Technical Report

A Framework for Automation of System-Level Design Space Exploration

Manan Kathuria and Andreas Gerstlauer

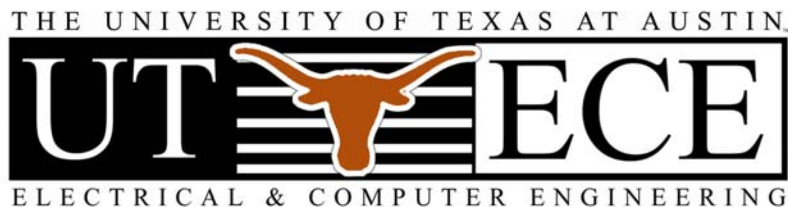
UT-CERC-12-05

June 25, 2012

Computer Engineering Research Center
Department of Electrical & Computer Engineering
The University of Texas at Austin

201 E. 24th St., Stop C8800
Austin, Texas 78712-1234

Telephone: 512-471-8000
Fax: 512-471-8967
http://www.cerc.utexas.edu



A Framework for Automation of System-level Design Space Exploration

Manan Kathuria and Andreas Gerstlauer

Computer Engineering Research Center

The University of Texas at Austin, 2012

Abstract

Design Space Exploration is the task of identifying optimal implementation architectures for an application. On the front-end, it involves multi-objective optimization through a large space of options, and lends itself to a multitude of algorithmic approaches. On the back-end, it relies extensively on common capabilities such as model refinement, simulation and assessment of parameters like performance and cost. These characteristics present an opportunity to create an infrastructure that enables multiple approaches to be deployed using generic back-end services. In this work, we describe such a framework, developed using the System-on-Chip Environment, and we demonstrate the benefits and feasibility of deploying a variety of design space exploration approaches built on top of this basic infrastructure.

Table of Contents

List of Tables iii

List of Figures iv

Chapter 1: Introduction 1

1.1 Background 3

1.2 Motivation 5

Chapter 2: Related Work 7

2.1 System-on-Chip Environment (SCE) 7

2.2 Existing Design Space Exploration Tools 7

2.3 Limitations and Opportunities 9

Chapter 3: Design Space Exploration Framework 10

3.2 SCE_GDS: Design Space Generation Service 14

3.2.1 Design Space Encoding 14

3.2.2 SCE_GDS Description 15

3.3 SCE_REFINE: Model Refinement Service 22

3.3.1 Design Modeling and Evaluation 22

3.3.1 SCE_REFINE Description 23

3.4 SCE_SIMULATE: Simulation and Assessment Service 27

Chapter 4: Demonstration and Results 28

4.1 Simple Brute Force Exploration Flow 28

4.2 Two-step Brute Force Exploration Flow 28

4.3 Results 31

Chapter 5: Conclusion 34

References 36

List of Tables

Table 1.1: ESL design automation..3

Table 4.1: MiBench DSE time...33

List of Figures

Figure 1.1: Generic Embedded System Architecture 1

Figure 1.3: Example Pareto-optimal front 5

Figure 2.1: SCE flow 8

Figure 3.1: *sce_explore* environment...11

Figure 3.1: *sce_gds* flow 16

Figure 3.3: *sce_refine* flow 25

Figure 4.1: Brute-force DSE flow..29

Figure 4.2: Hierarchical, two step DSE flow...30

Figure 4.3: MiBench design...32

Figure 4.4: Pareto-optimal front for MiBench DSE 33

Chapter 1: Introduction

Over a short history, embedded systems have grown in their abilities from simple single-purpose computers to complex systems-of-systems. A showcase example of this trend is the progression from devices such as cameras, GPS receivers and MP3 players to an integrated all in one device – the Smartphone. The generic architecture for an embedded system, as shown in Figure 1.1, can be described as an application executing on an embedded computing platform.

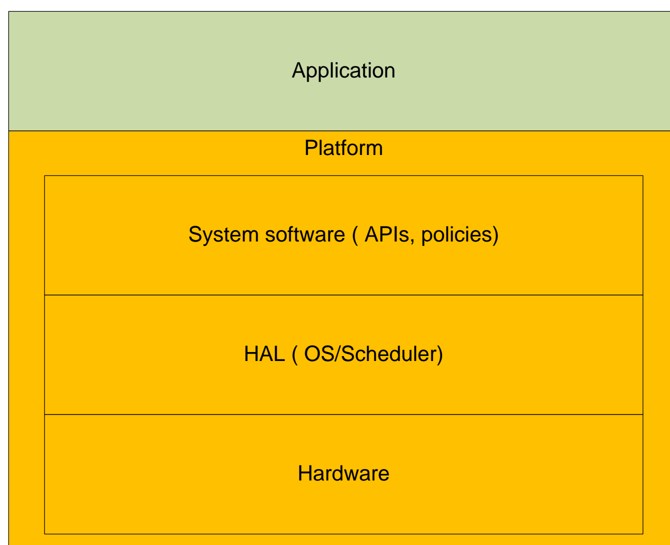


Figure 1.1: Generic Embedded System Architecture

System requirements have multiplied driven by the key factors of higher performance, lower power and tighter integration. This has led to an explosion in complexity at a number of levels:

- **Hardware:** single core microcontroller platforms are being replaced by dual core platforms with embedded DSP, hardware accelerators and FPGAs
- **Software:** real-time operating systems (RTOS) are replacing single-threaded and simple scheduler based systems

- **System:** system-level policies are becoming necessary to manage power and security, causing complex interactions between seemingly disconnected applications on an integrated platform
- **Network:** More and more embedded applications are getting connected to each other and the internet

These trends show embedded systems moving to more closely resemble general-purpose computing systems (for example, laptop computing) in complexity. It is important to note that system design in general-purpose computing is largely a meet in the middle approach. Platform design is application independent, and applications are designed within the constraints of the available platform. System-level requirements (which span both application and platform) such as power consumption and response times are relaxed. As a result, system design complexity is isolated within the application domain.

On the other hand, embedded systems are rooted in their application-specific nature. Requirements such as application response times, low cost and long battery life are an order of magnitude stricter, failing which systems do not serve their intended purpose. Therefore, system design includes both application and platform design. Traditionally, in a low complexity regime, static analysis sufficed to predict system-level behavior and guarantee requirements. However, the highly complex nature of current and future systems as described above makes static analysis infeasible. Executable model-based design techniques are indispensable to be able to identify complex interactions and design systems to deal with them. This is referred to as the domain of Electronic System Level (ESL) design.

ESL raises the abstraction in the design process to the level of the system. The key enablers for ESL are (i) Executable models of system components to serve as collateral, and (ii) Design automation tools to drive designer productivity. Some of the broad categories of design

automation capabilities are shown in Table 1.1. These capabilities create an ecosystem which is leveraged by design automation tools to enable ESL techniques to be used in real-world designs. Already, there has been tremendous applied work in the field in academia as well as industry. System-level design languages (SLDLs) such as SystemC [1] have pervaded the industry and enabled a number of system-level design tools, especially in performance estimation and verification.

ESL Enabler	**Goal**
Modeling	Create fast yet accurate models that only represent effects necessary and sufficient for a system-level evaluation Automatically generate models from structural descriptions
Simulation	Simulate complex models of complete systems faster
Synthesis	Automatically synthesize systems from abstract specifications down to hardware (RTL) and software (C code)
Design Space Exploration	Pick the right system architecture to optimize design quality metrics while meeting design goals

Table 1.1: ESL design automation

1.1 BACKGROUND

Embedded systems designers need to find implementation architectures that satisfy or optimize technical requirements such as throughput, latency and power consumption, while also meeting cost and time to market goals. Figure 1.2 shows a typical embedded systems platform today. While system architectures have traditionally been derived from static analysis and an architect's domain expertise, these approaches are proving insufficient when faced with a large set of increasingly complex system architecture components. Hence, there has been interest in an automated approach to system architecture definition, referred to as Design Space Exploration (DSE).

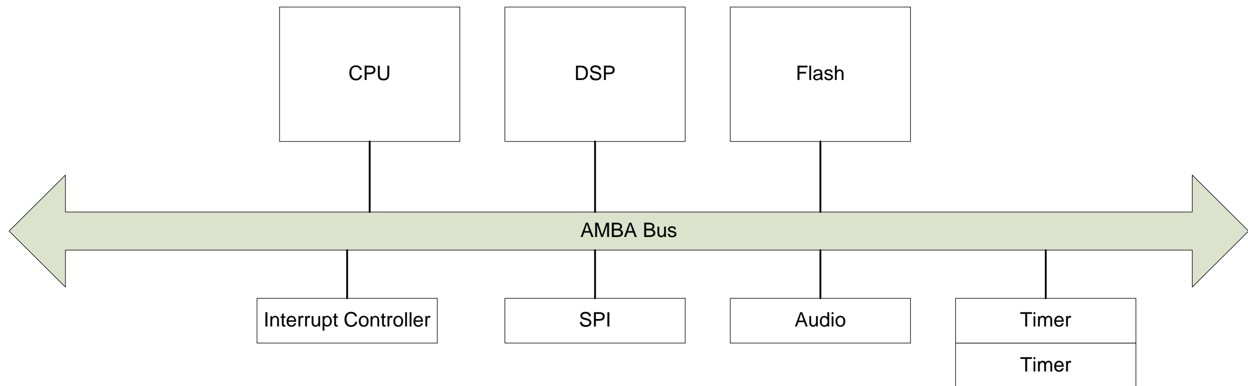


Figure 1.2: Typical embedded system platform

System architects have potentially a large pool of system architectures at their disposal, comprising of various combinations of system hardware, software and policy components. This is referred to as the *Design Space.* The key requirement of a DSE system is to take a set of system constraints, and be able to identify a set of optimal system architectures from the design space through multi-objective optimization. The output of a DSE system is the *Pareto-optimal front* of possible candidate designs, as shown in Figure 1.3. The Pareto-optimal front comprises of all the *Pareto-optimal points,* i.e. every point that is not worse than any other in all the measured system metrics. For example, design point 'A' is equal on objective 1 with point 'B', but exceeds it on objective 2. Design point 'C' is better than 'B' on objective 1, but worse on objective 2. As a result, 'B' and 'C' are both Pareto-optimal points, while 'A' is not. As can be seen, there can and will be many potential architectures that are optimal, and choosing between them requires an understanding of system design goals and constraints. For example, only optimizing objective 1 as much as possible would yield 'D' as the optimal design point, while neglecting many other equally Pareto-optimal points on the tradeoff front.

Also worth noting on Figure 1.3 is the line depicting a system constraint. In this case, this means that any possible implementation architectures should lie to the left of this line. Therefore, even though ‘E’ is a Pareto-optimal point, it exceeds the system constraint and is unsuitable.

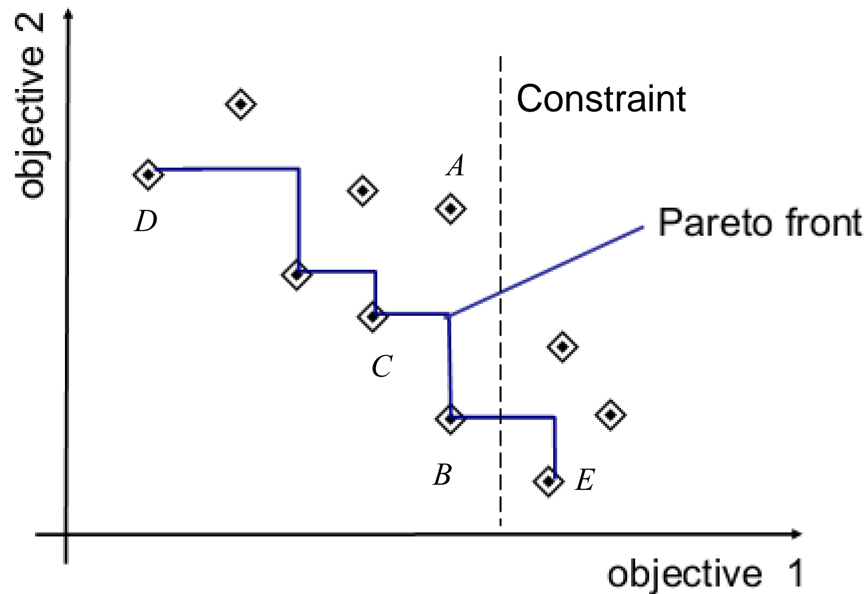


Figure 1.3: Example Pareto-optimal front

1.2 MOTIVATION

There are a number of algorithmic approaches to DSE. These range from simple but non-scalable brute force approaches to complex mechanisms mimicking processes such as genetic evolution and the social behavior of ants. All these approaches share a common set of capabilities:

(i) **Design Space Generation:** The ability to view, represent and encode the complete set of potential and feasible design architectures made up of different combinations of platform components.

(ii) **Model Refinement**: The ability to generate an executable model of the application executing on a given platform architecture.

(iii) **Assessment**: The ability to assess a set of parameters for the application executing on the given architecture.

(iv) **Selection** - The ability to test the fitness of a pool of design points, and to eventually prune the design space from a large pool to a smaller, Pareto-optimal set.

While selection policies vary from one DSE system to another, other capabilities – design space generation, model refinement and assessment – are the same and are only exercised in different ways. While Chapter 3 describes this in more detail, it is important to mention here that there is no "best" selection algorithm, and system performance depends on the design under test. Additionally, all the capabilities described above are active research areas and are constantly being improved. Therefore an ideal DSE approach is one that:

- Supports a number of algorithmic approaches using common lower level infrastructure.
- Supports abstraction of each capability to allow for advancements in each to be brought into the system without affecting other capabilities.

Our work focuses on developing a framework that enables such a modular and extensible approach to design space exploration using the System-on-Chip (SCE) environment [2]. Using this framework, various algorithmic approaches can be rapidly implemented and incorporated under the same umbrella. The remainder of this report is organized as follows: Chapter 2 details the capabilities of SCE as well as some of the existing DSE tools. Chapter 3 provides an overview of *sce_explore*, our DSE environment, and describes each of the services provided by *sce_explore* – *sce_gds*, *sce_refine* and *sce_simulate* - in detail. Then, Chapter 4 describes the realization of two simple DSE systems using the provided services and results achieved on a representative design. Finally, Chapter 5 concludes this work and suggests future directions to further improve on this effort.

Chapter 2: Related Work

This section provides necessary background on the capabilities of SCE, as well as some existing Design Space Exploration systems.

2.1 System-on-Chip Environment (SCE)

Figure 2.1 depicts the SCE flow [2]. SCE takes an application model written in the SpecC SLDL [6] as input, consisting of actors performing computation, and communicating through abstract channels. SCE takes in design decisions at levels of computation decisions and communication decisions. Incorporating these decisions with platform component models, it then generates refined executable models in SpecC form. At each level (computation-only/computation-and-communication), these models can then be simulated to estimate parameters such as cost and performance. SCE is also a testbed for innovations in system-level modeling and synthesis. Some of these innovations include faster and more accurate simulation methodologies [7] and abstract models of multi-core operating systems (OSs) [8].

2.2 Existing Design Space Exploration Tools

There are a number of DSE engines using different algorithms. System Co-Designer [3], for instance, uses Genetic Algorithms (GA) for "intelligent" random explorations, whereas DESERT [4] relies on an Ordered Binary Decision Diagrams based symbolic search. Sesame [5], as part of the Daedulus framework, has support for an exhaustive simulation-based DSE (for small design space) as well as for heuristic explorations using GAs or other DSE algorithms.

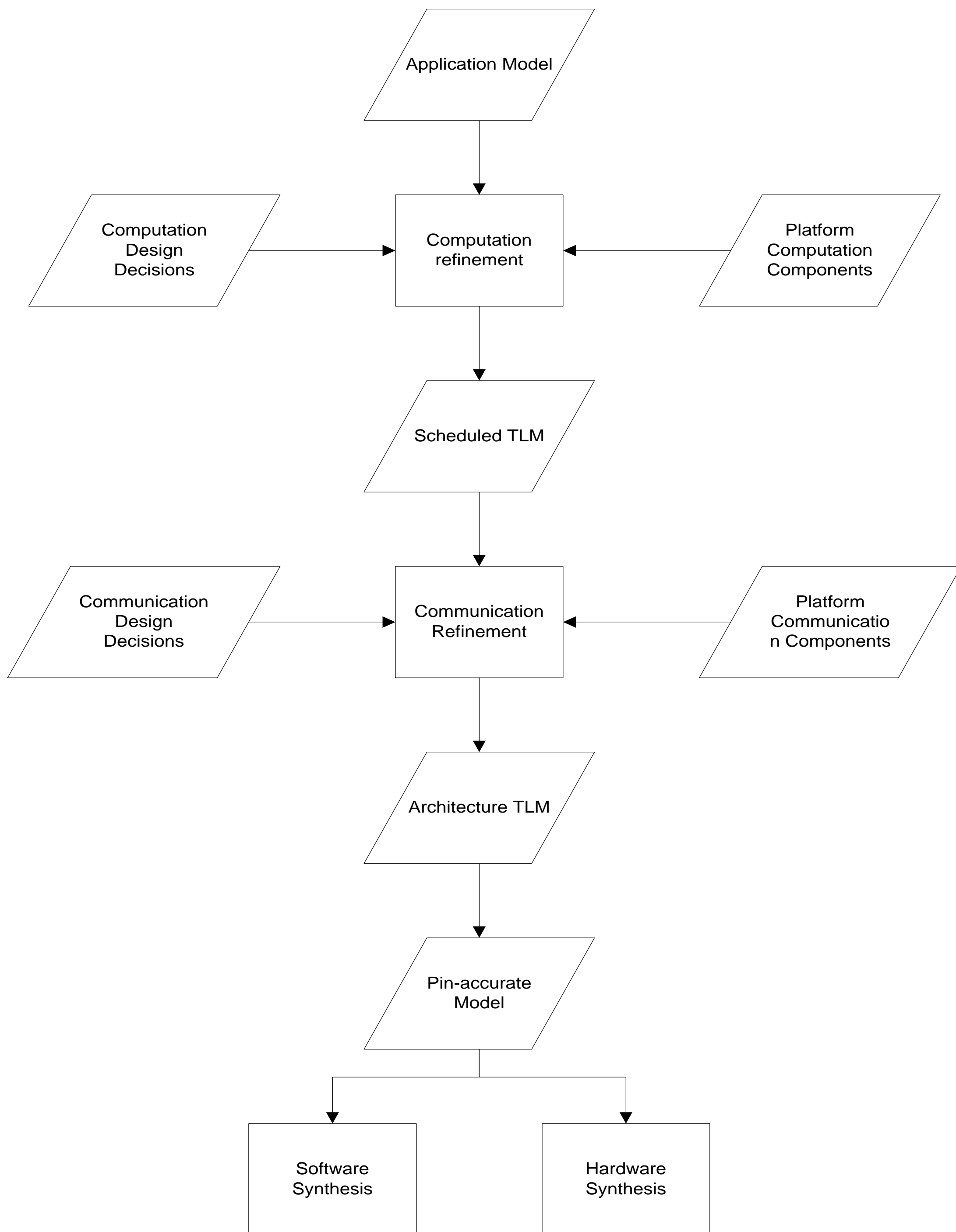


Figure 2.1: SCE flow

2.3 LIMITATIONS AND OPPORTUNITIES

There are some serious limitations to existing tools, which also presents opportunities for improving DSE methodologies. As mentioned earlier, most DSE tools are restricted to a single algorithm. An ideal solution is one that has the ability to put together a number of algorithms, and use the one most suited to the design under exploration. For example, a small design can be explored best by using a simple brute-force approach, while a large design requires a sophisticated heuristics-based system.

All DSE engines also build their own assessment flow. Given that synthesis, simulation and assessment is a research field in itself, these engines display serious deficiencies in trying to master these fields. For example, the Daedulus framework provides good specification and computation decision making and refining capabilities, but lacks in the areas of communication decision making and refinement as well as in support for dynamic scheduling of processes.

Finally, most of the current engines complete the decision making process in one go, without taking feedback from successive refinements of the system model as is possible in the SCE flow. Therefore, as shown in this work, an integration of higher-level algorithmic engines with the inherent refinement, simulation and assessment capabilities of SCE provides an opportunity to bring the best of both worlds together.

Chapter 3: Design Space Exploration Framework

In this section, we present the ***sce_explore*** framework for design space exploration. The key to enabling the effective incorporation of multiple algorithmic approaches within the same environment is to separate the control plane for decision making and selection from the data processing plane which handles tasks such as model refinement. The data plane provides abstract back-end services common to algorithmic approaches, while the control plane provides a canvas to implement a front-end DSE approach utilizing these services.

Figure 3.1 shows the *sce_explore* environment. This environment requires the application specification, platform component information and the model database as inputs. The data plane for *sce_explore* encapsulates three distinct services, namely (i) ***sce_gds*** (ii) ***sce_refine*** and (iii) ***sce_simulate***. The control plane shows a generic DSE flow as described below.

The role of the data plane services and the configuration of the control plane can be best shown by describing the implementation of a generic DSE flow using the *sce_explore* framework. Each of the services is described in detail in subsequent sections of this chapter.

(i) The system starts by calling *sce_gds* to build a representation of all the design space points. This enables the system to gather an understanding of the entire design space, and pick an efficient way to encode various architectures for representing them to the decision-making engine. The output of this step is a pool of individual design instance representations in the form of XML files, as well as a look-up table containing the encoding for each of those instances.

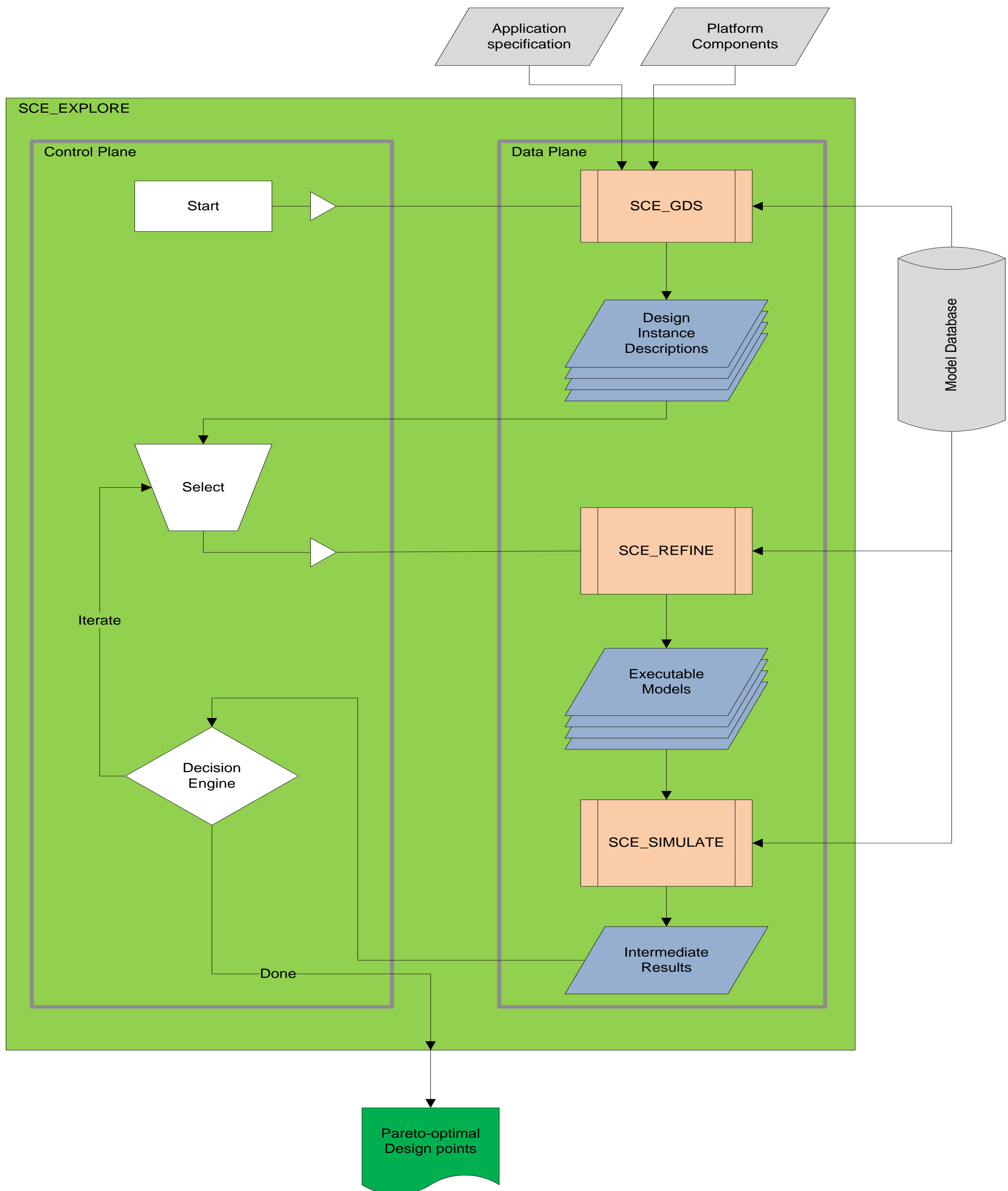


Figure 3.1: *sce_explore* environment

(ii) Next, the system selects a subset of these potential architectures and generates refined executable models capturing platform architecture effects (computation-only effects or both computation and communication effects) using *sce_refine*

(iii) These executable models are then simulated using *sce_simulate* to estimate design parameters such as performance, power etc.

(iv) These results are fed back into the decision engine which now selects a different set of design points

(v) Steps (ii) through (iv) are repeated until the decision engine decides it has completed, and identified a set of Pareto-optimal points

(vi) The system generates an output containing the Pareto-optimal points it has identified.

In this manner, the *sce_explore* environment can be used to implement a number of approaches. For example, in a simple brute force approach, the decision engine would select, simulate and assess all possible design points to discover the Pareto-optimal points. This approach is definitive and useful for small designs, but prohibitive for large complex designs.

For large complex designs, a genetic-algorithm based approach can be incorporated. Genetic algorithms mimic the process of natural evolution [9]. In the first step, the design space is encoded with a binary representation, with a fixed size and position representation for each architectural choice. The search is then initialized with a randomly generated population. This population is taken through refinement and simulation, and the design parameters thus assessed are used to select individual solutions through a fitness-based procedure, in which fitter solutions are more likely to be selected. This pruned population is then used to generate a 'child' population through genetic operators such as crossover and mutation. This child population shares many of the characteristics of its parent population. This process is repeated until the algorithm reaches a termination condition (for example, a fixed number of iterations, or a fixed

run-time). At this time, the system has assessed a number of design points, and can output the optimal points.

The computing tasks within the *sce_explore* framework are highly parallel, and hence we structure them in separate compute processes that can be run in parallel on an underlying managed compute cluster. Importantly, in case a managed compute framework is not available, *sce_explore* builds a simple managed compute framework using available networked machines. It requires a specification of the machines to be used, and the number of cores available per machine. It then schedules jobs on each individual core in a round robin manner, waiting until a job is finished on one core to assign the next job to it. Since this is intended as a basic compute cluster, it does not take into account effects such as processor load in scheduling.

3.2 SCE_GDS: DESIGN SPACE GENERATION SERVICE

Any DSE system has to start first with knowing its potential design space and picking an efficient way to represent it. This section describes the *sce_gds* service built on top of SCE for this purpose.

3.2.1 Design Space Encoding

As part of the *sce_explore* environment, we choose to generate and store design space points offline, i.e. before the decision-making engine begins. The DSE algorithm then refers to each design point using an encoded key into a look-up table. This is motivated by observing certain key factors common to most DSE algorithms that would use the *sce_gds* service:

- The design space is finite and will only change when the design changes. Hence, it can be generated once and the same representation can be used in multiple explorations. Since the number of design space points can be fairly large and time required for generation substantial, this approach does have an initial setup hit but saves time over the exploration process.
- Exploration time is the critical path, and reducing time taken to reach an optimal decision set is a key objective of any DSE approach. Offline decision generation takes that step out of the critical path.
- A DSE algorithm can potentially be non-deterministic (such as genetic algorithms) causing it to hit the same design point in exploration multiple times. Additionally, and especially for genetic algorithms, the system needs to go through a minimum set every time to be able to converge. This will involve re-assessing design points that may have already been assessed in previous explorations. An offline, lookup-table based approach makes caching of previously assessed design points very easy.

- Randomized design point selection and operations such as recombination are characteristic of many DSE algorithms. These operations are made easier by a lookup table based approach, since every design point is represented by an encoding.

We represent design architectures in human-readable XML format, with each design point being a unique XML file, and having an entry in a look-up table file. Two design spaces are generated, considering computation-only architectures and computation-and-communication architectures. This is motivated by the fact that computation-only exploration provides an opportunity to reduce the complexity of exploration, as will be shown in Chapter 4.

3.2.2 SCE_GDS Description

Figure 3.2 shows the flowchart for the design space generation process. Each of the steps is described in the following sub-sections.

Platform Elements Specification

The following set of platform element specifications are required to generate the design space:

1. Software processing elements (CPU)
    - The processor types to be considered in the design space
    - The maximum number of software processors to be considered in the exploration
    - The cost of each processor type
2. Hardware processing elements (HW)
    - The maximum number of hardware processing elements to be considered in the exploration
    - The cost of each hardware processing element
3. Memory elements

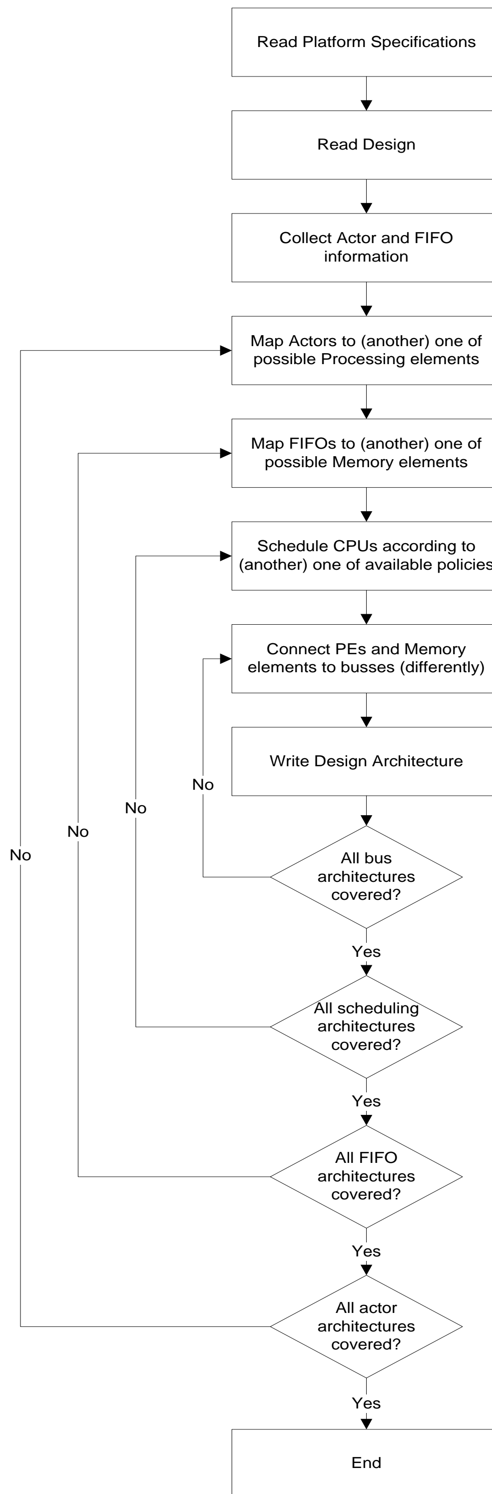


Figure 3.1: *sce_gds* flow

- The memory types to be considered
- The cost of each memory element

4. Interconnect elements
   - The types of busses to be considered
   - The cost of each bus element
5. Transducer elements
   - The type of transducers to be considered
   - The cost of each transducer element

Additionally, another key platform specification is the hierarchical depth of exploration. Any application is hierarchically built from specification actors, which are themselves built from other actors. At any particular depth, there will be a finite number of actors and channels. We set a particular depth limit 'n', and any actors and channels at depth level n+1 and beyond will always be mapped to the same processing elements. This helps restrict the complexity of the exploration.

Read Design

Once the exploration depth has been set, the framework creates a list of all actors, shared variables and communication channels by parsing through the design. This is stored in the internal database.

Mapping Actors and Variables

Actors in the design need to be bound to one of the possible processing elements (PEs), which could be any of the software processors (CPU) available through the platform, or a Hardware processor (HW). Similarly, variables need to either be in a shared global memory, or

be stored in PE internal memories with a synchronization mechanism to keep the individual copies in sync.

Our framework operates by recursively allocating each actor and variable to each of the potential options in separate passes. For example, if global constraints allow 2 ARM processors and 2 hardware units, actor 'A' will be mapped onto ARM1, ARM2, HW1 and HW2 in 4 separate passes. Each of these architectures is a starting point for an actor 'B' to be mapped onto the same set of PEs. Once all actors are mapped, the framework moves on to variable mapping, considering each actor mapping architecture as a starting point for decisions about FIFO mapping. Each of the variables is evaluated to check if the actors sharing it are mapped onto the same or different PEs. In case all actors are mapped to the same PE, the variable will be part of the PE's internal memory (which is part of the PE model and does not require a separate memory element). However, in case the variable is shared between two different PE's, we need to make a mapping choice as described earlier[1]. In this recursive manner, we are able to generate all possible combinations of architectures involving the actors and variables on the PEs and memory elements.

Scheduling CPUs

Hardware blocks can run everything mapped to them in parallel, and therefore do not require any scheduling. However, software processors can only run one actor at one time. Therefore, the scheduling of actors on software PEs is another part of the design space. Within SCE, the following choices are available:

- Statically schedule all actors to run serially in a preset order. In this case, there is a single thread on the PE comprised of the serialization of all the actor threads.

[1] Architectures involving shared memories can be blocked altogether using a switch, in which case all variables will be mapped to the PEs. This is useful in reducing complexity of the decision set.

- Use a pre-emptive thread scheduler which schedules actors in a round-robin manner
- Use a pre-emptive thread scheduler which schedules actors with a priority based policy, in which an actor with a higher priority executes before others with lower priority.

Within our framework, only the round-robin and priority thread scheduling are considered as design space options[2]. Additionally, there is a subset of architectures within the priority scheduling, comprised of different priorities for different actors.

At this point, the entire design space consisting of computation-only architectures (i.e. ignoring communication effects such as bus delays) has been generated. The *sce_gds* service has the option of generating XML output files for this design space, since this design space can be useful as an intermediate step.

Bus Mapping

We restrict potential architectures in *sce_gds* to generic bus-based MPSoCs. Not only are bus-based architectures most common, memory mapped bus architectures are also the focus of the SystemC TLM 2.0 standard [1]. We further establish a set of possibilities and restrictions for bus architecture exploration:

- Each CPU needs to be connected to its own bus.
- PEs and shared memories cannot connect directly to each other, and can only be connected through a bus.
- There can be a dedicated memory bus which only connects to shared memory elements (and transducers to be able to talk to other busses).
- Similarly, there can be a dedicated Hardware bus with only Hardware PEs (and transducers).

[2] Complex embedded systems, which are the subject of this report, are very likely to require some sort of thread scheduling mechanism, if not a RTOS

- Finally, there can also be a bus that can have both Hardware processing elements and shared memory, but no CPU[3].

As a result of these rules, there will be a minimum of *n* busses in the design architecture; where *n* is the number of software processors. Each software processor in the design is then assigned a bus type, carefully making sure only valid bus types out of the available set are assigned. For example, a Coldfire processor cannot be assigned to an AMBA bus. The processor is connected to the bus as a master on the highest priority master port (e.g. Master0 for ARM9 processors).

Next, hardware PEs are recursively assigned to each of the busses or the dedicated HW bus. A hardware PE is connected both to a master and a slave port on the bus. For shared memory elements, there needs to be a decision on whether to connect the shared memory in a dual-port fashion to both busses that access it, or to connect it to a dedicated memory bus. Shared memory elements are connected to the bus as slaves.

At this time, it is also necessary to add transducers between PEs and memories that communicate with each other using channels, but are on different busses in the architecture. This can be done statically by identifying channels and the actors they connect. Transducers are connected as slaves on all the busses they connect with.

Write Design Architecture

To complete network connectivity, the following architectural elements still need to be defined:

- Master and slave priorities for various entities connecting to the bus

---

[3] This bus is available only in case there actually is at least one shared memory, AND at least one HW. In the absence of either, this bus would become the same as either a dedicated HW bus or a dedicated memory bus, and is hence not an exploration option.

- Synchronization policy (interrupt or polling)
- Interrupt priorities

These factors are ignored as part of DSE because they may not make a large difference in the exploration results, yet can dramatically increase the design space. Therefore, these elements are defined as part of the synthesis process described in the next Chapter.

Since all the design space parameters for this particular architecture are now known, the architecture description is written to a XML file. The framework also encodes the description in a key, and appends the key-to-XML mapping to a lookup table file, defining the link between the unique key for the architecture, and the file containing its description. The cost for this architecture is also written to the XML file (since it can be statically determined now that all the architecture elements are known)

This process is then repeated until all the design points in the potential design space have a description in the form of an XML file. At this time, the entire design space has been generated and represented.

3.3 SCE_REFINE: Model Refinement Service

A DSE system chooses a single or a set of decision points from a large pool for the purpose of determining their fit for the goals of the system. As part of this process, an executable model of the application within the platform architecture needs to be built as per the design point, and then the system estimates the performance parameters for this model. This section describes the *sce_refine* service, which provides the facilities for automatically generating such an executable model using SCE's basic refinement engine.

3.3.1 Design Modeling and Evaluation

As explained earlier, there can be three distinct levels of performance estimation relevant to the DSE process:

Static Analysis

Static models estimate the application performance based on profiling the application and extracting parameters such as number and types of mathematical operations, data movement operations etc. Platform component models already contain information about their behavior with respect to these parameters. For example, a DSP model will reflect the fact that it is 5 times faster than an ARM processor model at multiply-accumulate (MAC) operations. These extracted parameters are then used to statically estimate the performance of the application on the platform.

While this approach is very fast since there is no simulation involved, it is highly inaccurate when the architecture is complex, since it completely ignores any temporal effects in application execution. These temporal effects affect application parameters such as real-time latency, which are very important for embedded systems. Therefore we do not consider this approach to be effective in a DSE system, and do not provide any facilities for using this at the moment.

Computation-only Model Simulation

Computational-only simulation models are generated after SCE's architecture and scheduling refinement stages. As the name suggests, these accurately model computational effects, but do not model communication overhead. For example, no bus delay or bus contention effects will be seen in these models. These effects are very important especially in MPSoCs and cannot be ignored, making these models unsuitable for complete DSE.

However, this stage does provide a convenient intermediate point in DSE to evaluate various design points, and prune the design space before beginning DSE involving communication modeling, which can add large overheads in simulation and assessment time. Such a hierarchical, two-step DSE process can have a dramatic effect on overall DSE time and quality. By eliminating potentially a large percentage of design points, the complexity of the process going forward is significantly reduced. This is shown quantitatively with an example in Chapter 4.

Computation- and-communication Model Simulation

Communication model simulation, or TLM simulation, is done using TLM models generated after SCE's communication and network refinement. These models are only one level of abstraction above RTL and C code/instructions, and capture most effects that are required for effective DSE, but due to the added detail, they do so at exponentially larger simulations times compared to earlier computation-only models.

3.3.1 SCE_REFINE Description

The *sce_refine* service, shown in Figure 3.3, is used to generate both computation-only and computation-and-communication simulation models. *Sce_refine* engages refinement for each

design point in the selected pool to generate a simulation model from an architecture description in one shot. It also deals with any errors in generating simulation models. The sub-sections below describe the operation of *sce_refine*.

Read Design

The application can be read in as a SpecC code file, or a pre-processed SpecC Internal Representation (SIR) file. The system builds an internal database of the parameters of the design, such as the actors, channels and their connectivity. Users can specify the top-level of the design. This is useful when the architecture for a portion of the design is already known, and we only need to perform design space exploration for a particular sub-system.

Read Architecture Description

The system next reads in the architecture description from the provided XML file (generated in the design space generation stage) and builds it into an internal database. At this point of time, it performs sanity checks to ensure consistency between the design and specified architecture. Amongst other things, it makes sure that:

- there is no duplicity in mapping
- at least one PE and one system bus is specified
- processes and target PEs specified for mapping actually exist in the design and SCE model database, respectively

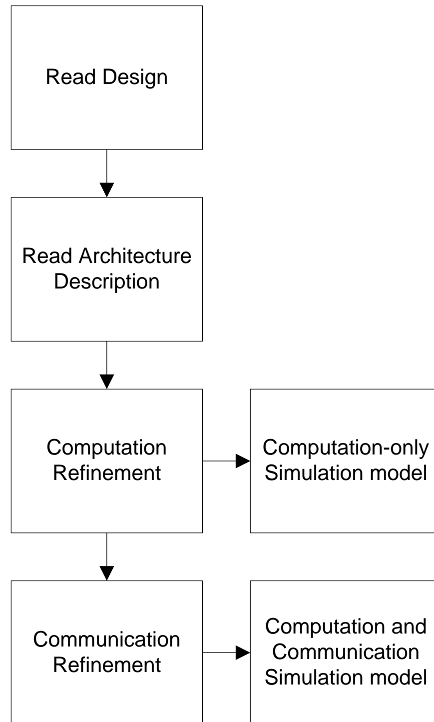


Figure 3.3: *sce_refine* flow

Computation Refinement

The system exercises SCE to implement computation and scheduling decisions as specified in the architecture description. The system is also built to handle a particular case of when PEs are scheduled with a priority scheduler, but actors are not assigned priorities in the architecture specification. In this case, the system automatically and randomly assigns unique priorities to actors.

Communication Refinement

Again, the system exercises SCE to implement communication and network decisions specified in the architecture description, as well as a number of decisions which may not be specified in the architecture, but are necessary to complete the specification. Specifically, the following architectural elements are defined automatically:

- Master and slave numbers on the bus are assigned randomly, except the processor on the bus which is always assigned Master0. This is done under the assumption that all the masters and slaves will have the same priorities on the bus[4]
- All synchronization is interrupt based. Polling-based synchronization mechanisms are not considered
- Interrupt numbers and addresses are assigned randomly to channels, under the assumption that all interrupts are of equal priority[5]

Generate Simulation Model

Simulation model are generated after both computation and communication refinement to complete synthesis. These models are used further by the DSE system to measure design evaluation parameters such as latency and power consumption.

---

[4] Therefore, bus contention effects arising out of priority assignment cannot be seen in the design space exploration

[5] This means processor contention effects arising out of interrupt priorities cannot be seen in the design space exploration

3.4 SCE_SIMULATE: Simulation and Assessment Service

This section describes the *sce_simulate* service, which provides simulation and assessment facilities. The *sce_simulate* service simulates the design model on the host for all the design points in the current pool. The design model is expected to contain an embedded testbench that assesses and generates performance evaluation parameters. Since the testbench is part of the design, and therefore out of the scope of the DSE system, it may refer to the output parameters from the simulation in a number of ways. For example, latency may also be referred to as output delay or jitter in the simulation output. The *sce_simulate* service therefore requires the user to define keywords associated with design evaluation parameters, such as 'latency'. The exploration framework then provides a mechanism to extract these parameters from the simulation results, and accumulating them in an intermediate results output file.

In a similar manner, *sce_simulate* also extracts statically determined evaluation parameters, such as cost, from the architecture specification and includes them in the output file. The output file is used by the decision making engine to prune the pool of design space points, or it can be processed by a number of tools to compute and plot weighted cost and performance functions. In the next Chapter, we show the parsing of results in MATLAB to generate a visual representation of the Pareto-optimal front.

Chapter 4: Demonstration and Results

We demonstrate the capabilities of our framework by implementing two relatively simple DSE systems using a brute force approach, and a hierarchical two-step exploration approach. These systems are then exercised with an industrial strength example design [7]. More advanced systems, using genetic algorithms for example, can be built using this framework but are beyond the scope of this work.

4.1 Simple Brute Force Exploration Flow

Figure 4.1 shows a DSE flow that implements a one-shot, brute force exploration flow. The entire design space representation is generated initially. A set of design points is then picked and taken through synthesis, simulation and gathering of results. This is repeated until all the design space points have been simulated and assessed. At this time, all the results can be plotted and the Pareto-optimal curve can be obtained.

4.2 Two-step Brute Force Exploration Flow

Figure 4.2 shows an optimized DSE flow that exploits the step-wise refinement capabilities of SCE by utilizing a hierarchical, two-step exploration flow. In the first pass, solely computation-only models are considered, i.e. communication effects such as bus delays are ignored. This has a twofold benefit:

1. The design space pool without considering communication decisions is potentially much smaller
2. Models without communication level of detail execute much faster

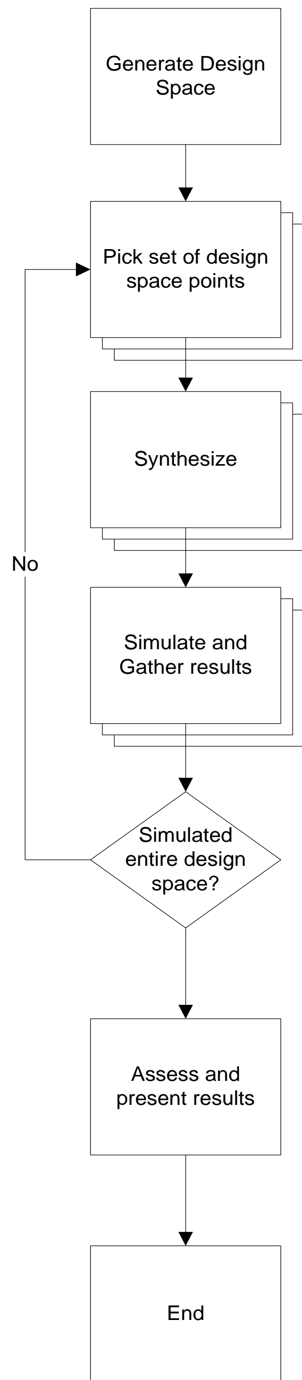


Figure 4.1: Brute-force DSE flow

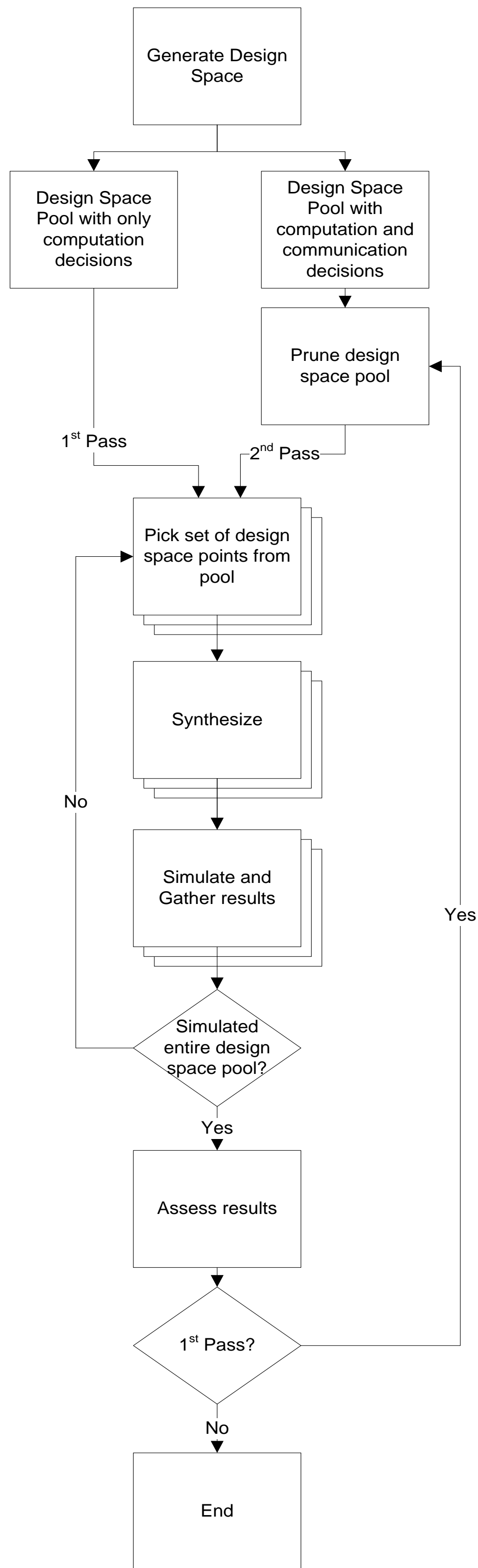


Figure 4.2: Hierarchical, two step DSE flow

As a result, we can run through a first pass of exploration quickly, and use results to prune the design space. There can be a number of strategies to pruning the design space. At the very least, the design space can be pruned of design points that are highly likely to be infeasible. For example, if the application latency for a set of architectures already exceeds the latency constraint, latency can only get worse once bus delays are considered. DSE systems can also choose to be more aggressive and select design points lying on the first and second-level Pareto-optimal curves. Second-level Pareto-optimal curves are expected to account for inaccuracies arising out of ignoring communication effects. Once the computation-only design space is pruned, we then select only the subset of the computation-and-communication design space that is derived from this pruned region. A second pass of exploration then provides final results.

4.3 RESULTS

Using the two DSE systems described above, we perform design space exploration for an example design. Figure 4.3 shows the chosen task set, which is composed out of a modified subset of applications from the automotive category of the MiBench suite. The tasks *basicmath*, *qsort* and *susan_edge+susan_corner* are scheduled to run periodically with periods of 2.5, 2 and 1 second(s) respectively. The output of *susan_edge* is thereby communicated to the input of *susan_corner*, forming a pair of dependent tasks. The resulting task set is run for 10 seconds of simulated time.

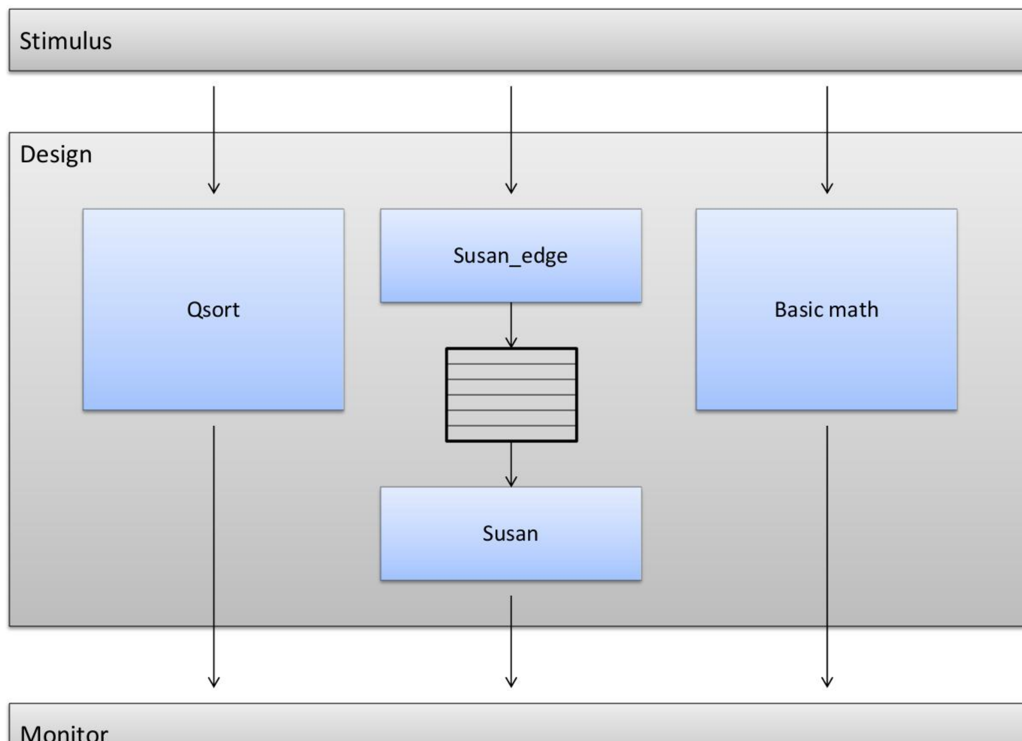


Figure 4.3: MiBench design

We explored a design space with up to four ARM processors, a choice of two OS schedulers (priority-based or round-robin) and required communication architectures in the form of busses and bridges. We consider an exploration depth of 2. This forms a design space of almost 2000 unique architectures. By utilizing a one-shot brute force approach using our framework (Figure 4.1), we automatically generated executable models for all architectures. Simulating those models provides total delay, power consumption and cost for each design alternative. This results in 35 distinct design points (Figure 4.4), where the set of Pareto-optimal designs, at 3 points, comprises less than 1% of the total design space. As shown in Table 4.1, this approach was able to evaluate all 2000 design alternatives in 47 hours of CPU time on a cluster of 2.5GHz Intel Xeon workstations.

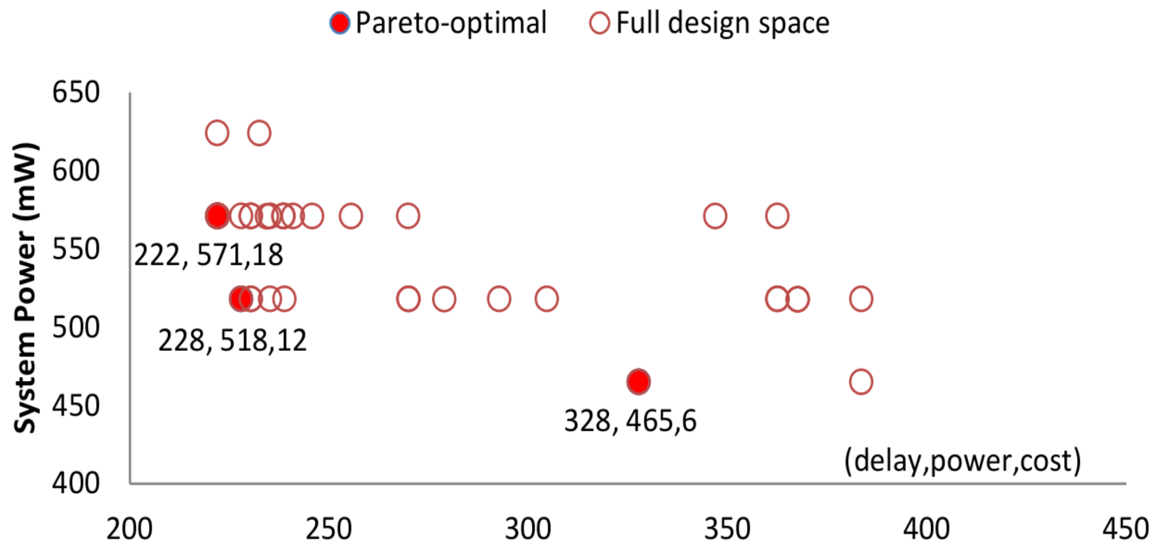


Figure 4.4: Pareto-optimal front for MiBench DSE

	CPU Time (hours)
Design Space ISS	500
System-level full DSE	47
System-level step-wise DSE	8

Table 4.1: MiBench DSE time

Further, the design was tested with a two-setup brute force approach as shown in Figure 4.2. A pruning strategy that retained both the first and second Pareto fronts after the first step was used. Moving to this exploration approach enabled the same results to be obtained in 8 hours of CPU time.

Chapter 5: Conclusion

We have successfully developed a Design Space Exploration infrastructure that captures the opportunity to consolidate various algorithmic approaches under one common framework using generic back-end services. Such an approach enables the right front-end optimization approach to be used based on design characteristics. It also enables advances in underlying fields of ESL design, such as simulation and modeling, to be incorporated into the design space exploration process in the fastest possible manner. We go on to implement two exploration flows using this framework, and we evaluate a representative industrial-strength application to prove the feasibility and value of this approach.

We can imagine a number of extensions and improvements to the *sce_explore* environment. Some of these are listed below

- Currently, the interface to a graphing utility such as MATLAB is manual. This can be automated by incorporating an API that calls a standalone application created using MATLAB
- The pruning step in a two-step exploration is currently done by the user. This can be improved by incorporating APIs that automatically prune the design space using some of the strategies described earlier.
- Interrupt priorities are assigned randomly at present. Master and slave assignments are also made randomly. The design space generation framework can be enhanced to include decision making on these parameters. This can be important in systems such as communications processors, where interrupt planning is very important owing to a large number of interrupts with differing latency requirements.

- The cost estimate can be improved. For example, currently, a shared memory has the same cost whether it is a single or dual ported memory, and regardless of its size. This is not true of actual designs.

References

[1] "SystemC," [Online]. Available: http://www.systemC.org

[2] R. Domer, A. Gerstlauer, J. Peng, D. Shin, Cai, L., Yu, H., Abdi, S., Gajski, D.: System-on-Chip Environment: A SpecC-based Framework for Heterogeneous MPSoC Design. EURASIP JES 2008(647953), 13 (2008).

[3] J. Keinert, M. Streubuhr, T. Schlichter, J. Falk, J. Gladigau, C. Haubelt, J. Teich and M. Meredith: System CoDesigner – an automatic ESL synthesis approach by design space exploration and behavioral synthesis for streaming applications. ACM transactions on design automation of Electronic systems, Vol. 14 no. 1, article 1, January 2009.

[4] S. Mohanty, V. K. Prasanna and S. Neema, J. Davis: Rapid Design Space Exploration of Heterogeneous Embedded Systems using Symbolic Search and Multi-Granular Simulation. LCTES'02–SCOPES'02, June 19-21, 2002.

[5] H. Nikolov, M. Thompson, T. Stefanov, A.D. Pimentel, S. Polstra, R. Bose, C. Zissulecu and E.F. Deprettere: Daedalus: Towards composable, multimedia MP-SoC Design.

[6] "SpecC", [Online]. Available: http://www.cecs.uci.edu/~specc

[7] A. Gerstlauer, S. Chakravarty, M. Kathuria, P. Razaghi: Abstract System-Level Models for Early Performance and Power Exploration. ASPDAC, Sydney, Australia, January 2012.

[8] P. Razaghi, A. Gerstlauer: Predictive OS Modeling for Host-Compiled Simulation of Periodic Real-Time Task Sets. IEEE Embedded System Letters (ESL), vol. 4, no. 1, March 2012.

[9] E. Zitzler, L. Thiele: Multiobjective evolutionary algorithms: A comparative case study and the strength pareto approach. IEEE Trans. on Evolutionary Computation, vol.3, no.4, pp.257-271, Nov 1999