations from software to hardware [Ernst et al. 1993] and from hardware to software [Gupta and De Micheli 1993], to allocate functions in an 1-CPU/ n-ASIC system [Vahid et al. 1994], to use multiple task graphs for heuristic cosynthesis [Yen and Wolf 1995b, 1995a], and to derive method data flow graphs from object-oriented specifications for the construction of distributed hardware-software topologies [Wolf 1996].

According to the authors' knowledge and survey, there is currently no literature on the hardware-software codesign of *general-purpose* parallel

systems. This may be due to a number of reasons including the large size of the design space that requires an exploration time exponential in the total number of hardware-software components, the lack of hardware architecture synthesis methodologies for general-purpose systems before PSM and ICOS were proposed, the intricate explicit dependence between the parallel hardware and parallel software architectures, and the lack of an objective since general-purpose systems are supposed to be able to solve any kind of application problems with a performance-balanced execution environment.

Nevertheless, application-specific hardware-software codesign has indeed been a valuable source of codesign experiences and techniques that may be applied to the codesign of general-purpose systems. For example, the codesign framework proposed by Kumar et al. [1993] presented an important concept of iterative system refinements using an integrated hardware-software model. The codesign methodology proposed by Thomas et al. [1993] used a mixed hardware-software system model that facilitated cosimulation and cosynthesis which included performing system partitioning at the task level. Compared to Kumar's framework, Thomas' is not an iterative approach, it is a constructive one. Kalavade and Lee [1993] also propose a codesign methodology for DSP applications using the Ptolemy framework for simulation, prototyping, and software synthesis. This methodology, besides cosynthesizing hardware and software, also synthesizes the interface between them in a constructive way. Gupta and De Micheli [1993] proposed the cosynthesis of digital systems which used timing constraints to delegate tasks between hardware and software. Three different kinds of system implementation: design-oriented, synthesis-oriented, and codesign approaches were discussed. Graph models were proposed for multirate systems where the general-purpose processor and application-specific hardware may run on different clocks and speeds. Program threads were also represented as directed graphs. Yen and Wolf [Wolf 1994; Yen and Wolf 1996] considered the codesign of embedded computing systems. Their target design consisted of a hardware engine made up of several processing elements (PE) which could be either CPUs or ASICs and an application software architecture with allocation and scheduling of processes and communication [Yen and Wolf 1995a]. Several new techniques such as fixed-point iterations, phase adjustments, and separation analysis were proposed for efficient delay estimation. A hardware-software cosynthesis algorithm with techniques such as sensitivity analysis, priority prediction, and idle-PE elimination was developed [Yen and Wolf 1995b]. The advantages of an object-oriented (OO) specification were explored by Wolf [1996], including the two levels of partition granularity inherent in OO specifications, the encapsulation of system objects, and the natural cut points provided by method decomposition.

As defined and described in Section 3, the cosynthesis problem considered in this paper is a new and important one. Our target architectures are parallel systems which include general-purpose multiprocessor systems, distributed multicomputer systems, and application-specific parallel systems. A hardware-software codesign methodology is presented to solve the

cosynthesis problems of all such parallel systems. Currently, there is no hardware-software codesign methodology that can be used to solve the cosynthesis problems of both general-purpose as well as application-specific parallel systems, hence the presented methodology is a novel and pioneer effort at deriving such a solution. Experiments on the system-level codesign of several designs using the proposed methodology show its feasibility, scalability, and easy technology upgradability. From the above literatures, we have adopted and adapted a few techniques in our methodology such as the iterative refinement of an integrated system, the mixed hardware-software model, and the graph-based software models. The methodology will be presented in Section 4.

3. COSYNTHESIS PROBLEM FORMULATION

System-level cosynthesis of application-oriented general-purpose parallel systems is defined as follows:

Definition 1. AOGPP system cosynthesis: Given an application problem composed of several elementary problems, a parallel system including a set of hardware system architecture specifications and a software algorithmic pseudoprogram solution, is to be synthesized such that the given problem can be optimally solved by executing the pseudoprogram software on the hardware architecture.

Optimal execution of software tasks on a parallel system requires *multiprocessor task scheduling* [Lin and Chen 1996] which is a known NP-complete problem [Ullman 1975], hence it is concluded that AOGPP system cosynthesis is at least NP-hard.

Since we work at the system-level of design, scalability in terms of the complexity of the application problem and the upgradability to new technologies are two major issues of any proposed codesign methodology. Scalability is increased in our methodology through the use of modularized problem models.

We define an *elementary problem* to be a very simple *known* problem which has algorithmic solutions. An *elementary algorithm* is an alogorithmic solution to an elementary problem. A target application may be a complex real-world problem which is composed of several subproblems, each of which is further composed of one or more elementary problems. For example, a multimedia application may consist of a graphical tool, a WYSIWYG editor, a music synthesizer, and a simple animator, each of which is a subproblem and is composed of several elementary problems such as computing discrete Fourier transforms, sorting a sequence of pixel positions, solving sets of linear or nonlinear equations, generating permutations and combinations, spline calculations, etc.

A user can specify a complex application problem by referring to *elementary problems* in a Problem Database and describing how the selected elementary problems compose into the desired application problem. Upgradability is made easy through the use of *elementary algorithms* which

act as off-the-shelf building blocks for software and the use of subsystem architecture models for hardware. Three repositories are used in our methodology, namely *Problem Database* (PD), *Algorithm Database* (AD), and *Model Database* (MD), which represent the modularizations of specification input, of software synthesis, and of hardware synthesis, respectively.

PD is used to store elementary problems and related information such as the unique problem name and pointers to the corresponding elementary algorithms that can be used to solve the specific problem. For example, sorting a sequence, solving a set of linear equations, generating permutations and combinations, and computing the discrete Fourier transform are all elementary problems. A list of elementary problems is shown in Table V. Formally, PD is defined as follows:

Definition 2. Problem Database (PD): PD is defined as a tuple (P, A, μ), where

—P is a set of elementary problems,

—A is a set of elementary algorithms,

—μ is a function mapping each problem in P to a subset of elementary algorithms from A, which can solve the specific problem, i.e., $\mu : P \rightarrow 2^A$.

AD is a collection of elementary parallel algorithms that can be used to solve the problems in PD. Related information such as time and space complexities, and the requirement restrictions on hardware architecture are all stored along with each algorithm. A partial Algorithm Database is shown in Table V. The illustrated algorithms are from Akl's book on parallel algorithms [1989]. Formally, AD can be defined as follows:

Definition 3. Algorithm Database (AD): AD is defined as a tuple (A, P, ν, ψ, ϕ), where

—A is a set of elementary algorithms,

—P is a set of elementary problems,

—ν is a function mapping each algorithm in A to a subset of elementary problems from P, which can be solved by the algorithm, i.e., $\nu : A \rightarrow 2^P$,

—ψ is a *software characteristics function* which maps an algorithm to its software requirements. $\psi(a) = (t(n), u(n))$, where $a \in A$ and $t(n)$ and $u(n)$ are bounds on the execution time and number of processors, respectively.

—ϕ is a *hardware characteristics function* which maps an algorithm to its hardware requirements. $\phi(a) = (cm, ml, ma, co)$, where $cm \in CM$, $ml \in ML$, $ma \in MA$, and $co \in CO$, and CM, ML, MA, and CO are the sets of communication models, memory latency models or system inter-

connection models, memory access models, and control models of a general-purpose parallel computer system as discussed in Definition 4.

Often a parallel algorithm is designed specifically for a particular system interconnection model; in such cases, ML represents the system interconnection model, otherwise ML is as given in Definition 4.

MD is a repository of models for hardware subsystems, such as *Communication* models (CM), *Memory Latency* models (ML), *Memory Access* models (MA), and *Control* models (CO). Formally, MD can be defined as follows:

Definition 4. Model Database (MD): MD is defined as a cartesian product of the coordinate sets of subsystem models, that is, $M = CM \times ML \times MA \times CO$, where

—CM is a set of *Communication* models, CM = {SM, MP} where SM is shared-memory and MP is message-passing,

—ML is a set of *Memory Latency* models, ML = {NUMA, COMA, UMA, NORMA},

—MA is a set of *Memory Access* models, MA = {CRCW, CREW, EREW}, and

—CO is a set of *Control* models, CO = {MIMD, SIMD}.

The mnemonics used in the above definition are as follows. We mainly consider two communication models: *Shared-Memory* (SM) and *Message-Passing* (MP). The memory latency models include the *Non-Uniform Memory Access* (NUMA), the *Cache-Only Memory Access* (COMA), the *Uniform Memory Access* (UMA), and the *NO-Remote Memory Access* (NORMA) models [Hwang 1993]. The memory access models include the *Concurrent Read Concurrent Write* (CRCW), the *Concurrent Read Exclusive Write* (CREW), and the *Exclusive Read Exclusive Write* (EREW) models [Fortune and Wyllie 1978]. The control models include the *Multiple Instruction Multiple Data-stream* (MIMD) and the *Single Instruction Multiple Data-stream* (SIMD) models [Flynn 1972].

The above discussion on the modularization of the problem, of the software, and of the hardware models further allow us to analyze the design space size for system-level cosynthesis. Assume that an application problem as specified by a system designer is composed of n elementary problems, $\{p_1, p_2, \ldots, p_n\}$, where $n \leq |P|$ and $p_i \in P$, $1 \leq i \leq n$, and for each i, there are x_i algorithms from A that can solve subproblem p_i, where $x_i \leq |A|$. Further, assume there are y_{ij} feasible hardware model configurations for the jth algorithm of p_i, where $y_{ij} \leq |M|$, $1 \leq i \leq n$ and $1 \leq j \leq x_i$. Thus, the total design space size is

$$\mathcal{S}_{codesign} = \prod_{i=1}^{n} \sum_{j=1}^{x_i} y_{ij} \tag{1}$$

For ease of observation, assume that $x_i = x$ for all i and $y_{ij} = y$ for all i and j, then the design space size becomes

$$\mathcal{S}_{codesign} = (xy)^n = x^n y^n \tag{2}$$

From above, we conclude that the codesign space size can be expressed in terms of the software design space size ($\mathcal{S}_{software} = x^n$) and the hardware design space size ($\mathcal{S}_{hardware} = y^n$).

$$\mathcal{S}_{codesign} = \mathcal{S}_{software} \times \mathcal{S}_{hardware} \tag{3}$$

4. COSYNTHESIS METHODOLOGY FOR APPLICATION-ORIENTED PARALLEL SYSTEMS

Having gone through the basic concepts, we explain our methodology called *Cosynthesis Methodology for Application-Oriented Parallel Systems* (CMAPS) in detail in this section. As shown in Figure 1, the design flow is divided into three main phases: (1) *Initialization*, (2) *Modeling and Evaluation*, and (3) *Synthesis and Simulation*.

In brief, designers can input their specifications by constructing a Problem Graph using elementary problems from a Problem Database, along with subproblem sizes and other related constraints. First, CMAPS maps this graph into an initial solution. Then, CMAPS transforms the initial solution into hardware models and software models, and coevaluates them while checking which models can be eliminated to decrease the complexity of synthesis. Finally, the hardware and software models are synthesized into hardware system-level specifications and software pseudoprograms, respectively, and a cosimulation of hardware and software is performed after having chosen an appropriate scheduling algorithm.

4.1 Initialization Phase

The designer specifies his or her problem using a *Problem Graph* (PG) which is a directed acyclic graph $G_P(V_P, E_P)$, such that $V_P = \{v_i \mid v_i$ represents a problem $p_i \in P\}$ and $E_P = \{(v_r, v_s) \mid v_r$ must be solved before v_s and $v_r, v_s \in V_P\}$. This graph is similar to the traditional task graph specification used in distributed system synthesis [Chu and Tan 1987] and cosynthesis algorithms [Prakash and Parker 1992; Yen and Wolf 1995; Wolf 1996; 1997].

The result of this phase is a *Solution Graph* (SG), which is defined to be a directed acyclic graph $G_S(V_S, E_S)$, where each vertex in V_S represents an elementary algorithm from AD and each edge in E_S represents the order of precedence between two algorithms.

A PG input ($G_P(V_P, E_P)$) is transformed into an SG result ($G_S(V_S, E_S)$) through the following solution modeling process:

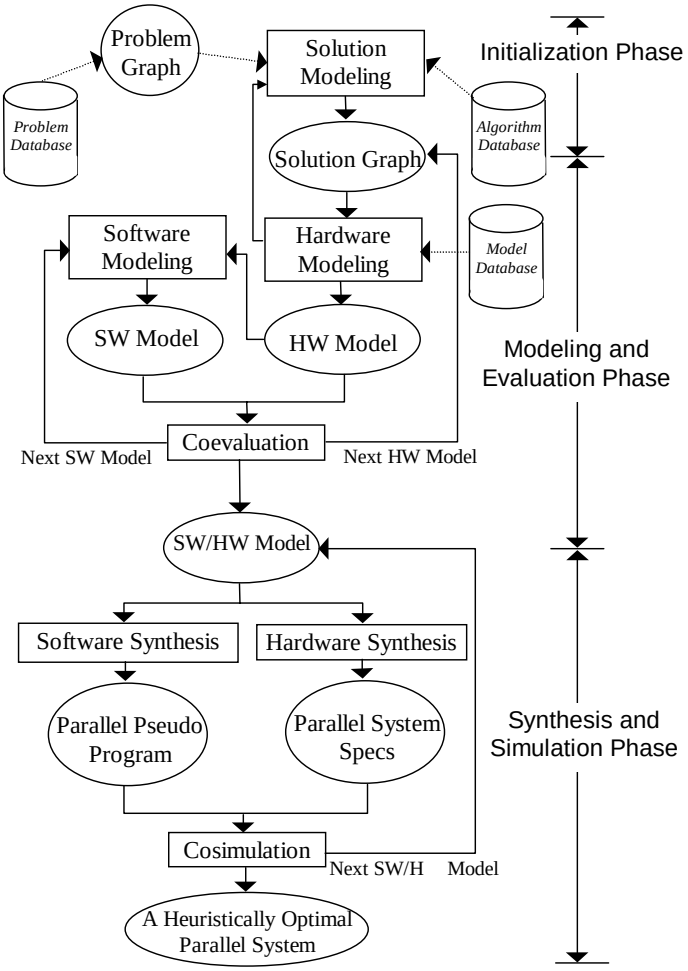

Fig. 1. CMAPS design flow.

```
model_solution(G_P, G_S, A)
begin
  for each v_i in V_P do
    select a_i from A such that
      (1) a_i solves p_i (represented by v_i) and
      (2) cost(a_i) = MIN{cost(a_k) | a_k solves p_i }
        where "cost(a_i) = exec_time(a_i) * num_processor-
        s(a_i)"
    V_S = set_union(V_S, {a_i})
  endfor
  for each (v_r, v_s) in E_P do
    E_S = set_union(E_S, {(a_r, a_s)})
  endfor
end.
```

The resulting SG is most probably not a feasible solution, but it serves as a useful initial solution for the next *Modeling and Evaluation* phase. The

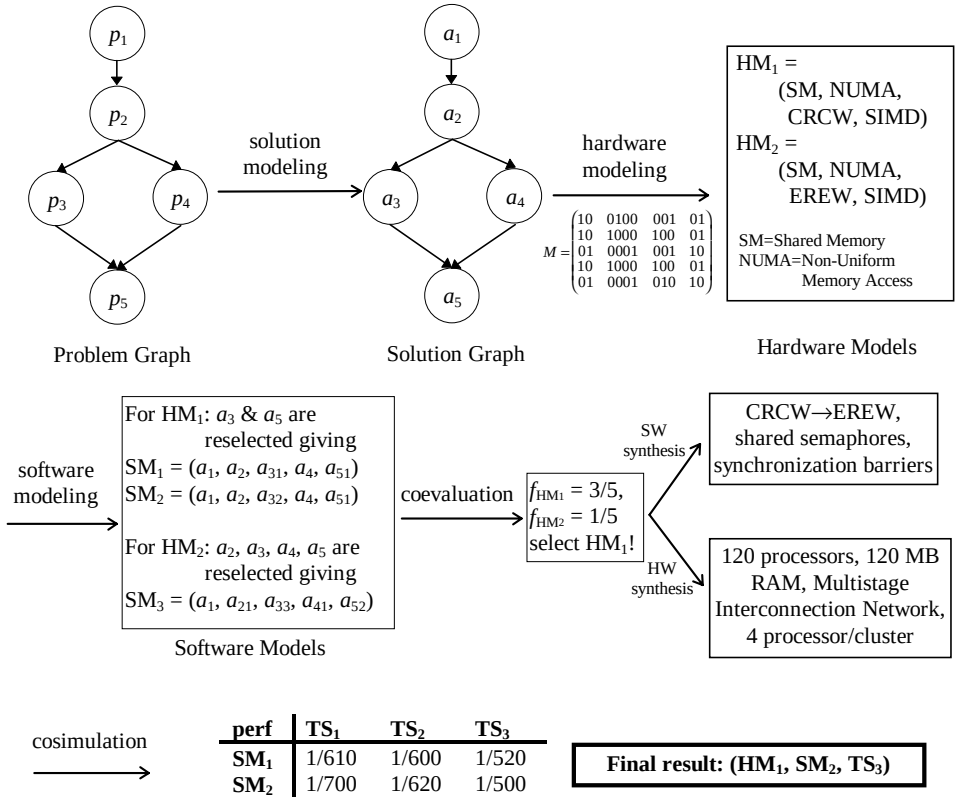

Fig. 2. CMAPS running example.

graph SG is not unique when the selected algorithm a_i for problem p_i is not unique. For ease of illustration, the above pseudocode generates one SG; it can be easily extended to generate all possible SGs.

The various phases in this section are illustrated using a small running example given in Figure 2. The Problem Graph, as input by the designer consists of five elementary problems, p_1, p_2, . . . , p_5, each being an elementary problem from the Problem Database. Five subalgorithms, a_1, a_2, . . . , a_5, are selected from the Algorithm Database, each being the algorithm that best solves the corresponding problem. These five subalgorithms are composed into an initial solution, the Solution Graph, as drawn in Figure 2.

4.2 Modeling and Evaluation Phase

The Solution Graph (SG) obtained in the *Initialization* phase is made feasible iteratively through an interleaving of hardware and software modeling processes. This phase consists of three subphases: *Hardware Modeling*, *Software Modeling*, and *Coevaluation*. Using SG, a *Hardware Model* (HM) is generated in the *Hardware Modeling* subphase by going

through the following steps: *Initialization*, *Model-Space Exploration*, and *Model Configuration* steps. The *Software Modeling* subphase mainly constitutes the transformation of a Solution Graph (SG) into a *Software Model* (SM), the difference being that SG may be nonfeasible, but SM has to be feasible, that is, its requirements matching those provided by the corresponding HM. The final *Coevaluation* subphase reduces the number of hardware and software models to be considered for synthesis, thus significantly decreasing the complexity of cosynthesis.

4.2.1 *Hardware Modeling Subphase*. In the following, we assume that a given problem has n subproblems, that is, $|V_S| = n$, where SG = (V_S, E_S) is the Solution Graph of the given problem and each subproblem is an elementary problem from PD. Further, we assume that a *Hardware Model* (HM) has m *features*, where a *feature* is a hardware design characteristic; for example, some features can be the memory organization, the system interconnection network, etc. Further, each hardware feature may have different values assigned to it; we call them *feature options*; for example, `Shared Bus`, `Mesh`, and `Hypercube` are feature options for the system interconnection network feature. As shown in Figure 3 and described below, this subphase consists of three steps: *Initialization*, *Model-Space Exploration*, and *Model Configuration*.

Step a. *Initialization*: An $n \times m$ *hardware requirement matrix*, $M(m_{ij})$, is constructed as follows such that m_{ij} represents the jth hardware model feature (f_j) of the ith subalgorithm (a_i), $i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, m$.

(1) Sort the hardware model features in a descending order of the overall degree of effect that a feature has on the system or in a descending order of the degree of importance as stipulated by a system designer. For example, a typical order may be *CM* (communication model), *ML* (memory-latency model), *MA* (memory-access model), and *CO* (control model).

(2) Denote feature options using binary values from the set {1,10,100, . . . } such that a larger value indicates a functionally stronger option, e.g., CRCW = 100, CREW = 010, and EREW = 001 in the case of memory access models.

(3) Let bit(m_{ij}, k) return the kth least significant bit of m_{ij} and let b_j be the number of significant bits in the binary representation of the jth feature f_j,

$$\text{bit}(m_{ij}, k) = 1 \text{ if } a_i \text{ requires the } k\text{th option of } f_j, k = 1,2, \ldots, b_j \quad (4)$$

For instance, the matrix M for the small running example given in Figure 2 is

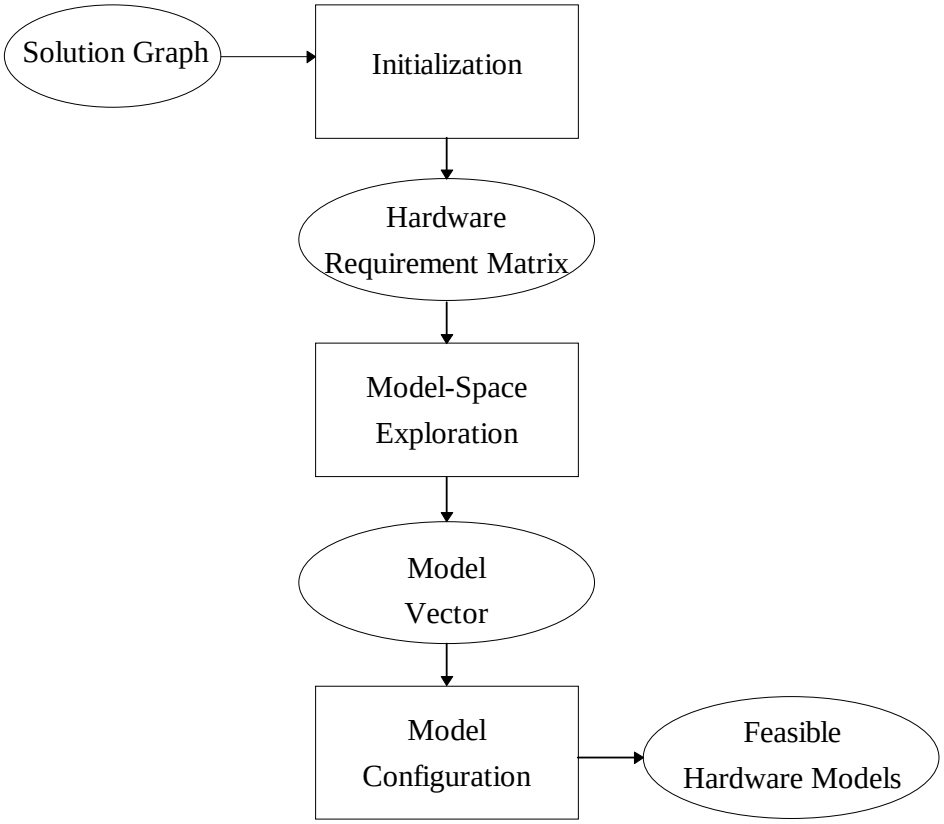

Fig. 3. Hardware modeling.

$$M = \begin{pmatrix} 10 & 0100 & 001 & 01 \\ 10 & 1000 & 100 & 01 \\ 01 & 0001 & 001 & 10 \\ 10 & 1000 & 100 & 01 \\ 01 & 0001 & 010 & 10 \end{pmatrix} \tag{5}$$

where

—The first column represents the Communication (CM) model with 10 = Shared-Memory (SM) and 01 = Message Passing (MP),

—The second column represents the Memory Latency (ML) model with 1000 = NUMA, 0100 = COMA, 0010 = UMA, and 0001 = NORMA,

—The third column represents the Memory Access (MA) model with 100 = CRCW, 010 = CREW, and 001 = EREW, and

—The fourth column represents the Control (CO) model with 10 = MIMD and 01 = SIMD.

Step b. *Model-Space Exploration*: In this step, as given in Equation (6) the kth option of the jth feature is considered for further software modeling (denoted by $t_{jk} = 1$) if the *option demand* (s_{jk}) is at least the *mean demand* (n/b_j), $k = 1, 2, \ldots, b_j$ and n is the problem size given in terms of the number of elementary problems in PG. Here, the *option demand* is defined as the number of subalgorithms which demand the kth option of the jth feature and the *mean demand* is the average weight assigned to each feature, that is, the mean demand for the jth feature is n/b_j.

$$t_{jk} = \begin{cases} 1 & \text{if } s_{jk} \geq \dfrac{n}{b_j}, \text{ where } s_{jk} = \sum_{i=1}^{n} \text{bit}(m_{ij}, k) \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

For the running example in Figure 2, $n = 5$, $m = 4$, and using Equation (5) and Equation (6) t_{jk} are computed as follows:

	CM	ML	MA	CO
a_1	10	0100	001	01
a_2	10	1000	100	01
a_3	01	0001	001	10
a_4	10	1000	100	01
a_5	01	0001	010	10
s_{jk}	32	2102	212	23
n/b_j	2.5	1.25	1.66	2.5
t_{jk}	10	1001	101	01

Step c. *Model Configuration*: In this step, the hardware model configurations are generated from the *hardware model vector*, $\vec{v}$, which is defined from t_{jk} as follows:

$$\vec{v} = (t_{11} \ldots t_{1b_1}, t_{21} \ldots t_{2b_2}, \ldots, t_{m1} \ldots t_{mb_m}) \tag{7}$$

For the running example, $\vec{v} = (10,1001,101,01)$. Further, a designer may specify some hardware requirements which will be represented by a *hardware specification vector* $\vec{u}$. The hardware model configurations are generated as follows.

The configurations are generated starting from the *functionally strongest* one. In other words, for each feature in $\vec{v}$ generate configurations starting from the leftmost bit which has a value of '1'. Due to the order in which the configurations are generated, they are already sorted in a descending order of their binary values and there are totally $\Pi_{j=1}^{m} \Sigma_{k=1}^{b_j} t_{jk}$ configurations. Consider one configuration at a time. If the configuration selected is not feasible (as defined later), it is discarded and the next one is considered. Let $\vec{v}(v_j)$ be a hardware model vector generated in **Step b** and $\vec{u}(u_j)$ be a user-given *hardware specification vector*, $1 \leq j \leq m$. Nonfeasible hardware models are eliminated as follows:

(1) *Eliminate Models with Contradictions*: CMAPS uses basic hardware system assumptions such as a *Shared Memory* (SM) architecture ($v_1 = 10$) should not have a *NO Remote Memory Access* (NORMA) model (bit(v_2, 1) = 1), hence we eliminate such SM/NORMA combinations by performing a Boolean conjunction of v_2 with 1110 (& is the bit-wise AND operator in the following equations).

$$\textbf{if } (v_1 = 10) \wedge (\text{bit}(v_2, 1) = 1) \textbf{ then } v_2 = (v_2 \& 1110) \textbf{ endif} \tag{8}$$

Further, a *Message-Passing* (MP) architecture should not share any global memory, such as in the NUMA, COMA, and UMA models (i.e., $\Sigma_{k=2}^{4}\text{bit}(v_2, k) \geq 1$ if any memory is shared). Hence, we eliminate all such combinations by a Boolean conjunction of v_2 with 0001.

$$\textbf{if } (v_1 = 01) \wedge (\sum_{k=2}^{4}\text{bit}(v_2, k) \geq 1) \textbf{ then } v_2 = (v_2 \& 0001) \textbf{ endif} \tag{9}$$

(2) *Eliminate Models with Conflicts*: A Boolean conjunction of a generated hardware model vector ($\vec{v}$) and a user-given hardware specification vector ($\vec{u}$) will eliminate all the hardware models that are in conflict with the user given specification vectors.

$$v_j = (v_j \,\&\, u_j) \text{ for all } j,\ 1 \leq j \leq m \tag{10}$$

For example, suppose a user specifies two desired configurations ($\vec{u}$ and $\vec{u'}$), $\vec{u}$ is an MIMD machine with a NUMA memory latency (i.e., u_2 = NUMA and u_4 = MIMD) and $\vec{u'}$ an SIMD machine with a UMA memory latency (i.e., u'_2 = UMA and u'_4 = SIMD). Further, if a generated hardware model vector is an SIMD machine with NUMA memory latency (i.e., v_2 = NUMA and v_4 = SIMD), then there will be a conflict with the user given specification vectors, thus this hardware model vector $\vec{v}$ is eliminated.

(3) *Generate Feasible Configurations*: Finally, if no component of the hardware model vector is completely eliminated through the above steps (i.e., $\nexists j$, such that $v_j = 0$), then at least one feasible hardware model can be generated.

$$\textbf{if } \exists j,\ v_j = 0, \textbf{ then } \text{report\_error}(); \textbf{ else } \text{generate\_configs}(v) \textbf{ endif} \tag{11}$$

For our running example in Figure 2, after eliminating nonfeasible hardware models, the final feasible configurations generated are (10,1000,100,01) and (10,1000,001,01) corresponding to HM_1 = (SM, NUMA, CRCW, SIMD) and HM_2 = (SM, NUMA, EREW, SIMD).

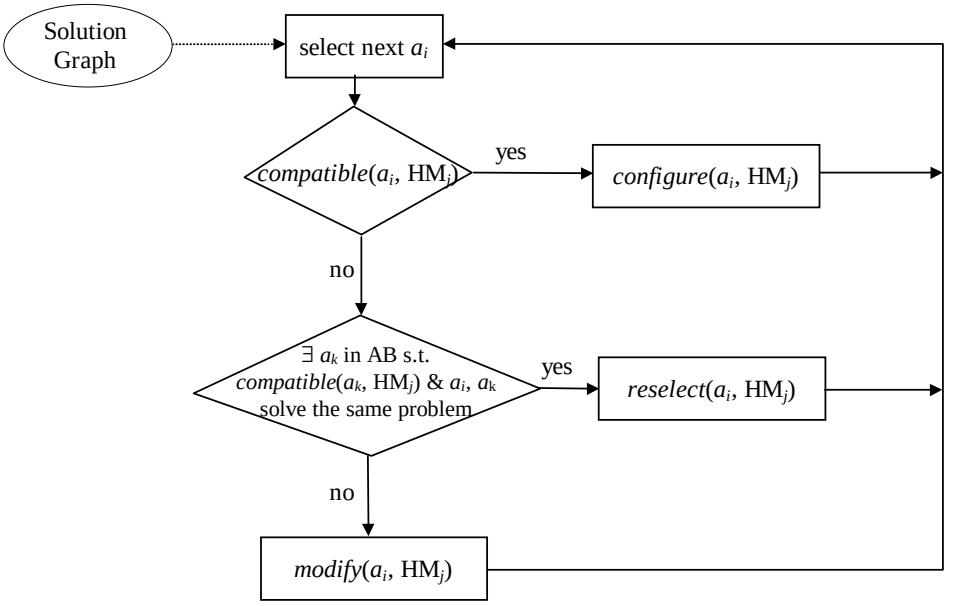

Fig. 4. Software modeling.

4.2.2 *Software Modeling Subphase*. Considering the feasible hardware models generated in the *Modeling and Evaluation Subphase* one at a time, software models are generated by transforming the Solution Graph (SG) into a feasible software solution. This transformation process, as shown in Figure 4, checks the compatibility of each subalgorithm in SG with the hardware model under consideration. *Compatibility* is defined in terms of the hardware model features, {*CM*, *ML*, *MA*, *CO*}, of which *Communication* (*CM*) model and *Control* (*CO*) model require an exact match, whereas *Memory Latency* (*ML*) model and *Memory Access* (*MA*) model are defined *compatible* when the hardware feature is functionally stronger than the software requirement; for example, CRCW, PRAM, and CREW algorithms are assumed compatible.

Considering a subalgorithm `a_i` with hardware requirements `(cm_i, ml_i, ma_i, co_i)` and a hardware model `HM_j = (CM_j, ML_j, MA_j, CO_j)`, their compatibility can be checked as follows.

```
compatible(a_i, HM_j)
begin
  if ( (cm_i == CM_j) && (ml_i <= ML_j)
       && (ma_i <= MA_j) && (co_i == CO_j)) then
    return TRUE
  else
    return FALSE
  endif
end.
```

If a subalgorithm is compatible with the hardware model under consideration then it is configured as follows:

```
configure(a_i, HM_j)
begin
  if (ml_i < ML_j) then
    ml_i = ML_j
  endif
  if (ma_i < MA_j) then
    ma_i = MA_j
  endif
end.
```

Noncompatibility of a subalgorithm calls for a *reselection* of the subalgorithm from the set of algorithms A. Reselection is possible only when there exists a compatible algorithm in AD such that it solves the same problem as the incompatible algorithm under consideration. If there is more than one such algorithm in A, then the one with the minimum cost will be selected for further synthesis.

```
reselect(a_i, HM_j)
begin
  Let R = {a_r | a_r is in A,
                   a_r solves the same problem as a_i, and
                   compatible(a_r, HM_j) }
  if (R is not empty) then
    select a_k such that
      cost(a_k) = MIN {cost(a_r) | a_r belongs to R }
  endif
end.
```

Finally, if there is no compatible algorithm for use in A, then the original one is modified into a compatible one. The modification of an algorithm for execution on a hardware system requires emulation techniques and is considered only when the software algorithm requires a stronger function feature than that existing on a hardware system, for example, a CRCW algorithm needs to be modified for possible execution on an EREW parallel system. As far as memory access is concerned, this modification is done by serializing reads and writes in PRAM models using PRAM simulation techniques [Harris 1994].

```
modify(a_i, HM_j)
begin
  serialize_mem_access(a_i);
end.
```

For the running example, assume that subalgorithms a_3 and a_5 are reselected from A for compatibility with HM_1, whereas a_2, a_3, a_4, and a_5 have to be reselected from A for compatibility with HM_2. As shown in Figure 2, for HM_1 two choices of a_3 from A generate two software models, SM_1 and SM_2, and for HM_2 one software model, SM_3, is generated.

4.2.3 *Coevaluation*. The final subphase is to coevaluate the software models (SM_i) with their corresponding hardware models (HM_j) in order to reduce the total number of models that have to be synthesized in the next phase. Corresponding to each combination of (SM_i, HM_j), compute the

degree of feasibility, $f = e/n$, where n is the number of elementary problems in PG and e is the ease of software modeling for SM_i with respect to HM_j as defined below,

$$f_{HM_j} = \frac{e(HM_j)}{n} = \frac{1}{n}\sum_{k=1}^{n} e_k, \text{ where } e_k = \begin{cases} 1 & \text{if compatible } (a_k, HM_j) \\ 0.5 & \text{if } a_k \text{ is reselected from } A \\ 0 & \text{if } a_k \text{ is modified} \end{cases} \tag{12}$$

Hardware models are selected based on the following criteria:

$$\text{Select } HM_r \text{ such that } f_{HM_r} \geq \frac{1}{h}\sum_{j=1}^{h}(f_{HM_j}) \tag{13}$$

where h is the total number of HM generated in the *Hardware Modeling* step.

For the running example as shown in Figure 2, the degrees of feasibility for HM_1 and HM_2 computed using Equation (12) are 3/5 and 1/5, respectively. Using the criteria given in Equation (13), only HM_1 is considered for further synthesis.

4.3 Synthesis and Simulation Phase

The previous *Modeling and Evaluation* Phase corresponds to requirements analysis, and the phase called *Synthesis and Simulation* described in this section corresponds to system design, as defined in Section 1. In this phase, the hardware and software models are now individually synthesized into parallel system specifications and parallel pseudoprograms, respectively, and then cosimulated by scheduling the parallel program on the parallel architecture defined by the parallel system specifications. The *Hardware Synthesis* subphase consists of four steps: *System Configuration*, *Processor Clustering*, *System Interconnection Selection*, and *Cluster Design*. The *Software Synthesis* subphase interconnects the final choice of algorithms by *Algorithm Interface Construction*, *Serialization of Memory Accesses*, and *Addition of Communication Constructs*. In the *Cosimulation* subphase, the hardware and software solutions generated in the previous two subphases are now interrelated by scheduling the software on the hardware.

4.3.1 *Hardware Synthesis Subphase*. After the three subphases of hardware modeling, software modeling, and coevaluation, pairs of feasible hardware-software models are generated. At this stage corresponding to a specific model pair, we have a set of hardware system-level specifications such as the communication model, the memory latency model, the memory access model, and the control model. For the running example in Figure 2, we have HM_1 = {SM, NUMA, CRCW, SIMD}, which says that the user-given problem graph can be solved by a parallel computer system which uses shared-memory (SM) under the uniform memory latency (NUMA),

along with concurrent read concurrent write (CRCW) memory access, and single instruction multiple data-stream (SIMD) control. At the same time, we also know from the requirements of the corresponding generated software model: the kind of system interconnection network required for each of the elementary algorithms, the total number of processors required, and the communication locality (i.e., the processor clustering) required. Given all these specifications, this subphase generates an actual feasible overall architecture that meets the demands of a hardware-software models pair generated in the previous subphase. This subphase consists of four steps as described in the rest of this subsection.

Step a. *System Configuration*: The type of processors, the total number of system processors (N_{SP}), and the total amount of memory are determined for each system combination (SM_i, HM_j) by using the *Performance Synthesis Methodology* (PSM) [Hsiung et al. 1996], a system-level object-oriented hardware synthesis methodology for multiprocessor systems. For better efficiency, *Intelligent Concurrent Object-Oriented Synthesis* (ICOS) methodology [Hsiung et al. 1998] may also be used.

Step b. *Processor Clustering*: The system processors are then grouped into clusters of N_{CP} processors each, where N_{CP} is calculated by taking the least common multiple (LCM) of the individual number of *processors per cluster* requirement (ppc_i) of each subalgorithm (a_i). If this LCM is larger than the total number of system processors (N_{SP}), then no clustering is done.

Step c. *System Interconnection Selection*: Given the set of interconnections required by the subalgorithms of a software model, it is reduced to a singleton set by using the following techniques:

(1) *redundancy elimination*: if two interconnections are functionally equivalent, one of them is removed from the set,

(2) *set reduction*: embeddable interconnections are removed from the set, and

(3) *set integration*: a functionally more powerful interconnection which can embed two or more interconnections from the set, is included into the set and the embeddable ones removed.

The final single element in the set is the system interconnection choice. The above process is guaranteed to terminate resulting in a singleton set because in the worst case the `Crossbar Interconnection` can embed all interconnections.

Step d. *Cluster Design*: The target parallel system is assumed to have symmetrical clusters. A cluster is designed using the same principles as for the global design of the whole parallel system. It includes the cluster memory subsystem configuration and the cluster interconnection network selection.

4.3.2 *Software Synthesis Subphase*. Software synthesis subphase consists of the following steps: *Algorithm interface construction*, *Serialization*

of memory accesses, and *Addition of communication constructs* for each of the selected or modified subalgorithm of a software model.

(1) *Algorithm interface construction* is the transformation of an algorithm into a pseudo-program by declaring data types, implementing functions and procedures, using structured programming constructs, and realizing the passing of computation results from a parent subalgorithm to a child subalgorithm in the software model. User specified program interfaces have also to be implemented.

(2) *Serialization of memory accesses* is mainly the serializing of read and write accesses in CRCW and CREW models into the EREW model [Harris 1994]. Since all parallel programs eventually have to be executed on real machines which do not allow concurrent memory accesses (i.e., EREW machines), the programs must have all memory accesses serialized.

(3) *Addition of communication constructs* such as shared semaphores, send-receive primitives, and synchronization barriers into corresponding shared-memory and message-passing subalgorithm in the software model is carried out.

The result of software synthesis is a parallel pseudoprogram which is implementation independent.

4.3.3 *Cosimulation Subphase*. The final part of synthesis is cosimulation of the above synthesized hardware system specifications and software pseudoprograms by scheduling the software solution on the hardware architecture using three different parallel tasks scheduling algorithms, namely, *List Scheduling* (LS) [Graham 1969], *Largest Scheduled Parallelism First* (LSPF) [Lin et al. 1995], and *Largest Width with Largest Processing Time first* (LWLPT) [Lin and Chen 1996] algorithms. Each design alternative in the form of a software/hardware model pair (SM_i, HM_j) is simulated by scheduling the software on the hardware using each of the above three task scheduling algorithms (TS_k). This simulation results in an execution time $\text{Time}(SM_i, HM_j, TS_k)$ which is used for the final evaluation. The combination of the software model (SM_i), the hardware model (HM_j), and the task scheduling algorithm (TS_k) that gives the best performance, as defined in Equation (14), is the final selected output.

$$\text{perf}(SM_i, HM_j, TS_k) = \frac{1}{\text{Time}(SM_i, HM_j, TS_k) \times \text{Cost}(HM_j)} \tag{14}$$

where $\text{Time}(SM_i, HM_j, TS_k)$ is the execution time of SM_i on HM_j scheduled using TS_k and $\text{Cost}(HM_j)$ is the total hardware cost of the system. The combination that gives the greatest $\text{perf}(SM_i, HM_j, TS_k)$ is the best choice.

For the running example, the results of hardware and software syntheses are given in Figure 2. Cosimulation of the software and hardware solutions

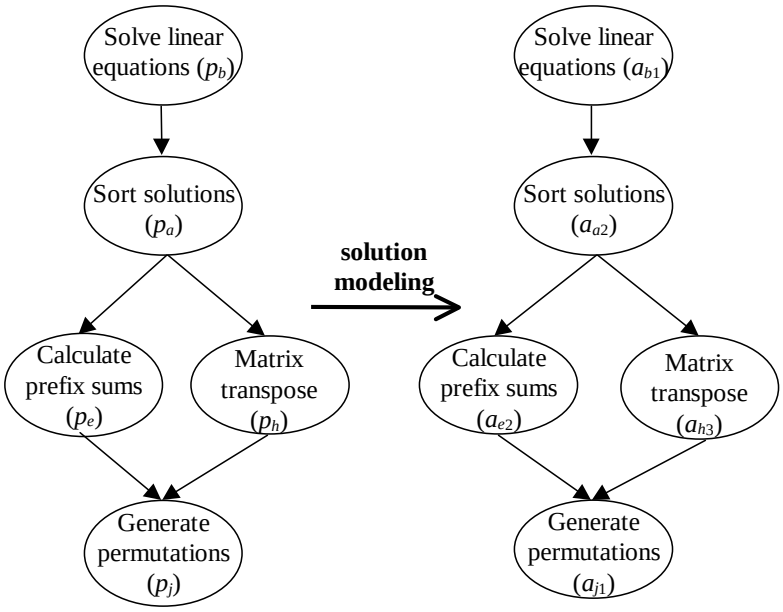

Fig. 5. Small example: problem graph and a solution graph.

is performed by scheduling the software solutions on the hardware systems using three different task scheduling algorithms: LS, LSPF, LWLPT labeled as (TS_1, TS_2, TS_3). Performances are evaluated using Equation (14) and the final result (HM_1, SM_2, TS_3) is the best codesign result since it gives the best performance.

5. APPLICATION EXAMPLES

Three application examples designed using CMAPS are presented. The first running example, presented in the previous section, is a hypothetical one and is used for illustrating the various design phases of CMAPS. Two other examples will be described in this section. They are both real design examples synthesized from problem specifications.

5.1 A Small Real Codesign Example

This is a small problem consisting of five elementary problems, namely, solving a system of linear equations (p_b), sorting the solutions (p_a), computing prefix sums (p_e), matrix transpose (p_h), and generating permutations (p_j). The Problem Graph specification input is given in Figure 5 and the table of elementary problems used in this paper is described in Table V. Here, $V_P = \{p_b, p_a, p_e, p_h, p_j\}$ and $E_P = \{(p_b, p_a), (p_a, p_e), (p_a, p_h), (p_e, p_j), (p_h, p_j)\}$ where $G_P(V_P, E_P)$ is the Problem Graph.

The *Initialization* phase of CMAPS results in three Solution Graphs with the sets of elementary algorithms as follows: SG_1 with $\{a_{b_1}, a_{a_2}, a_{e_2}, a_{h_3}, a_{j_1}\}$,

SG_2 with $\{a_{b_1}, a_{a_3}, a_{e_1}, a_{h_3}, a_{j_1}\}$, and SG_3 with $\{a_{b_1}, a_{a_1}, a_{e_1}, a_{h_3}, a_{j_1}\}$. Only SG_1 is illustrated in Figure 5. We explain in detail the cosynthesis methodology on the illustrated SG_1, leaving the other two in the final results discussion. Here, $V_S = \{a_{b_1}, a_{a_2}, a_{e_2}, a_{h_3}, a_{j_1}\}$, $E_S = \{(a_{b_1}, a_{a_2}), (a_{a_2}, a_{e_2}), (a_{a_2}, a_{h_3}), (a_{e_2}, a_{j_1}), (a_{h_3}, a_{j_1})\}$, and $G_S = (V_S, E_S)$ is the illustrated Solution Graph in Figure 5.

Using Equations (4) and (6), two hardware models, HM_1 and HM_2, are generated from SG_1 in the *Hardware Modeling* subphase in the following table:

	CM	ML	MA	CO
a_{b_1}	10	1000	010	01
a_{a_2}	10	1000	010	01
a_{e_2}	01	0001	001	01
a_{h_3}	10	1000	001	01
a_{j_1}	10	1000	001	01
s_{jk}	41	4001	023	05
n/b_j	2.5	1.25	1.66	2.5
t_{jk}	10	1000	011	01
HM_1	10	1000	010	01
HM_2	10	1000	001	01

The *Software Modeling* subphase mainly involves a replacement of the elementary algorithm a_{e_2} due to it being incompatible with either of the hardware models. Incompatibility results because the elementary algorithm chosen for p_e, i.e. a_{e_2}, requires a message-passing communication model for execution, whereas the two generated hardware models both have shared-memory-based communication model. From Table V and the reselection criteria given in the *Software Modeling* subphase, a_{e_1} is chosen from the Algorithm Database for solving subproblem p_e. In the *Coevaluation* subphase, using Equation (12), the degrees of feasibility of $SM_1 = \{a_{b_1}, a_{a_2}, a_{e_1}, a_{h_3}, a_{j_1}\}$ corresponding to HM_1 and HM_2 are $f_{HM_1} = 0.9$ and $f_{HM_2} = 0.5$, respectively. Hence, based on the criteria given in Equation (13), only HM_1 is selected for further synthesis.

At this stage, the hardware specifications are: a shared-memory communication model, with NUMA mean access latency, a CREW memory access model, and an SIMD machine. The hardware requirements of the software algorithms are: any shared-memory-based system interconnection network and the number of processors required is n^2, where n is the problem size (here, it was specified as $n = 100$). Based on the user constraints specification of maximum hardware cost ($100,000 in this example) and performance specifications of 950 MFlops throughput, the derived specifications was input to the ICOS methodology [Hsiung et al. 1998]. This resulted in a multiprocessor system consisting of 10,000 processors, a generalized cube-based multistage interconnection network connecting the processors with a memory of 1024 MB RAM and 10 MB cache memory.

Software synthesis subphase consists of how the five elementary algorithms a_{b_1}, a_{a_2}, a_{e_1}, a_{h_3}, a_{j_1} are composed into a single feasible pseudocode solution. The five elementary algorithms are all from Table V, the source of which is Akl's book on parallel algorithms [1989]. The algorithms are, respectively, an algorithm to solve a set of linear equations (SIMD GAUSS-JORDAN), a parallel sorting algorithm (CREW SORT), a parallel summation algorithm (PARALLEL SUMS), a parallel matrix transpose algorithm (EREW TRANSPOSE), and a parallel permutation generation algorithm (FULL PERMUTATION). The outputs of each algorithm must be input to its one or more successor algorithms in the solution graph. Code interfaces are required to accomplish this task and since it is not an easy job to maintain the consistencies between algorithms, this portion of software synthesis is still an open problem of how algorithms may be interfaced together. Currently, our implementation is only semiautomatic; it only checks the data-type consistencies. As far as logical consistency and other types of interface problems are concerned, it still basically depends on the user-provided codes for interfacing. In Figure 6, the user specifies that A and b are two $n \times n$ and $n \times 1$ matrices, the first problem is to solve the matrix equation $Ax = b$. The resulting solution vector $\vec{x}$ is then transposed (into vector $\vec{y}$) and input to the CREW SORT sorting algorithm which sorts the vector in an ascending order of element values and then prefix sums are calculated in parallel by the PARALLEL SUMS algorithm for the sorted vector ($\vec{z}$). The $n \times n$ product matrix $\vec{x}\vec{z}$ is then transposed using the EREW TRANSPOSE algorithm which results in a single value (i.e., a 1×1 matrix) and then permutations are generated in parallel for that single value by the FULL PERMUTATION algorithm. Algorithm interfaces required in this example are how the solution vector $\vec{x}$ is transposed before input to the CREW SORT algorithm, and how the resulting vectors $\vec{x}$ and $\vec{z}$ are multiplied before input to the EREW TRANSPOSE algorithm.

The above-detailed discussion was on how SG_1 is cosynthesized, resulting in a hardware-software model pair HM_1/SM_1. The cosyntheses for SG_2 and SG_3 are similar and not discussed here. The *modeling and evaluation* phase results in the model pairs HM_1/SM_1, HM_3/SM_2, and HM_3/SM_3, where $SM_2 = \{a_{b_1}, a_{a_4}, a_{e_1}, a_{h_3}, a_{j_1}\}$ and $SM_3 = \{a_{b_1}, a_{a_1}, a_{e_1}, a_{h_3}, a_{j_1}\}$. Under the constraints given in this example: maximum cost of $100,000 and minimum throughput of 950 MFlops, the final synthesis and simulation phase results are tabulated in Table I, where we can observe that only the model-pairs HM_1/SM_1 and HM_3/SM_2 satisfy the constraints, of which the former is the heuristically optimal architecture according to the "perf" metric calculated using Equation (14). Hence, the final result is a parallel system with shared-memory, NUMA memory latency, CREW memory access, SIMD control scheme, and 100,000 processors costing totally $9800 and a pseudo-program solution as given in Figure 6.

Table I. Simulation Results of Example 2

Model Pairs	Time			Cost	Perf	MFlops
	TS_1	TS_2	TS_3			
HM_1/SM_1	500	450	520	9800	2.26×10^{-7}	1003
HM_3/SM_2	700	720	760	9000	1.58×10^{-7}	977
HM_3/SM_3	800	912	788	7800	1.62×10^{-7}	946

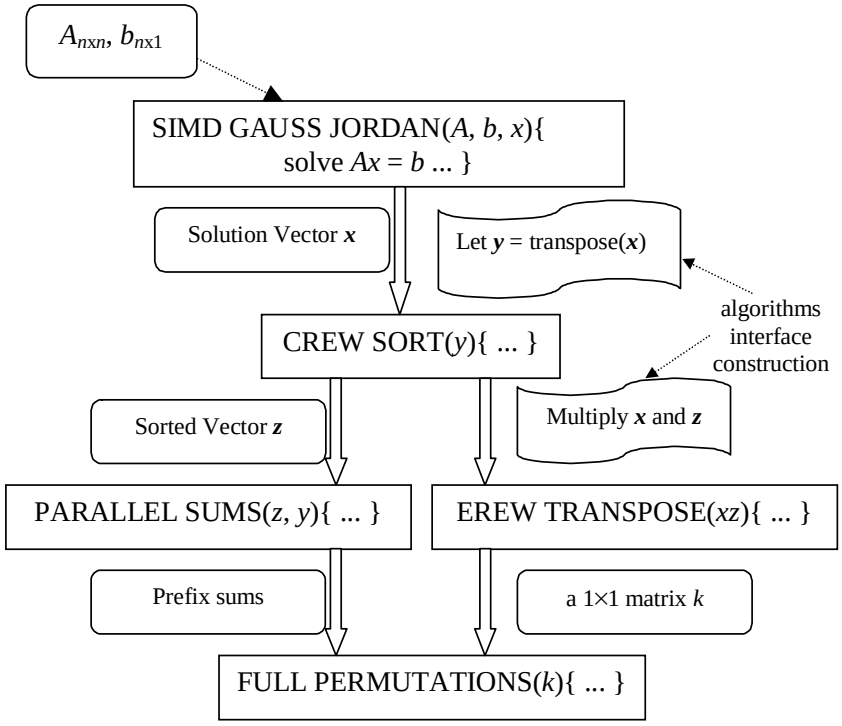

Fig. 6. Small example: software synthesis.

5.2 A Large Real Codesign Example

This example illustrates the scalability of the presented methodology by synthesizing a large target system that can solve a problem with nine elementary problems. The problem graph specification input is shown in Figure 7.

In the *Initialization* phase, eight *Solution Graphs* are generated using the solution modeling process. SG_1 is illustrated in Figure 8 and the others are tabulated in Table II. For ease of illustration, only SG_1 is described in a little more detail, while the results of the others are given in the final simulation discussion. Hardware models are then generated using Equations (4) and (6) for SG_1 as follows:

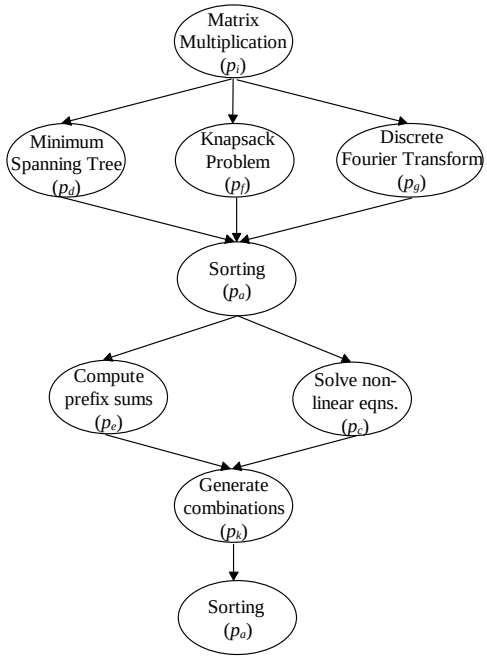

Fig. 7. Large example: problem graph.

	CM	ML	MA	CO
a_{i_3}	10	1000	100	01
a_{d_1}	10	1000	001	01
a_{f_2}	01	0001	001	01
a_{g_2}	01	0001	001	01
a_{a_2}	10	1000	010	01
a_{e_1}	10	1000	001	01
a_{c_1}	10	1000	010	01
a_{k_2}	10	1000	001	01
a_{a_2}	10	1000	010	01
s_{jk}	72	7002	135	09
n/b_j	4.5	2.25	3	4.5
t_{jk}	10	1000	011	01
HM_1	10	1000	010	01
HM_2	10	1000	001	01

Referring to Table V and using the `model_solution()` process given in Section 4, there are two choices for the first subalgorithm (a_{i_3} or a_{i_1}), the sixth subalgorithm (a_{e_1} or a_{e_2}), and the seventh subalgorithm (a_{c_1} or a_{c_2}). The algorithm combination given in Figure 8 is selected as an initial solution, if this *Solution Graph* cannot be made feasible after the generation of hardware models, then another initial solution can be generated and used.

Software modeling produces two software models, one for each of the hardware models as follows: (1) for HM_1: since no choice of shared-memory

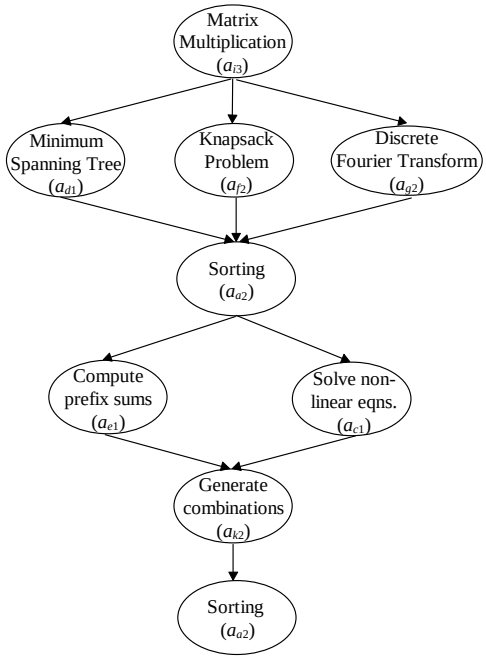

Fig. 8. Large example: a solution graph.

Table II. Solution Graphs for Example 3

Problem	**Solution Graphs**							
	SG_1	SG_2	SG_3	SG_4	SG_5	SG_6	SG_7	SG_8
p_i	a_{i_3}	a_{i_3}	a_{i_3}	a_{i_3}	a_{i_1}	a_{i_1}	a_{i_1}	a_{i_1}
p_d	a_{d_1}	a_{d_1}	a_{d_1}	a_{d_1}	a_{d_1}	a_{d_1}	a_{d_1}	a_{d_1}
p_f	a_{f_2}	a_{f_2}	a_{f_2}	a_{f_2}	a_{f_2}	a_{f_2}	a_{f_2}	a_{f_2}
p_g	a_{g_2}	a_{g_2}	a_{g_2}	a_{g_2}	a_{g_2}	a_{g_2}	a_{g_2}	a_{g_2}
p_a	a_{a_2}	a_{a_2}	a_{a_2}	a_{a_2}	a_{a_2}	a_{a_2}	a_{a_2}	a_{a_2}
p_e	a_{e_1}	a_{e_1}	a_{e_2}	a_{e_2}	a_{e_1}	a_{e_1}	a_{e_2}	a_{e_2}
p_c	a_{c_1}	a_{c_2}	a_{c_1}	a_{c_2}	a_{c_1}	a_{c_2}	a_{c_1}	a_{c_2}
p_k	a_{k_2}	a_{k_2}	a_{k_2}	a_{k_2}	a_{k_2}	a_{k_2}	a_{k_2}	a_{k_2}
p_a	a_{a_2}	a_{a_2}	a_{a_2}	a_{a_2}	a_{a_2}	a_{a_2}	a_{a_2}	a_{a_2}

algorithm exists in A (Table V), we modify a_{f_2} and a_{g_2} such that they can be executed on a shared-memory architecture, and we serialize writes in a_{i_3}, (2) for HM_2: in addition to the above three modifications, two more are needed: serialize reads in a_{a_2} and a_{c_1}. Coevaluation of the hardware and software models based on the criteria in Equation (13) results in HM_1 being selected, because the degrees of feasibility of the hardware models as calculated using Equation (12) are $f_{HM_1} = 6/9$ and $f_{HM_2} = 3/9$.

For hardware synthesis, the initial set of interconnections as required by the hardware model HM_1 is {`SM-SI, Mesh`}, where `SM-SI` is any shared-

Table III. Hardware Models for Example 3

SG	Models	CM	ML	MA	CO	Status
SG_1	HM_1	10	1000	010	01	feasible
	HM_2	10	1000	001	01	eliminated
SG_2	HM_3	10	1000	001	01	feasible
SG_7	HM_4	10	1000	001	01	feasible
	HM_5	10	1000	001	01	feasible
	HM_6	01	1000	001	01	inconsistent
	HM_7	01	1000	010	01	inconsistent
	HM_8	10	0001	001	01	inconsistent
	HM_9	10	0001	010	01	inconsistent
	HM_{10}	01	0001	001	01	feasible
	HM_{11}	01	0001	010	01	feasible
SG_8	HM_{12}	10	1000	001	01	feasible
	HM_{13}	10	0001	001	01	inconsistent
	HM_{14}	10	0001	001	01	inconsistent
	HM_{15}	01	0001	001	01	feasible

memory system interconnection. Using set reduction technique, `Mesh` can be emulated by `SM-SI` so `Mesh` is removed and the final singleton set is {`SM-SI`}, that is, any shared-memory interconnection can be used. We let $N = n^3$ and thus the final number of processors is n^3, where n is the problem size which is 64 in this example, thus totally 218 processors are used. Software synthesis involves interface construction, serializing read and write accesses, emulation of message-passing algorithms on shared-memory architectures, and human intervention and interpretation.

Having discussed the cosynthesis of SG_1, we come to the overall design alternatives of this example. There were totally 15 hardware models generated for the 8 solution graphs, but 6 were eliminated due to architectural inconsistencies (i.e., such as a message-passing system with shared-memory) and one was eliminated to reduce the design space by using the criteria in Equation (13), thus there were 8 feasible hardware models. There were three corresponding software models. The hardware models generated in this example are given in Table III.

Finally, the synthesized hardware and software are simulated and performance evaluated as shown in Table IV. Out of the seven design alternatives simulated, only five of them, namely, HM_1/SM_1, HM_4/SM_2, HM_5/SM_2, HM_{12}/SM_3, and HM_{13}/SM_3, satisfy the user constraints which are a maximum cost of \$6,000 and a minimum throughput of 400 MFlops. Comparing the "perf" metric, the first design alternative in Table IV is the heuristically most optimal solution. The final solution is thus a set of hardware system-level specifications: shared-memory architecture, nonuniform memory access latency, CREW memory access, SIMD control scheme, 218 processors, a pseudoprogram solution, and the LSPF task scheduling algorithm.

The above-presented two examples, respectively, show the feasibility and the scalability of CMAPS. Besides high scalability in solving complex problems and easy upgradability to new technologies, some other advan-

Table IV. Simulation Results for Example 3

Model Pairs	Time			Cost	Perf	MFlops
	TS_1	TS_2	TS_3			
HM_1/SM_1	360	376	347	5200	5.54×10^{-7}	516
HM_4/SM_2	406	401	380	4800	5.48×10^{-7}	491
HM_5/SM_2	402	390	442	5200	4.93×10^{-7}	505
HM_{10}/SM_2	600	504	490	5400	3.78×10^{-7}	394
HM_{11}/SM_2	579	650	625	5700	3.03×10^{-7}	399
HM_{12}/SM_3	471	479	499	4800	4.42×10^{-7}	506
HM_{13}/SM_3	505	500	506	5400	3.70×10^{-7}	442

Table V. The Three Repositories: PD, AD, and MD. Parallel Algorithms Source: Akl's Book [Akl 1989]

$p\#$	Problem Names	$a\#$	$t(n)$	$u(n)$	*CM*	*ML* or *SI*	*MA*	*CO*
p_a	Sorting a sequence	a_{a_1}	O(1)	n^2	SM	NUMA	CRCW	SIMD
		a_{a_2}	O ($n \log n$)	N	SM	NUMA	CREW	SIMD
		a_{a_3}	O (n)	n	SM	NUMA	EREW	SIMD
p_b	Solving systems of linear equations	a_{b_1}	O (n)	n^2	SM	NUMA	CREW	SIMD
		a_{b_2}	N/A	N	SM	NUMA	CREW	MIMD
p_c	Finding roots of nonlinear equations	a_{c_1}	O ($\log_{N+1} w$)	N	SM	NUMA	CREW	SIMD
		a_{c_2}	N/A	N	SM	NUMA	CRCW	SIMD
p_d	Minimum spanning tree	a_{d_1}	O (n^2/N)	N	SM	NUMA	EREW	SIMD
p_e	Prefix sums	a_{e_1}	O ($\log n$)	n	SM	NUMA	EREW	SIMD
		a_{e_2}	O ($\log n$)	$2n-1$	MP	Tree	EREW	SIMD
		a_{e_3}	O ($n^{1/2}$)	n	MP	Mesh	EREW	SIMD
p_f	Knapsack problem	a_{f_1}	O (n)	n	MP	Tree	EREW	SIMD
		a_{f_2}	O ($n^{1/2}$)	n	MP	Mesh	EREW	SIMD
p_g	Discrete Fourier Transform	a_{g_1}	O ($\log n$)	n^2	MP	Mesh of trees	EREW	SIMD
		a_{g_2}	O ($n^{1/2}$)	n	MP	Mesh	EREW	SIMD
p_h	Matrix transpose	a_{h_1}	O (n)	n^2	MP	Mesh	EREW	SIMD
		a_{h_2}	O ($\log n$)	n^2	MP	Shuffle	EREW	SIMD
		a_{h_3}	O(1)	n^2	SM	NUMA	EREW	SIMD
p_i	Matrix multiplication	a_{i_1}	O (n)	n^2	MP	Mesh	EREW	SIMD
		a_{i_2}	O ($\log n$)	n^3	MP	Cube	EREW	SIMD
		a_{i_3}	O(1)	n^3	SM	NUMA	CRCW	SIMD
p_j	Permutations	a_{j_1}	O (${}^nP_m \log m$)	m	SM	NUMA	EREW	SIMD
		a_{j_2}	O ($\lceil n!/N \rceil n$)	N	SM	NUMA	EREW	SIMD
p_k	Combinations	a_{k_1}	O (${}^nC_m \log m$)	m	SM	NUMA	EREW	SIMD
		a_{k_2}	O ($\lceil {}^nC_m/N \rceil m$)	N	SM	NUMA	EREW	SIMD
p_l	Convolution	a_{l_1}	O (n)	n	MP	Linear array	EREW	SIMD
		a_{l_2}	O (n)	n	MP	Tree	EREW	SIMD

tages of CMAPS are as follows. Firstly, CMAPS is a complete codesign methodology which can synthesize a parallel computer architecture start-

ing from user requirements instead of detailed system specifications, thus CMAPS has increased the degree of automation possible in the synthesis process. Secondly, the interleaving of hardware and software modeling phases and the coevaluation of model pairs is a novel way of *model-based codesign* which has the advantage of reducing the design space by eliminating contradictory models at an early stage. Thirdly, CMAPS is the first methodology that can be used to design the hardware and the software of both AOGPP and ASP systems. Lastly, by integrating specification, synthesis, and simulation into a single design environment, CMAPS ensures that there is no semantic loss when transiting from one design phase to another as is often observed when output designs have to be transformed into a different input format either for further synthesis or simulation. But, at the same time, CMAPS being a pioneer effort in the new direction of synthesizing a system starting from user requirements, there is much work left to be done; for example, more general scheduling policies must be used to simulate the generated hardware-software models; the coevaluation process should be more thorough, taking into account the implicit interaction possible between the hardware and the software models; and furthermore, the three repositories should be made more general in order to cover more and larger parallel systems.

6. CONCLUSION AND FUTURE WORK

Unlike traditional synthesis methods, an effort was made to synthesize a system starting directly from a user's requirements, in the form of an application problem, rather than from detailed behavioral or architectural specifications. Such an approach simplifies the user input and increases the degree of design automation possible in hardware-software cosynthesis.

Motivated by designing a system starting from user requirements rather than system-level specifications, the work presented in this paper can be summarized as follows: given an application problem, specified as a directed acyclic graph of interconnected elementary problems, a software-hardware solution is synthesized such that the synthesized software (a parallel pseudoprogram), when scheduled using a multiprocessor task scheduling algorithm on the synthesized hardware (a parallel system), optimally solves the given problem. Designers catering to the diverse requirements of a user can save considerable cost and design efforts by synthesizing one application-oriented general-purpose parallel (AOGPP) system instead of several specialized systems. An AOGPP system is defined to be a general-purpose parallel system whose subsystem are configured for solving a given application problem. Specialized application-specific systems can only solve the problem they were designed for, whereas AOGPP systems, being general-purpose, can be used to solve any other problem besides optimally solving the given application problem.

A methodology called *Cosynthesis Methodology for Application-Oriented Parallel Systems* (CMAPS) is presented for synthesizing both the software and hardware of an AOGPP systems. CMAPS uses an iterative procedure

beginning with a solution graph and going through interleaved phases of software and hardware modeling. The software-hardware model combinations are coevaluated in order to decrease the size of the design space to be explored. Hardware and software are then synthesized separately and cosimulated by scheduling the synthesized software on the hardware using multiprocessor task scheduling algorithms.

Complex problems can be solved by CMAPS due to the scalability achieved through modularization of the problem input specification (Problem Database), of the hardware system (Model Database), and of the software system (Algorithm Database). These three repositories, constructed using Object-Oriented (OO) technology [Hsiung, Lee, and Chen 1997b], also contribute toward easily upgrading to new technologies such that new hardware components, new algorithms, and new elementary problems can always be integrated into existing repositories.

Future work in this direction would be applying OO technology not only to the repositories, but to the codesign process itself [Wolf 1996]. Hardware and software dependence on each other also need further investigation. A formal verification model [Hsiung, Lee, and Chen 1997a] for the codesign of AOGPP systems would also be an interesting research topic.

REFERENCES

AKL, S. G. 1989. *The Design and Analysis of Parallel Algorithms*. Prentice-Hall, Inc., Upper Saddle River, NJ.

BERGE, J. M., LEVIA, O., AND ROUILLARD, J. 1997. *Hardware/Software Co-Design and Co-Verification*. Kluwer Academic Publishers, Hingham, MA.

BIRMINGHAM, W. P., GUPTA, A. P., AND SIEWIOREK, D. P. 1989. The MICON system for computer design. In *Proceedings of the 26th ACM/IEEE Conference on Design Automation* (DAC '89, Las Vegas, NV, June 25–29, 1989), D. E. Thomas, Ed. ACM Press, New York, NY, 135–140.

CHU, W. W. AND LAN, M.-T. 1987. Task allocation and precedence relations for distributed real-time systems. *IEEE Trans. Comput. C-36*, 6 (June 1987), 667–679.

ERNST, R., HENKEL, J., AND BENNER, T. 1993. Hardware-software co-synthesis for microcontrollers. *IEEE Des. Test 10*, 4 (Dec. 1993), 64–75.

FLYNN, M. J. 1972. Some computer organizations and their effectiveness. *IEEE Trans. Comput. C-21*, 9 (Sept. 1972), 948–960.

FORTUNE, S. AND WYLLIE, J. 1978. Parallelism in random access machines. In *Proceedings of the 10th Symposium on Theory of Computing* ACM Press, New York, NY, 114–118.

GADIENT, A. J. AND THOMAS, D. E. 1993. A dynamic approach to controlling high-level synthesis CAD tools. *IEEE Trans. Very Large Scale Integr. Syst. 1*, 3 (Sept.), 328–341.

GRAHAM, R. L. 1969. Bounds on multiprocessing timing anomalies. *SIAM J. Appl. Math. 17* (1969), 416–429.

GUPTA, A. P., BIRMINGHAM, W. P., AND SIEWIOREK, D. P. 1993. Automating the design of computer systems. *IEEE Trans. Comput.-Aided Des. Integr. Circuits 12*, 4 (Apr.), 473–487.

GUPTA, R. K. AND DE MICHELI, G. 1993. Hardware-software cosynthesis for digital systems. *IEEE Des. Test 10*, 3 (Sept. 1993), 29–41.

HARRIS, T. J. 1994. A survey of PRAM simulation techniques. *ACM Comput. Surv. 26*, 2 (June 1994), 187–206.

HSIUNG, P.-A., CHEN, C.-H., LEE, T.-Y., AND CHEN, S.-J. 1998. ICOS: an intelligent concurrent object-oriented synthesis methodology for multiprocessor systems. *ACM Trans. Des. Autom. Electron. Syst. 3*, 2, 109–135.

HSIUNG, P.-A., CHEN, S.-J., HU, T.-C., AND WANG, S.-C. 1996. PSM: an object-oriented synthesis approach to multiprocessor system design. *IEEE Trans. Very Large Scale Integr. Syst. 4*, 1, 83–97.

HSIUNG, P.-A., LEE, T.-Y., AND CHEN, S.-J. 1997. MOBnet: An extended Petri net model for the concurrent object-oriented system-level synthesis of multiprocessor systems. *IEICE Trans. Inf. Syst. E80-D*, 2 (Feb.), 232–242.

HSIUNG, P. -A., LEE, T. -Y., AND CHEN, S. -J. 1997. An object-oriented technology transfer to multiprocessor system-level synthesis. In *Proceedings of the 24th International Conference on Technology of Object-Oriented Languages and Systems* (Sept.) 339–348.

HWANG, K. 1993. *Advanced Computer Architecture: Parallelism, Scalability, Programmability*. McGraw-Hill, Inc., New York, NY.

KALAVADE, A. AND LEE, E. A. 1993. A hardware-software codesign methodology for DSP applications. *IEEE Des. Test 10*, 3 (Sept. 1993), 16–28.

KUMAR, S., AYLOR, J. H., JOHNSON, B. W., AND WULF, W. A. 1993. A framework for hardware/software codesign. *IEEE Computer 26*, 1, 39–45.

LIN, J. -F. AND CHEN, S. -J. 1996. An analysis of multiprocessor tasks scheduling. *Comput. Syst. Sci. Eng. 11*, 2, 117–120.

LIN, J. -F., SEE, W. -B., AND CHEN, S. -J. 1995. Performance bounds on scheduling parallel tasks with communication cost. *IEICE Trans. Inf. Syst.* (Mar. 1995), 263–268.

PRAKASH, S. AND PARKER, A. C. 1992. SOS: synthesis of application-specific heterogeneous multiprocessor systems. *J. Parallel Distrib. Comput. 16*, 4 (Dec.), 338–351.

ROZENBLIT, J. AND BUCHENRIEDER, K. 1995. *Codesign: Computer-Aided Software/Hardware Engineering*. IEEE Press, Piscataway, NJ.

THOMAS, D., ADAMS, J., AND SCHMIT, H. 1993. A model and methodology for hardware/software codesign. *IEEE Des. Test*, 6–15.

ULLMAN, J. 1975. NP-complete scheduling problems. *J. Comput. Syst. Sci. 10*, 384–393.

VAHID, F., GAJSKI, D. D., AND GONG, J. 1994. A binary-constraint search algorithm for minimizing hardware during hardware/software partitioning. In *Proceedings of the European Conference on Design Automation* (EURO-DAC '94, Grenoble, France, Sept. 19–23, 1994), J. Mermet, Ed. IEEE Computer Society Press, Los Alamitos, CA, 214–219.

WOLF, W. 1994. Hardware-software co-design of embedded systems. *Proc. IEEE 82*, 7 (July 1994), 967–989.

WOLF, W. 1996. Object-oriented cosynthesis of distributed embedded systems. *ACM Trans. Des. Autom. Electron. Syst. 1*, 3, 301–314.

WOLF, W. H. 1997. An architectural co-synthesis algorithm for distributed, embedded computing systems. *IEEE Trans. Very Large Scale Integr. Syst. 5*, 2, 218–229.

XIONG, X., BARROS, E., AND ROSENSTIEL, W. 1994. A method for partitioning UNITY language in hardware and software. In *Proceedings of the European Conference on Design Automation* (EURO-DAC '94, Grenoble, France, Sept. 19–23, 1994), J. Mermet, Ed. IEEE Computer Society Press, Los Alamitos, CA, 220–225.

YEN, T.-Y. AND WOLF, W. 1995. Communication synthesis for distributed embedded systems. In *Proceedings of the 1995 IEEE/ACM International Conference on Computer-Aided Design* (ICCAD-95, San Jose, CA, Nov. 5–9), R. Rudell, Ed. IEEE Computer Society Press, Los Alamitos, CA, 288–294.

YEN, T.-Y. AND WOLF, W. 1995. Sensitivity-driven co-synthesis of distributed embedded systems. In *Proceedings of the Eighth International Symposium on System Synthesis* (Cannes, France, Sept. 13–15, 1995), P. G. Paulin and F. Mavaddat, Eds. ACM Press, New York, NY, 4–9.

YEN, T. -Y. AND WOLF, W. 1996. *Hardware-Software Co-Synthesis of Distributed Embedded Systems*. Kluwer B.V., Deventer, The Netherlands.

Received: July 1997; revised: December 1997; accepted: May 1998