Hardware/Software Partitioning in Co-Design

Nitin K. Agarwal and Divya Jain
2001MCS025 —– 2001MCS022

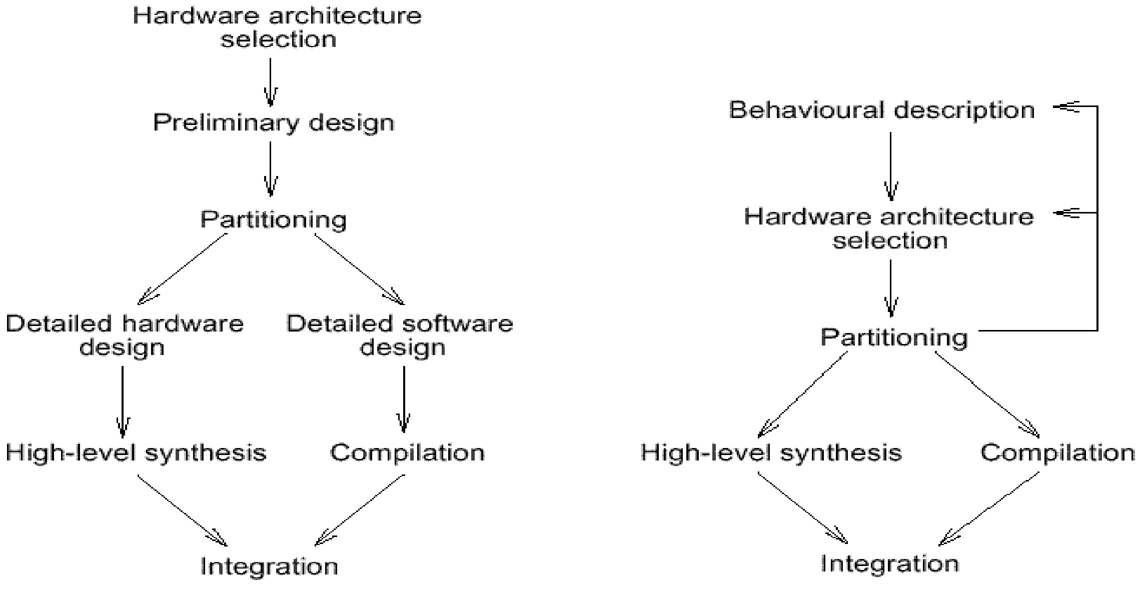
April 26, 2002

Contents

1 Introduction

The field of hardware/software codesign has received increasing attention during the last few years, and numerous publications have appeared describing direct systems and methodologies (a recent bibliography 1 lists over 250 publications related to codesign). Most approaches are based on the fact that a single executable description of a system can nowadays be compiled into either silicon or machine code, and this opens up the possibility to uniformly describe the behaviour of a system, which will be implemented in a combination of application-specific hardware and software. This description is then partitioned, with assistance from more or less automatic tools, into separate hardware and software parts, which are passed on to a high-level synthesis tool respectively a compiler to produce the final implementation.

One class of applications particularly well suited for hardware/software codesign is embedded real- time systems. These systems have well-defined timing constraints, which are sometimes so severe that they can only be met in a hardware implementation, but at the same time they might have less time-critical parts that can benefit from the lower cost and higher flexibility of a software solution. Therefore, heterogeneous implementations are often necessary for these applications, and are indeed frequently employed today. But there is a lack of design methodologies and tools that can support their implementation across the hardware/software technology barrier.



(a) Traditional design (b) HW/SW codesign

Figure 1: Traditional design (a) compared to hardware/software codesign (b).

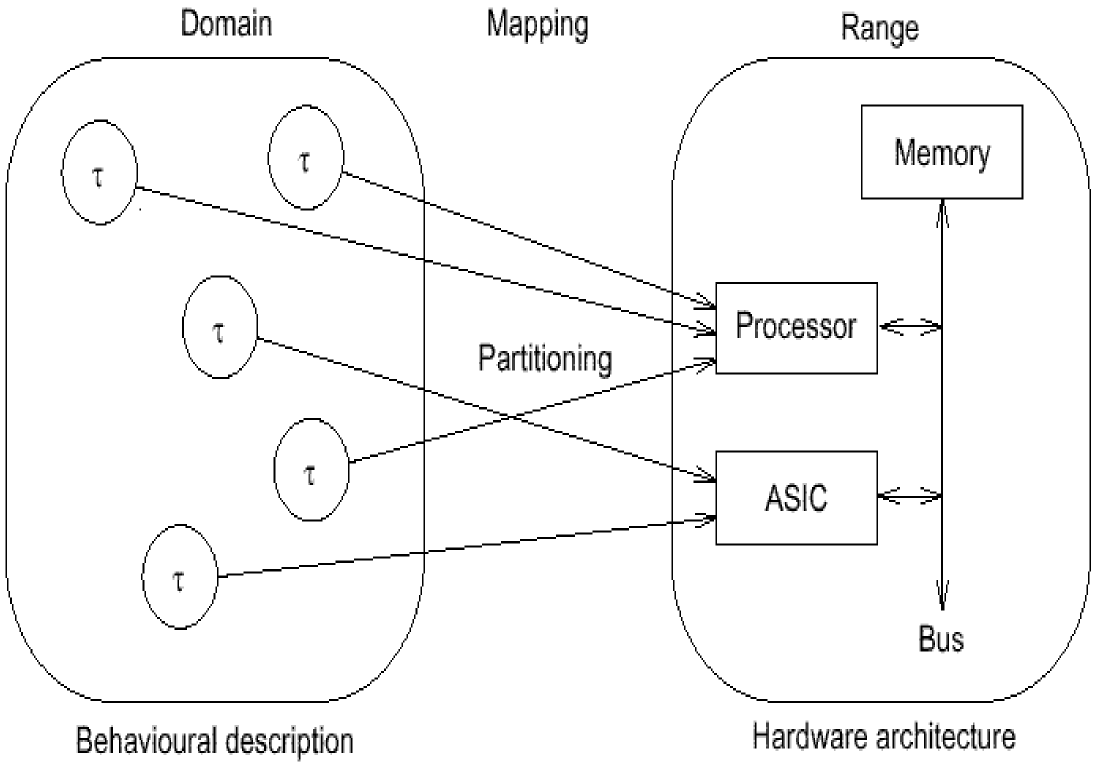
Figure 1 illustrates how the current practice of real-time systems design (adapted from Lawrence and Mauch 2) compares with codesign. Major differences are that the hardware architecture selection and the partitioning are performed later than in traditional design, thus allowing these design decisions to be made basedon more complete and accurate information.And in the co-design partitioning process itrate over over and try to get a optimal solution.It explore the design space and do estimation to get optimal solution.

Our discussion go in the following way. Next subsection describe partitionign in more detail. Then We discuss in more detail the issues involveed in partitioning process. Then we disscuss the actual algorithms that are used to partion the given system.After that we shed some light on some real partitioning tool We chossen CUSYMA and LYCOS.

2 Partitioning Process

The partitioning is a way of deciding for each part/task of the behavioural description, which part of the microprocessor or ASIC of the hardware architecture should be used to implement it. Therefore the partitioning can be thought of as a mapping, whose domain is the set of tasks in the behavioural description, and whose range is the set of components with processing capabilities in the hardware architecture (see Figure 2). Similarly, all variables in the behavioural description have to be mapped to elements in the architecture with storage capacity, i.e. memories or ASICs. The problem of constructing this mapping has an exponential complexity: if there are k components and m task, there are O(k^m) different ways to partition. But fortunately, the problem lends itself well to a solution using a branch- and-bound algorithm, which only needs to evaluate a small fraction of all combinations in order to find the optimum.

The goal function that we try to optimize for partitioning is the schedulability of the system, captured by the minimal required speedup S of the task set. In this way, the partitioning is dependant on the performance estimations which are based on the models of the hardware architecture and scheduler. But often there might be other criteria besides performance that are relevant to the partitioning, but that can be difficult to quantify. As an example, the designer may know that one task in the behavioural description is likely to change in future versions of the system, and it is therefore preferably implemented in software. This, together with other similar arguments, leads us to the conclusion that the designer must be given the opportunity to manually guide the partitioning processAnother consideration is the partitioning grain size. We have chosen tasks as the entities to assign to hardware or software, but one could of course use smaller parts, e.g. single instructions. The advantage of a smaller grain size is that we can adjust the partitioning more exactly than with a coarser grain. But on the other hand, the data dependencies between instructions are often much stronger than between tasks, so if we move some instructions to hardware, it can be problematic to execute hardware and software parts in parallel. We can also expect to get a larger runtime overhead with a smaller grain. Partitioning on task level on the other hand gives the user some control over the grain size, since he can often choose, to some extent, how big the tasks should be. On these grounds, we feel that a task-level partitioning is justified.



So This the basic partitioning process.Now,we take different issues in this process.

Nature of the applications

Target Architecture

Closeness Matrics and Cost Function

Granularity and Estimations

Structural Vs Functional Partitioning

Actual Algorithms

Partitioning Tool

3 Different Issue involved in Partitioning Process

3.1 Nature of the application

The main difference between general-purpose ,highest volume microprocessors and embedded systems is specialization.We can optimize an embedded system for a given application or a class of applications.So given the specialization technique ,designer should identify the potential of the target applications with restect to specialization technique before a architecture selection is made.More over flexibility should not be compromised.i.e.it should cover the whole family of applications and should allow the changes.This also limits the specializations.What is needed is a terget applications analysis which idenfies tha applications properties that can be used for specializations, quantifies the individual specializations effects and takes flexibility requirements into account.Unfortunately,There is no formal method to this task.This is moreorless depends on the designer's experiences. We will distinguish four application system classes.

Computations oriented System such as workstations,PCs or scientific parallel computer.

Control Dominated System, which basically react to external events with relatively simple processes.

Data-dominated System ,where the focus is on complex transformations or transportations of data, such as signal processing, or packet routing.

Mixed systems such as mobilephones or motor which contain extensive operations on data as well as reaction to external events

3.2 Target Architecture

3.2.1 Architecture for control-dominated system

Control-dominated systems are reactive systems showing a behavior which is driven by external events.The typical underlying semantics of the system description, the input model of computatio, are coupled FSMs or Petri nets Examples are StateCharts or Esterl processes.Time constraints are given as the maximum or minimum times.These shall be mapped to communicating hardware and software components possibly after a trasformations which decomposes and / or merges part of the FSMs(of the Petri net)

The resulting co-design problems are the execution of the concurrent FSMs. (or Petrinet transitions) reacting to asynchronus input events, which have an input data dependent control flow and are subject to time constraints. Very often control applications are safety critical(automotive, avionics) .So, correctness(functions and timings),FSM transition synchronisation and event scheduling are the major problems while computations requirements are slow.

So for the trasformation ,each FSM is mapped to a process.Thses process contains small variable set containing the FSM state and FSM transitions are mapped as a short program segments in chains of branches or in jump tables.Program size grows with the number of ststes and transitions.In microcontrlllers,program memories often require more space than the CPU .So code size in an important issue.In microcontroller architecture, FSM communications uses shared memory.Since each FSM transition typically require a context switch ,context switching times are critical. This short application overview reveals a number of specialization opprtunities.

most of the operations are Boolean operations or bit manipulations

there are few operations per state transition the data set containing the FSM states is typically small.

Examples of the such microcontrollers are 8051 , Motorola MC68332 , Siemens 80C165

3.2.2 Architecture for data-dominated system

The term data-dominated system can be used for system where data transport requirements are dominant for the design process(e.g. in telecommunication) or where the data processing is the dominant requirement(e.g. in signal processing).In this overview, we want to focus on static data flow syatem i.e. system with periodic behavior of all the system parts. Often such systems are described in flow graph language such as in signal flow graph which is comman in digital signal processing or control engineering.Similar to data flow grapha, signal flow graphs represent data flow and functions to be executed on data.

Such data-dominated applications typically have a demand for high computation performance with an emphasis on high throughput rather than short latency deadline.Large data variables,such as vedio frames in image processing, require memory optimization problems. The flow graph input model of computation offers many opprtunities for specializations.

The periodic execution often corresponds to a (mostly) input independent system data flow.The input data is generated synchronously with a fixed period, the sample rate. Such system can be mapped to a static schedule thereby applying transformations for high concurrency,such as loop unrolling.The resulting static control and data flows allows to specialize control, interconnect function units and data path while, at most, a limited support for interrupts and process context switch is required.Furthermore, static scheduling leads to regular and a priori known address sequences and operations which gives the opprtunity to specialize the memory architecture and address unit. Digital signal processors (DSP) are the most prominent examples of the target architecture for this class of the applications.ADSp21060 and TMS3200C80 are the good examples of the DSP based processors.

3.2.3 Mixed and less specialized system

Many system such as for mobile communication where signal processing must executed in parallel to an event driven complex protocol, and contain interconnected data-dominated and control-dominated functions. There are approaches to such system. One approach is to design a hetrogeneous system consisting of independently controlled, communicating specialixed components, such as DSPs,microcontrollers and peripheral processors.This appreach is migrating tosystem-on-a-chip.Such a decomposition in different functions is possible and efficient.The second approach is to limit specialization.Dependent on the design objectives, such less specialized system can still be tailored to an application domain.We can distinguish.

computation application without specific potential Application examples are printer or scanner controller,game controller,etc.Such system often use modified version of standard general purpose architecture which are selected according to the computation requirement, such as the **M68000** , the**MIPS3000**,the **PowerPC** architecture, e.g. the **MPC8xx** (RCPU) or the **Alpha** 21066

application with specific non-functional constraints Most popular are low power application (portable system). Examples are general purpose architecture which are tailored to minimize power consumption and cost (cost sensitive application) for required level of performance, such as the **ARM** family of processors, or the Motorola **ColdFire** family with an instruction set which is opward to the M68xxx.

mixed system A widely used example of aprocessor family for mixed application is the **Intel i960**

3.2.4 Modern Embedded Architecture

This subsection presents a description of trends in architecture used in today's product in multimedia, wireless communication, and telecommunication.Customized cores are increasingly used to meet cost/performance goals.

3.2.5 Architecture in Multimedia, Wireless and Telecommunications

Market for workstation/PC mocroprocessors is driven by compatibility.In contrast, embedded architecture show a very different picture-custom programmable arcitecture tuned to application types(ASIPs) are containing many application domain.In multimedia application such as MPEG-1 and 2 encoding and decoding, Dolby AC-3, and videotelephones,many companies are tuning toward custom programmable solution.Some specific examples are: the Philips TM a custom VLIW processor for MPEG-1 and MPEG2 coding and decoding.Chromatic Mpact media processor which performs MPEG-1 and 2 video decoding, the NTT MPEG-2 video encoder chip set which uses a set of custom processor, and the TCEC VLIW processor for MPEG-2 Audio,Dolby AC-3, and Dolby Pro-logic decoding.Thses are just a few of the ASIP designs being used in multimedia.

Custom programmable architecture are popular in multimedia system for three main reasons.First,a tailored architecture is typically more cost-effective than general purpose architecture.This is particularly true for a second,cost reduced design which can often be the major source of revenue.Second, de signers can make considerable performance gains with special purpose memories and data paths.Third,instruction-set compatibility is not an issue for a chip which execute a small set of specialized routines,so specialized instruction sets which save memory and/or increase performance are attractive. A comman argument against the use of ASIPs is that amny recent application which required an ASIP solution could be replaced by the latest general purpose processor. Unfortunately, the application themselves do not stand still, and by the time MPEG-1 audio decoding can be performed by a standard processor, the market demand is for low-cost DSPs which perform MPEG-2,Dolby AC-3, and MPEG-4.

Following reason shows reasons for the popularity of a custom ASIP soution in the area of wireless and telecommunications which can be summarized as follows

Performance: Stringent constraints on speed, esprcially for wireless applications mean that complex network of amny commercial DSPs is needed for the required thrughput. A custom solution reduces the number of components and the complexity od the communication between components.

Power: For handheld application, low power is essential.Architecture containing extra features which are not used imply unnecessary multiplexing of hardware and consumption of power.

Cost: Many wireless and telecommunication application cannot afford the the luxury of a many chip solution. Especially in high volume products, it is more cost effective to have a custom chip set which is few in number.

3.2.6 Examples of Emerging Architectures

Commercial programmable DSPs These architecture are typically characterized by a Harvard architecture with separate program and data memories,a fixed point core,and fixed I/O peripherals. The instruction-set is highly encoded and tuned to multiply-accumulation operation used for convolution algorithms comaman in DSP.

VLIW processors This type of architecture is charactrized by a network of horizontally programmable execution units

ASIPsAs described above,ASIPs are processors specialized to a specific domain. An ASIP can be optimized for the performance, speed, and power characteristic of the application and can be reused for variations of the application programming

3.3 Closeness Matrics

We already defined partitioning process to group the objects(according to granularity level of the partitioner)and then map these group to system component.To find the optimal solution ,Partitioner has to do estimation to chose the best solution the among the the possible solution.So for estimation it should have some matrics and objective function or cost function.Closeness metrics plays important role in the estimation and to guide the partitioning process.Closeness metrics defines how close the two objects are So that we can pair the objrcts accordingly and form the group optimally and then assign to system components. We can divide the closeness metrics in the three folowwing category.

3.3.1 Interconnect

For interconnect,Connectivity metric is defined as follows.This metric measures the estimated number of wires shared two sets of functional units if they were separated among two different parts.Grouping objects that would share wires should result in less interconnection between parts.This metrics requires finding the data objects(e.g. parameters and flobal variables) accessed by both sets.In some cases ,we must estimates the wires by encoding these data objects into bits.Then the number of wires for a scalar variable in the number of address bits plus the number of word width.In other cases the width is already known.This cases arises when channel between objects has been assigned to a particular bus, where the bus is already has a specific number od the wires.

In order to combine this metric with other metric, It should be normalized to a number between 0 and 1.To do so we can divide the total number of wires accessed by either(not necessarily both) sets of objects.Thus if objects only transfer data between themselves then closeness metric will be 1.On the other hand if either of these two objects can transfer data to another object,then closeness will be less than 1.If they transfer no data between themselves, the closeness will be 0.

3.3.2 Performance Metrics

There are two ways inwhich interaction among objects can affect performance. First, data may need to be communicated from one object to another - grouping such objects onto the same part should reduce lengthy inter-part communication times and hence improve performance. Second two objects may be able execute concurrently-grouping such objects onto the same part may force sequential execution and hence hurt the performance, and should thus be avoided when possible.

Communication metric is defined as the number of bits transfered between two sets of objects. This metric differs from the connectivity metric in that it meausres the amount od data transfered between sets of objects, rather than the number of wires used to transfer that data.For example, if two objects communicate 16 bits of data 10 times over an 8 bit bus, t hen the communication metric would measure 10 X $16 = 160$ bits, whereas the connectivity metric would measure only 8 wires of the bus. The frequency of data transfer can be determined through profiling. We compute this metric by summing the bits transferred between the two sets of objects. We normalize by dividing by the total number of bits transferred between either set and any other object.It has been noted, however, that some communication do not affect the performance constraints. For example, an object A may transfer many parameters to another object, but if A is not constraints, and neither are any of A's calling objects, then the transfer is not particularly important. Thus, Constraints communication metric , which differs from the previous metric in that it evaluates to 0 if communication does not does not affect the performance of any constrainted object. Such a metric might prove especially useful for hardware/software partitioning since constrained objects are more likely to be assigned to hardware, with other objects being assigned to hardware.

Both of the communication metrics require profilling to obtain data transfer frequencies. At times, though, profilling information is not available. So there is an indirect performance metric called **comman accessors**. When two sets of objects are accessed by many of the same objects, grouping those sets may improve the performance. This metric considering an object to be its own accessor, to encourage grouping an object with the objects that access it. To compute this metric, we count the number of objects that access both the sets. We normalize by dividing by the total number of accessors of either set.

Finally,**sequential execution** is defined as the number of pairs of objects that can execute sequentially. where a pair consists of one object from each set. It can be normalized by dividing by the number of possible pairs.

3.4 Balancing the Size of groups

There are two ways in which grouping objects can affect size.First,objects grouped onto same custom part can often share hardware like multiplier, thus rerducing the overall hardware size. Second partitioning objects among some number of identical parts usually requires that we maintain a balance among required size of each group of objects.We have defined a **shared hardware** metric to measure the hardware shared between the two sets of objects B1 and B2. The shared hardware is computed as: $size(B1) + size(B2) - size(B1 + B2)$.Normalize it by dividing by the minimum of size(B1) and size(B2), since the largest sharab;e amount occurs when one objects can be implemented completely using other's object's hardware.

All the above metric group all the object in one group or partition may not of equal sizes.So for this ,we need a metric to measure the how balanced the partition is?One way to encourage balanced sizes is to favor merging smaller objects over the larger one. to prevent any one froup from getting large.**Balanced metric** is defined as the size of an implementation of both sets of objects. We normalize by dividing by the size of all objects in specification. Most of the above metrics make sense whether we are partitioning among hardware or software parts. The exception is shared hardware metric, for which there is no software equivalent metric.

3.5 Granularity and Estimation

An important presupposition for HW/SW partitioning are sophisticated estimation algorithms at a high level of abstraction that obtain high quality results. Therefore the granularities of estimation and partitioning have tobe adapted adequately.we will discuss the effects that arise when the granularities of partitioning and estimation are not adapted in a necessary way.

The well known design steps of a generic cosynthesis flow are:

High-level specification (C, C++,)

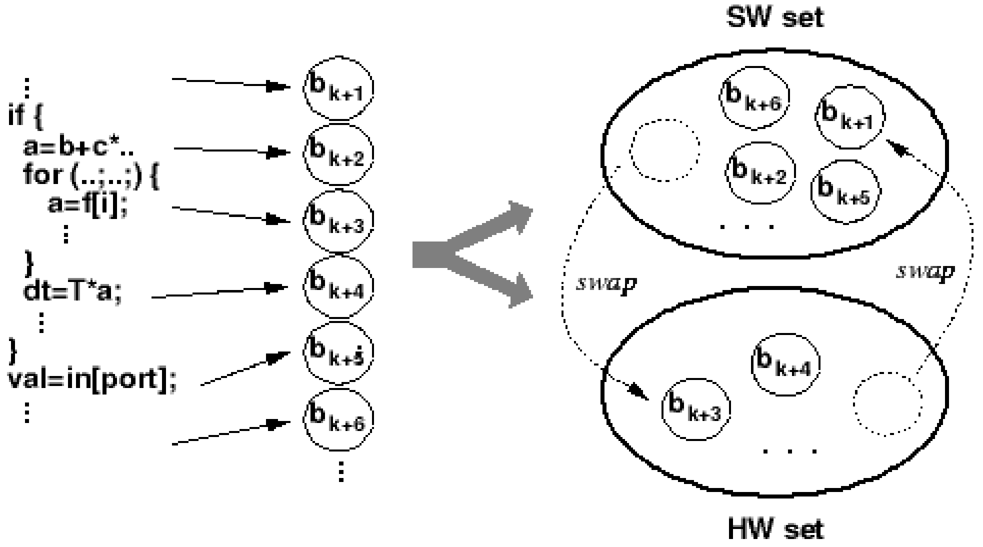
Estimation (HW run-time, SW run-time, chip area,)

HW/SW partitioning

Synthesis (HW, SW, interface)

Steps II. and III. are strongly correlated, so only if estimation values are close enough to the real values (obtained when really implemented) partitioning makes sense.The approaches used in the estimation phase - which is that relevant that it determines the quality (in terms of monetary costs) and the feasibility (given the structure of a target architecture) of a mixed HW/SW solution - cover a wide diversity of algorithms: it spans from simple profiling to so phisticated approaches that also take into account the micro architectural features (pipeline interlocks, register sets,)of a given processor core that is provided to execute, for example, the SW part. The same holds for the hardware part: the question is, how well the HW runtime estimation is suited to that algorithm used when the HW part is really synthesized and what will happen if there is a great deviation between estimation and real data. The same questions could similarly be applied to the communication mechanism between the hardware and software part and the implied costs of an application specific hardware (chip area).

In any case, the answer is that it is nearly impossible or at least economically of no interest to perform an auto matic HW/SW partitioning unless the underlying analyzing/estimation algorithms are of an appropriate accuracy. There are a lot of approaches to HW/SW cosynthesis that differ mainly in two points: the partitioning algorithm it self (automatic by means of a combinatorial optimization algorithm or manually driven) and the accuracy of the an alyzing/estimation algorithms.



a) b)

Figure 1: Defining a granularity from the behavioral description a) for the purpose of HW/SW partitioning b)

Estimating in HW/SW cosynthesis does not only mean estimating from behavioral specification but also estimating at a specific level of granularity. Thereby granularity is defined as follows: c Given the behavioral description of an algorithm the granularity determines how to cut this algorithm into a set B of pieces $b_0, b_1, ...b_n$ for the purpose of distributing these pieces among the hardware part and the software part of a mixed hardware/software system Examples for obvious granularities, directly derived from the structure of the behavioral descriptions, are: operator-level, statement-level, base-block-level, control-block-level and task/process-level. Note that the choice of the granularity does not necessarily mean that the largest code-segment to be implemented either in hardware or in software is limited to the extension of the according granularity3. But it does mean that the smallest code-segment can not be smaller than the smallest element $b_i \in B$. So the granularity determines a lower bound of a code-segment. try to reduce the number of blocks . This is equivalent So one can proceed at any level of granularity.But there are pros and cons of every level. Partitioning on even lower level like at the level of assembly language instructions or operation is not useful because it would be more processor dependent.In our discussion coarse grain partitioning means task level or functional level partitioning and fine grain partitioning means basic block level and statement level partitioning Partitioning at the task or subtask level is typical for mannual design.Some times, partitioning at this level is almost obvious.An example is a sigal -processing task consisting of high speed filtering of the input data exceeding a processors performance, followed by a more complex algorithm with lower performance requirements.

Finer grain partitioning, using coprocessors and second cores, becomes more and more important as processor performance rises and system software increases.At finer granularity,however partitioning is less obvious and moew difficult because its side effect has a high impact.The most important side effects are:-

Communication time overhead Additional I/O operations of the processor core require additional computation time. Load/store architecture typical of RISC controllers even require an extra instruction for each read and write operation.In an extreme case, the overall timing may be worst than before partitioning.

Communication area overhead Besides the obvious writing overhead ,communication can require buffers or memories.Buffer or memory size estimation is not always a simple problem

Interlocks If variables are allocated to an external hardware register,they might not (yet) be available by the time the processor software process them.This leads to waiting time in the software.

Compiler effects When a program is fragmented by the extractions of the statements or the basic blocks, the efficiencyof compiler optimization will change and concurrent unit utilization(superscalar architecture) will be different.These effects are hard to predict. In addition to the large design space including processor selection, peripheral components and coprocessor definition, and synthesis, which we pointed out earlier, these side effects make it even harder for a system design to partition at the level of finer granularity.There are few exception such a floating point or graphics coprocessor.

Nevertheless, fine grain partitioning offers a high potential for system optimization.The exploitation of fine grain partitioning is one opprtunity provided by hardware/software cosynthesis. On the other hand by choosing a coarse-grain granularity,question will arise during the partitioning that whether to move the whole task/process to hardware or to software - independent from the extension of the task (i.e. lines of code). But note that the area of application are small embedded systems which - in most cases - are very cost sensitive: even a few more gate equivalents for the part of the application specific hardware lead to an increase in price that makes this product uninteresting in the market of e.g. consumer products (audio, video,) where competition becomes harder and harder. So, if the designer describes an embedded system only by aspects like modularity, portability etc. - i.e. programming style - this might not be a good choice as a basis for the partitioning process. Therefore it is very probable that a coarse-grain granularity hides inherent optimization potential.Or, on the other side, a fine-grain granularity will keep all the potential open.

3.6 A Comparison of Functional and Structural Partitioning

In functional partitioning, we first partition a functional specification into smaller sub-specifications and then synthesize structure for each, in contrast of current to current approach of first synthesizing structure for the current approach of first synthesizing structure for the entire specification and then partitioning that structure .One advantage is that of greatly improved partitioning among a given set of packages with size and IO constraints, such as FPGAs or ASIC blocks, resulting in better performance and far fever required packages. A second advantage is that of greatly reduced synthesis runtimes.

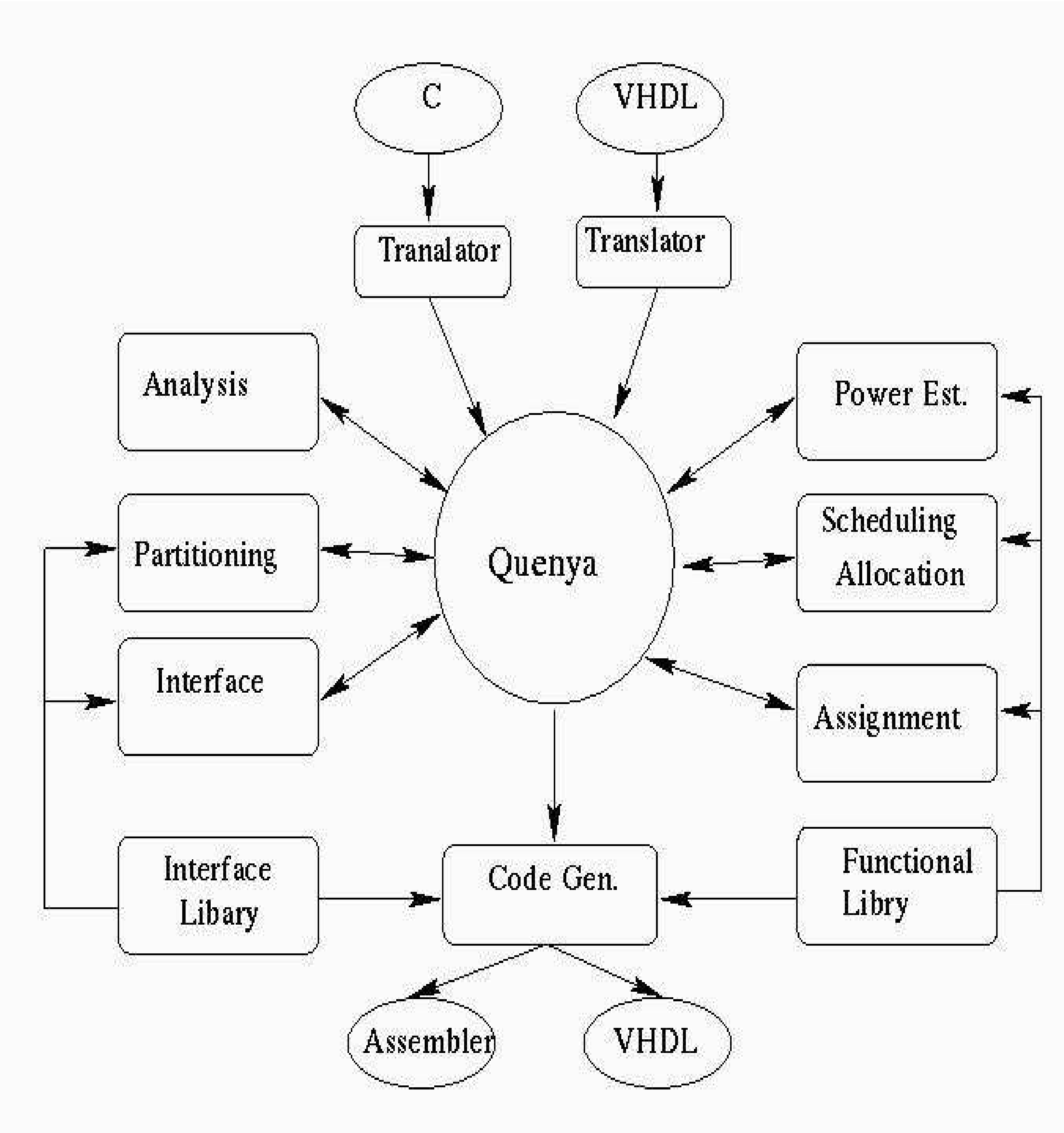
Intuitively, functional portioning excels by assigning each function to one part, rather than spreading a function over several parts. Such isolation: (1) reduces IO, (2) prevents the critical path from crossing parts, thus reducing the clock period, and (3) often yields simpler hardware, thus further reducing the clock period. More importantly, we have complete control over IO at the functional level, and can easily tradeoff performance and IO. In particular, data transfers between parts can be easily time-multiplexed over one bus by inserting an addressed-protocol behavior (or even an arbiter behavior), and transfers can be partially or fully serialized. Another advantage is that each parts behavior is readable and late changes are often isolated to one part, leading some designers to call this approach portioning for debug. These advantages come with drawbacks. First, functional partitioning must be guided be estimates of size, IO and performance for possibly thousands of examined partitions, but obtaining good size and performance estimates quickly can be hard. In contrast, in structural partitioning, we can easily estimate size and delay quickly and accurately, by summing object sizes and counting critical path cuts. However, sophisticated estimation techniques can help alleviate this drawback . Second, a functionally partitioned system often (though not always, as we shall see) uses mire gates, since hardware objects arent shared by functions on different parts. However, since partitioning is often IO dominated, this increase is often insignificant. Another problem solvable with functional partitioning is reducing synthesis tool runtime. In particular, one large process in a specification can result in excessive tool CPU time and memory use and can lead to an inefficient design. By functionally partitioning the specification into smaller ones, and imputing each to the synthesis tool independently, runtimes can be reduced by an order of magnitude.

IO and size Functional partitioning led to much better satisfaction of IO and size constraints.

Performance Functional partitioning led to better performance than structural partitioning. To analyze performance, we must look at two factors: (1) the number of clock cycles n to execute the specification (say on the average), and (2) the clock period r. The performance is then computed as n x r. In functional partitioning, we introduce more clock cycles stays the same, but we must extend the clock period to account for each intercomponent delay.

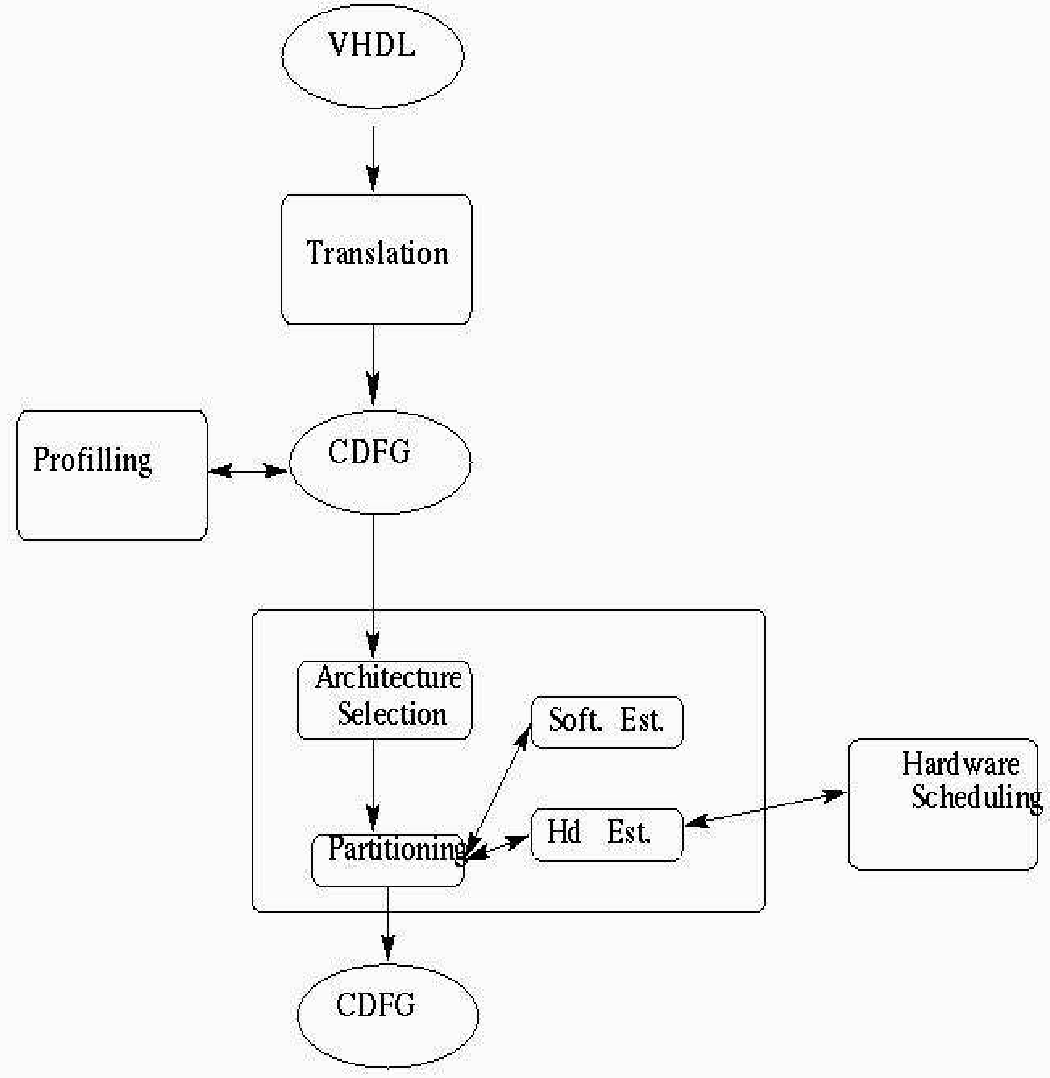
Design size One concern is that functional partitioning would lead to much larger designs caused by the inability to share functional unit across parts and to extra hardware for communication between parts. There is usually only a slight increase in size, roughly 10-20% (which is quite neglegible considering that structural partitioning cannot utilize even more gates since it is IO dominated.

4 Partition tool


C
VHDL
Tranalator
Translator
Analysis
Power Est.
Partitioning
Quenya
Scheduling
Allocation
Interface
Assignment
Interface
Libary
Code Gen.
Functional
Libry
Assembler
VHDL

4.1 LYCOS The Lyngby Cosynthesis System

One of the main ideas of the LYCOS system is to have a system which supports as easy inclusion of new design tools and algorithms, as well as new design methods, e.g. the sequence in which tools have to be applied in order to obtain a solution. Thus, a key issue for LYCOS has been to be able to test new ideas and algorithms not as separate entities but as part of a complete design trajectory. LYCOS is built as a suite of tools centered around an implementation independent model of computation, called Quenya, which is based on communicating control/data flow graphs. The information which is communicated between different tools in LYCOS basically consists of two parts: The central functionality of a design (i.e. the behavior derived from the input specification) and design hints obtained during synthesis (e.g. portioning or scheduling information, profiling, and performance). In order to achieve the necessary flexibility to support the requirements of different kinds of synthesis tools, while preserving the semantic integrity of the description, this dichotomy has been reflected in the design of Quenya. The functional behavior of a design is represented through a hierarchical network with strictly defined semantics. The network consists of a number of functional units communicating by asynchronous protocols, with the communication channels represented as shared variables. Thus, Quenya directly allows design partitioning to be represented, and supports representation of the systems environment in a uniform manner. The behavior of an individual functional unit is represented by a control/data flow graph (CDFG)



Synthesis trajectory for a hardware/Software partitioning in LYCOS

A PARTITIONING SESSION IN LYCOS Translating the Input Specification. First we have to prepare the input specification by translating the VHDL code into the Quenya CDFG format. This is done by the tool vhdl12quenya. Profiling the Input Specification. From a partitioning point of view it is important to have some information about time critical parts of the application, since these may have to be moved into hardware in order to meet timing constraints. This information may be obtained by profiling which collects execution statistics given a set of input data for the application. Profiling is performed on the CDFG and is therefore independent of the final implementation. The profile is obtained by translating the CDFG to C++ (using the tool called quenya2c). Loading the CDFG into the Partitioning Tool. We many change the partitioning granularity, i.e. we may choose whether to have a large number of rather small BSBs or to have a smaller number to larger BSBs. Selecting a Processor. Having loaded the CDFG, we have to decide upon the target architecture. First, we select the processor on which to run the software part. This is selected from the file menu by loading a processor technology file from software library. The technology file contains a set of generic assembler instructions, each of which corresponds to a particular operation using a particular addressing mode. Each generic instruction is associated with its execution time and size in bytes. Currently technology files for Motorola 68000, Motorola 68020, Intel 8086, and Intel 80286 processors are provided which enables us to perform design space exploration experiments where processors price/performance tradeoffs can be made. Selecting a Hardware Technology. The hardware library contains for each technology a set of functional components. For each component the library contains information on delay, latency, area, provided operations, storage capabilities, etc., as well as the number and types of ports. Estimating Hardware/Software Execution Times. When the target architecture is fully characterized, the all-software and all-hardware execution times are estimated. Software execution time is estimated within using the CDFG and the selected processor technology file. The estimate is obtained by first linearizing the CDFG and then allocating the registers of the processor. After this, each operation in the CDFG can be mapped to a generic instruction(corresponding to the instructions in the processor technology file) using the appropriate addressing mode. Finally, the execution time is calculated based on the clock cycle times in the processor technology file. Hardware execution time is estimated by the tool wx.albatresa which is a dynamic scheduling tool. Partitioning. At this point we are ready to perform the partitioning wx.palace allows to choose between different partitioning models and different partitioning algorithms.

4.2 COSYMA COSYnthesis for eMbedded micro Architectures

COSYMA (COSYnthesis for eMbedded micro Architectures) is a platform for exploration of the co-synthesis process.The COSYMA target architecture consists of a standard RISC processor core (we provide a SPARC architecture model with 33 MHz clock and floating point coprocessor with COSYMA), a fast RAM for program and data with single clock cycle access time,and an automatically generated application specific coprocessor. Peripheral units must be inserted by the designer. Processor and coprocessor communicate via shared memory in mutual exclusion.The input description may consist of several communicating processes with timing requirements. The description consists of a function description in C, which is a minimum extension of the C programming language by a process statement allowing parallel processes. Process communication uses predefined C functions accessing abstract communication channels that are later mapped to physical channels (or are removed by optimization). It is similar to the approach in except that both blocking and non blocking communication are supported . Peripheral devices must be modeled as C processes for simulation. C processes are also used for stimuli generation. Both must be removed for process scheduling and partitioning.

The function description in C is strictly separated from constraints and implementation directives to minimize C language extensions and to improve language portability. A constraint and user directives file contains time constraints which refer to labels in the C processes, as well as channel mapping directives, partitioning directives, and component selection constraints and global user directives, again, are separated from tool specific user control for a compact input description and to improve tool independence.COSYMA design flow and user interaction

The input description is translated to a syntax graph using a customized version of the Stanford SUIF compiler. This syntax graph is extended to an Extended Syntax Graph (ESG) by annotating the local data flow and the global data flow, as far as it can be analyzed. As an advantage, the program structure is conserved and the data annotations do not need to be complete, such as in a data flow graph (DFG) or in a control/data flow graph (CDFG). Instead, data flow is analyzed as good as possible while for all other data dependences default worst case assumptions can easily be derived from the syntax graph Next, the Cx processes are simulated on an RT-level model of the target processor to obtain profiling and software timing information. Since simulation uses abstract channel functions for process communication, the result is an approximation of the final system behavior. In newer versions of COSYMA, profiling is replaced by a symbolic analysis approach, which is outlined later. Software timing data for each potential target processor is derived with simulation or symbolic analysis. This is a time consuming but necessary approach to obtain correct timing data for the later steps.

Single process systems directly proceed to hardware/software partitioning. For multiple process systems, COSYMA uses different approaches to scheduling. The first one uses scalable performance scheduling (SPS) which finds a feasible and minimal serialization of processes which is then treated as a single process. In this case, process scheduling precedes hardware/software partitioning. This approach can only handle rate constraints on processes, while input/output timing constraints can only be regarded in a few special cases. A second, newer approach combines partitioning and scheduling and departs from mutual exclusive execution .

Hardware/software partitioning is the next step. Input to partitioning are the ESG with profiling (or control flow analysis) information, the CDR-file and synthesis directives. These synthesis directives include the number and type of functional units provided for the coprocessor implementation and they are needed to estimate the potential hardware performance. This parameter is controlled by user interaction.

Partitioning works at the basic block level. Since communication between basic blocks of a process is implicit, partitioning requires communication analysis and, thereafter, communication synthesis. This is a major difference to all approaches which partition on the level of processes using explicit communication described by the user. Communication is inserted when the internal ESG representation is translated back to C for software synthesis and to a hardware description language for high-level synthesis.

For high-level synthesis , the Braunschweig Synthesis System (BSS) is used. BSS creates a diagram showing the scheduling steps and function unit and memory utilization which allows to identify system bottlenecks. An RT-level synthesis tool, the Synopsys Design Compiler, generates the final netlist . For software synthesis, a standard C compiler is used. There is also a tool which allows to co-simulate the object code running on the target processor and an RT-level hardware description provided by BSS. This tool is mainly used for COSYMA software test.

Run time analysis is the last step which uses software simulation and hardware scheduling results in formal analysis for timing constraint validation, assuming worst-case communication channel timing. This saves an extra co-simulation step and has turned out be sufficiently accurate. The netlist is finally completed by VHDL models for the peripheral modules which were excluded from the synthesis process. The intermediate COSYMA results can be examined in many forms:

Process scheduling

Gantt diagram and process graph (HW/SW partitioning)

Marked list of hardware and software blocks (communication synthesis)

Communication statements in the created C code (high-level synthesis)

Scheduling diagram showing operations and memory access per control step

Hardware/Software partitioning

Partitioning in COSYMA starts with an all software solution and tries to extract hardware components iteratively until all timing constraints are met. The partitioning goals are in order of decreasing importance are

meet real-time constraints

minimize hardware costs

minimize the CAD system response time and, thus, allow the user to investigate the influence of system modifications on systems costs and performance.

While the first goal is a hard constraint (because it is a necessary condition), the other ones are (soft) optimization goals. Hardware/software partitioning can be considered as a scheduling problem which is known to be NP-hard. Given a user specified maximum amount of hardware components (i.e. ALU, multiplier, ...), our task is to find a minimum set of code segments that lead to a sufficient speedup when implemented in hardware. We selected Simulated Annealing, a stochastic optimization algorithm.

The elements in the partitioning process are basic blocks which seems to be a manageable compromise between statement level (fine grain) on the one side and function or process level partitioning (coarse grain) on the other side . The total (estimated) costs of a single basic block b - assumed that it is moved from software to hardware - amounts to:

$\delta c(b) = w.\delta t(b) = w.[(t_{HW}(b) - t_{SW}(b) + t_{com}(Z) - t_{com}(ZUb)).It(b)]$

where $\delta c(b)$is the estimated cost increment, w is a weight factor to control Simulated Annealing , $\delta t(b)$is the estimated execution time increment, $t_{HW}(b)$ is the estimated coprocessor execution time of b, $t_{SW}(b)$ is the estimated execution time of b on the processor and $t_{com}(Z)$ is the estimated processor-coprocessor communication time, given the current set Z of basic blocks on the coprocessor. It(b) is the number of iterations as determined by profiling/tracing on b. All estimations are required previous to partitioning because Simulated Annealing needs a fast cost function computation.

The execution time of a basic block on the processor and the coprocessor depends on the global optimization of the compiler and the synthesis tool. Each move of a block from software to hardware and vice versa changes the optimization potential. Since Simulated Annealing typically tries several hundreds of thousands of hardware/software configurations, off-line estimation techniques with fast incremental update after each move are necessary.

$t_{SW}(b)$ is estimated with a local source code timing estimation based on simulation data. Estimation inaccuracy results from data dependent instruction execution times (SPARC: e.g. mult, div), optimization and register allocation in the compiler. Alternatively, trace data from processor simulation could be used.

$t_{HW}(b)$ is estimated with a list scheduler (fast) or a path-based scheduler (more accurate) on b using the execution time (number of clock cycles) for each operator in high-level synthesis, given the user specified number of functional units used.

$t_{com}(ZUb)$is estimated by data flow analysis. For shared memory, communication costs are proportional to the number of variables to be communicated.