View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by ScholarBank@NUS

Synthesis of Multiprocessor Architectures for Multimedia Applications

Dinesh Kunchamwar

(B.E. (Hons.) Computer Science)

Supervised by

Samarjit Chakraborty

Weng Fai Wong

A THESIS SUBMITTED FOR A DEGREE OF MSC(BY RESEARCH) COMPUTER SCIENCE

SCHOOL OF COMPUTING

NATIONAL UNIVERSITY OF SINGAPORE

2006

Abstract

Streaming applications typically consist of a large number of tasks, each of which can be mapped onto a different processor on a Multi-processor System on Chip (MPSoC) platform. This gives rise to a large design space (whose size can be exponential in the number of tasks in the application). Previous work on design space exploration for this problem use evolutionary algorithms which do not guarantee optimality and are too time-consuming because of the need for full-system simulation. This problem has also been formulated as Integer Linear Programming problem assuming constant execution requirements. However high degree of burstiness in the arrival rate of streams and high variability in the execution demand of the data items, make those approaches inadequate. In this work we present a depth-first design space exploration technique which breaks down the process of system simulation into many task level simulations. The results of the task level simulations are represented using Variability Characterization Curves (VCCs) which are based on theory of network calculus. We formulated an algorithm which explores the design space in a depth first manner and combines the VCCs of task mappings along the explored path using purely analytical methods. The algorithm avoids exhaustive searching of the design space by using certain timing conditions and upper bounds on the costs to prune some portions of design space. We take the multi-objective optimization approach for this problem and device a technique that is capable of finding a pareto optimal front. This allows the designer to make trade-offs among various design goals to quickly narrow down the choice of various architecture design parameters. We implemented our scheme and performed a case study based on the MPEG-2 decoder application. We demonstrated the usefulness of our multi-objective technique using two objectives: the silicon area required to realize the MPSoC system and power requirements of the system. For this case study, 76% of the design space was pruned due to the timing and upper bound conditions which translated into 74% saving in the running time of the algorithm.

ACKNOWLEDGEMENTS

I feel deeply indebted to my supervisors Dr. Weng Fai Wong and Dr. Samarjit Chakraborty, without their support, guidance and constant motivation this thesis would not have been possible. I would like to sincerely thank them for allowing me to learn from their knowledge and experience and for their patience and encouragement which helped me to get through difficult times.

I thank my thesis committee members: Dr. Tulika Mitra and Dr. Wei Tsang Ooi, who evaluated my Graduate Research Paper and provided valuable suggestions during thesis proposal presentation.

Special thanks to all fellow students in the Embedded Systems Lab who shared their expertise and helped me in times of need. Thanks to Yanhong and Balaji for help on SimpleScalar, Priya, specially for help on latex, Unmesh, Yuan Yi, and Ramkumar for sharing their thoughts and experiences and Linh for her kind words.

Thanks to all my friends at hong leong for all the happiness and fun that we shared. I deeply appreciate all the moral support Laura provided and thank her for being with me. I express my sincere gratitude towards my parents for being constant source of motivation and for the values they have inculcated in me.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

Streaming application domain consists of a wide variety of applications from embedded systems (e.g. cell phones, hand held computers), desktop applications (e.g. real-time encryption, software radio, graphics packages) and high performance servers (e.g. network packet processors, cell phone base stations, software routers). Designing and implementing efficient HW/SW systems for these applications has been the focus of research for many researchers due to the huge commercial impact these systems create. With the increasing use of streaming media in hand held devices and increasing performance demand of applications running on these devices, it becomes more important for a system designer to utilize the system resources to the fullest. Although existing design methodologies for the HW/SW co-design of embedded systems can be applied to these systems, due to the ever growing complexity of the applications they may not be as efficient. Many multimedia streaming applications have to handle a highly compressed stream of media content. Most of the compression techniques exploit the similarity between successive frames in order to achieve more compression, and this similarity is not constant throughout the length of media. Highly compressed portions of the media need more processing for decoding and less compressed portions the less. Thus, for such media the decoder has to face the variable processing demand of the different portions of the stream. The system design methodologies can exploit this variability to optimize the system performance and resource utilization. Traditional design methodologies, hence, must be upgraded by taking into account these changing nature of applications.

In a streaming application, a potentially infinite stream of data items enters the system, a number of tasks are performed on each data item before the output is sent to an output device. Typically the tasks performed on data items are independent and communicate with each other by FIFO channels. Each task reads data items from its input channel in a FIFO order, does some processing and writes the resulting data into the input channel of the next task. Due to this data dependent behavior of tasks, they can be mapped to a different processing element (PE) in a multiprocessor system. Portions of

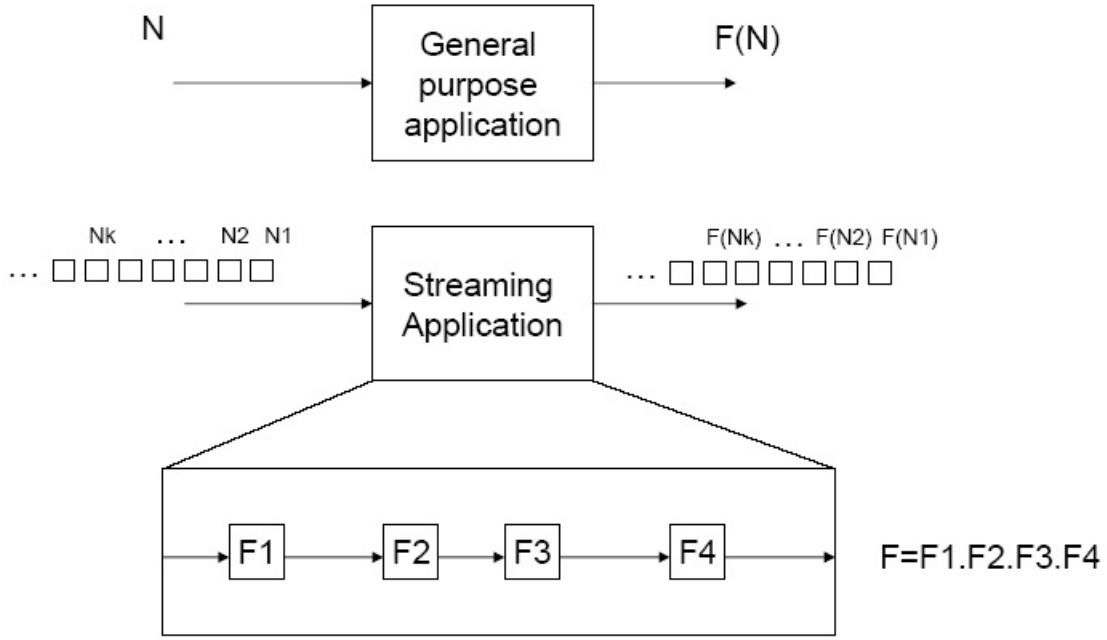


Fig. 1.1: Streaming Application

memory can be allocated as the FIFO channels. Both, the PE which writes to a channel and the PE which reads from the channel should share the memory allocated to the channel.

System-on-chip multiprocessors are widely used to implement these streaming applications due to their advantages in terms of performance, power, cost, and design turn-around time. In order to realize these advantages to the fullest, it is necessary to have an efficient design methodology which addresses the two aspects of the system design. First, efficiently mapping of tasks in the application to available computation units and second, designing an optimal communication network. In this work we concentrate on the first aspect.

Typically, a number of different processing tasks constitute a streaming application. Every task may have different processing requirements in terms of sheer processing power or control-intensiveness and computation intensiveness. Depending on these requirements, different processing architectures may be suitable for different tasks. For example, DSP's are most suitable for tasks which have regular computational loops where as general purpose processors are best for control intensive tasks. Further, in some streaming applications (e.g multimedia decoders) processing requirements are input dependent, in which case the processing requirements vary for different portions of the stream.

For example the execution time required for decoding different frames of a mpeg2 decoder is highly variable. Further, with the advances in processor technology, ever growing number of processors are becoming available for mapping of tasks. Thus the system designer is faced with the difficult task of mapping the tasks on processing units in order to optimize overall system performance. In order to make the systems commercially feasible, cost of the overall system (in terms of silicon area and/or power consumption) must also be taken into account. This problem has been addressed mainly in 2 different aspects: 1) Given fixed amount of silicon resources, find the mapping such that system performance is maximized and overall system can be implemented on given amount of silicon. 2) Given an application specification with minimum performance requirements, find the mapping such that overall system can be implemented with minimum cost (silicon area or power). In first, the cost of resources acts the constraint and performance as the goal function. This approach is most suitable for network packet processors, or cell phone base stations etc. since network traffic might go up in future and it is desirable to push as much traffic as possible. However in this work, we take the second approach, where performance requirements act as constraints and overall costs of the system acts as the optimization function. This approach is more suitable for multimedia applications for which the output performance requirements are fixed (e.g. 20 frames/sec for video decoders).

Our technique uses initial simulation of tasks on individual processor types in order to estimate the worst case bounds on the execution requirements of the application. Using some results from the theory of network calculus, it is possible to combine the worst case bounds for many task-processor mappings to find the effective worst case bounds. This results in significant simulation time savings as follows. If T is the number of tasks in the application and n is the number of candidate architectures, the total simulation time required is: $\sum_{0<i<T} \sum_{0<j<n} t_{ij}$, where t_{ij} is the time required for simulating i^{th} task on j^{th} processor. Let $t_{max} = max(t_{ij})\ \forall i,\ \forall j$. Then $\sum_{0<i<T} \sum_{0<j<n} t_{ij} \leq t_{max} \times T \times n$, which is $O(nT)$. On the other hand if we explore the design space exhaustively and perform system level simulation for each design point, then $total\ \ simulation\ \ time \leq t_{max} \times T \times n^T$ which is $O(Tn^T)$. However the designer, with his experience, may be able to quickly eliminate some processors which are not suitable for some tasks. After such elimination, let $n_i < n$ be the number of processors on which task T_i can be mapped. In this case the design space has $\prod n_i,\ n_i < n\ \forall i$ design points, and $simulation\ time \leq t_{max} \times T \times \prod n_i$ for exhaustive design space exploration with full system

level simulation. However with the task level simulation approach $simulationtime \leq t_{max} \times \sum n_i$. It should be noted, however, that a full system simulation allows a more accurate estimation of performance as well as communication delays. The task level simulations are aimed only at performance estimation and are based on the assumption that once task mapping is done, a suitable communication network can be designed. This approach of independently exploring computation and communication architectures has been followed by many researchers as mentioned in Section 1.1.

We used a depth-first approach for design space exploration, with every level in the design tree representing a mapping decision for a task and the number branches at every level representing the number of possible mappings. The cost of the explored path is computed by adding the costs of the candidate architectures chosen for tasks on the path. By candidate architecture we mean a processor with some microarchitectural properties like cache sizes, decode/issue widths etc. All commercial processors can be uniquely described by these microarchitectural properties. Hence we find it is more convenient to define the candidate architectures using their microarchitectural details.

A MPSoC designer is often faced with the challenge of designing systems with more than one design goals in mind. For example, in [31] performance of access network, performance of backbone network, and cost savings are used as objective functions. In [7] silicon cost of the system has been used as the objective function. One could imagine many more objectives like power, threshold etc. which could be of practical use to the designer. Further these objectives may be conflicting with each other, optimizing on one objective may result in degradation on the other objectives. In such a situation, it is essential for the designer to be able to evaluate trade-offs between different design points. Our DSE technique is capable of handling this multi-objective scenario and can produce a Pareto-Optimal front for a given set of objectives.

We used MPEG2 decoder application as the target application and implemented our scheme to find the pareto optimal front with mainly two objectives in mind: the architecture cost of the system and the average power consumption of the system. The architecture costs for different candidate architectures were derived from an empirical cost model described in [30] and the power numbers were obtained from Wattch simulator [2]. Our experiments prove the usefulness of the scheme.

1.1 Related Work

Traditionally, most commercial SoCs are platform based [26, 16]. PROPHID [15] multiprocessor architecture, for example, is a bus based architecture with a general purpose processor for control processes and a number of Application Domain Specific (ADS) processors for performing tasks in the applications. ECLIPSE [26, 27] is another such heterogeneous SoC platform aimed at implementation of multimedia applications. They consist of a predefined communication architecture with a possibility of customizations in terms of choice of processing units, scheduling policies on the processing units, and on-chip buffer size requirements. One of the popular approaches for application design for platform based systems has been a cyclic process of performance estimation and tuning the platform for the application. Many evolutionary algorithm based techniques are developed for this purpose[31]. Many of these perform full system-level simulation [25] which makes them very time-consuming. Many trace-based simulation techniques are developed in order to reduce the simulation time [17, 34]. SPADE [33] is a method and a tool in which applications and architectures are modeled in such a way that mapping of applications onto architectures, prior to architecture/application co-simulation, can be easily done. The co-simulation is trace-driven. The execution of the application generates traces of symbolic instructions, and these instructions are interpreted by the architecture thus revealing timing behavior. The TD cosimulation is fast because the architecture does not process the actual data. However the evolutionary approach does not guarantee optimality of the solution.

In DSE of multiprocessor SoC systems, many possible mappings of tasks onto different processors (e.g. GP, DSP, ASIC, FPGA) is possible. In order to find out *how good* a particular mapping is, there is a need for metrics which characterize each type of PE. In this context, the work in [28] finds out the *affinity* of each task toward different types of processors. An affinity of 1 indicates a perfect matching and 0 indicates no match. Given the source code for a task, they perform static analysis to find out metrics like *Control Flow Complexity (CFC), Loop Ratio (LR)* etc. A high value of CFC identifies a task which has lot of control flow and less computation. These tasks are suitable for GPPs. On the other hand a high value of LR indicates that the task is computationally intensive with lot of regular computations. These tasks are more suitable for DSPs. There are many more such metrics and weighted average of all metrics is taken to arrive at a global affinity value.

Ever growing demand of system performance forces the system designers to integrate more and

more processing elements in a single SoC to meet the performance requirement. Many researchers [12, 24, 14, 11] have proposed a new system design paradigm for such high performance SoCs, which involves separation between (1) function and architecture and (2) between communication and computation. Adopting this paradigm, design methodology proposed in [12], models the system behavior as a composition of function blocks and maps the function blocks to the processing elements of predefined target architecture. Separation between computation and communication in system design allows the system designer to explore the communication architecture independently of component selection and mapping.

An important aspect of system design is the representation of application using appropriate application model. Most of the work in the direction of HW/SW co-design [13, 4, 3, 18, 29] treats the application as a Task Graph [13, 3]. The tasks in such a task graph have data dependencies and a task can execute only when all its dependencies are satisfied. SoC design for such task graphs is typically solved as a scheduling problem. However as the applications become more and more complex, the number of tasks in the task graph grows rapidly. Representation of the applications using task-graphs thus becomes more cumbersome. Of late, Kahn Process Networks [10] have been used more popularly as the Model of Computation (MoC).

Synthesis of multiprocessor architecture for KPNs has been addressed in [8, 7, 6, 23, 19, 5]. The mapping problem is formulated as an ILP [8, 7] and thus produces optimal solution. However they assume fixed processing requirements for the tasks and regular memory access pattern, which may not be realistic in case of multimedia applications.

1.2 Contributions

In this thesis, we present an innovative design space exploration (DSE) method using depth first exploration approach. Figure 1.2 depicts an overview of our method. We use Kahn Process Networks to model the applications. The aim of the DSE is to find the optimal mapping of processes (or tasks) to processing units (or candidate architectures). Our technique can handle multi-objective optimization scenario and produces a Pareto-Optimal front. The designer can use the pareto-optimal front to choose trade-offs between different objective functions. We use a simulation based performance estimation, which is not a new technique. However our approach is novel in that we perform simulation at the

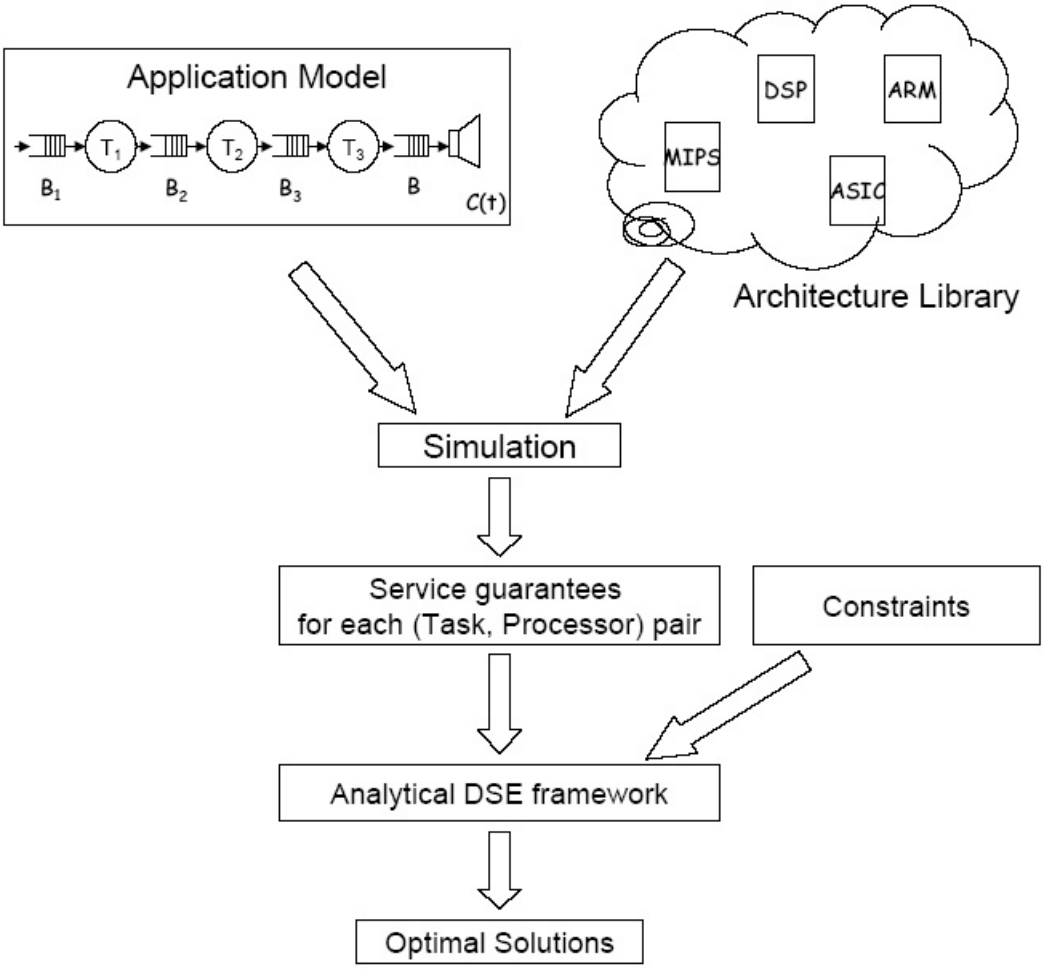


Fig. 1.2: Overview of DSE technique

task level rather than system level. This gives the system designer flexibility of modifying a task and simulating only that task again. In this thesis, we used variability characterization curves or VCCs [21] to model the variability in the execution demand of multimedia applications. An overview of VCCs is also given in Chapter 3. Thus the mapping found by our algorithm is more accurate as compared to previous techniques which assume constant execution demand. The execution trace obtained from simulation, which essentially records the number of processor cycles required for each activation of a task, is used to find the VCCs for each $(Task, Processor)$ pair. Once these individual curves are obtained, we use purely analytical methods from the theory of Network Calculus [1] to combine the individual VCCs for each task in order to estimate the overall system performance. Our depth-first approach for searching the design space allows us to incrementally combine VCCs depending on which path is chosen for further searching. We show that it is possible to prune away certain portions of the design space using conditions specified in terms of timing constraint of the application which is another important contribution of this work. This leads to faster design space exploration without compromising the optimality of the solution. Thus our technique combines the advantages of ILP based techniques which guarantee optimality and the full system simulation based evolutionary techniques which are more accurate in performance estimation. We also compute the minimum buffer sizes required to prevent buffer overflows. Thus the optimal solution found by our DSE technique not only guarantees required output rate but also guarantees that none of the buffers overflow.

1.3 Organization of the thesis

The two most common models of computation (MoCs), the Task Precedence Graph and Kahn Process Networks are discussed in Chapter 2. This chapter also discusses the advantages and disadvantages of the two MoCs and why we choose the later in this work. Chapter 3 describes the Variability Characterization Curves (VCCs) and how they can be used to represent the worst case characteristics of the on-chip traffic of data items created by the execution of different tasks. In Chapter 4, we develop an analytical framework using the VCCs for our design space exploration problem and provide some conditions which help us in pruning of the design space. Chapter 5 describes the experiments we performed in order to validate the usefulness of our DSE technique. We performed two sets of experiments, one using different cache configurations of the MIPS processor and another

using different frequencies for the same processor. However it should be noted that our technique is fairly generic and can be applied to any processor architecture, subject to the availability of a simulation framework for that architecture. Chapter 6 summarizes the main contributions of this thesis and concludes.

2. CHOICE OF APPLICATION MODEL

The main goal of application modeling is to use suitable Models of Computation (MoCs) for specifying a multi-media applications. Ideally, the MoC should allow a succinct representation of the application so that the inherent parallelism is exposed and can be exploited when mapping it an architecture. Two distinct models are generally used to describe streaming applications or parallel applications - Task Precedence Graph (or directed acyclic graphs) and Process Networks. Synthesis of optimal application specific multiprocessor architectures for DAG based periodic task graphs with real time processing requirements has been extensively studied in literature [4, 3, 18, 29] from cost as well as power optimization point of view. However, synthesis for process networks is not widely explored. The approaches for DAG based task graphs rely on static scheduling and solve the synthesis problem which essentially consists of architectural resource allocation, binding of application components to architecture, and scheduling. This approach can also be used for process networks as well by unrolling inner loops of processes and decomposing it in the form of a periodic DAG based task graph. Such an approach might lead to a very large size graph for most of the real life applications making the approach impractical.

2.1 Task Precedence Graph

In a task precedence graph, nodes represent the tasks and the directed edges represent the execution dependencies as well as the amount of communication. DAGs are commonly used in static scheduling of a parallel program with tightly coupled tasks on multiprocessors. For example, in the task precedence graph shown in Figure 2.1, task n4 cannot commence execution before tasks n1 and n2 finish execution and gathers all the communication data from n2 and n3. For most applications, a task precedence graph can model the program more accurately because it captures the temporal dependencies among tasks.

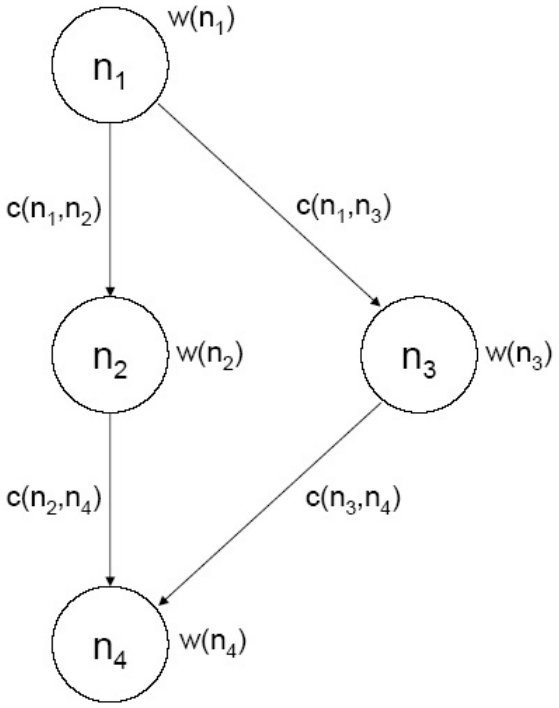


Fig. 2.1: Task Precedence Graph

Formally, A parallel program can be represented by a directed acyclic graph (DAG) $G = (V, E)$, where V is a set of v nodes and E is a set of e directed edges. A node in the DAG represents a task which in turn is a set of instructions which must be executed sequentially without preemption in the same processor. The weight of a node n_i is called the computation cost and is denoted by $w(n_i)$. The edges in the DAG, each of which is denoted by (n_i, n_j), correspond to the communication messages and precedence constraints among the nodes. The weight of an edge is called the communication cost of the edge and is denoted by $c(n_i, n_j)$.

Synthesis of application specific architecture for an application described using DAG has been solved as resource allocation, binding of application components to resources and scheduling problem in [13].

The objective of scheduling is to minimize the completion time of a concurrent application (such as streaming) by properly allocating the tasks to the processors. In a broad sense, the scheduling problem exists in two forms: *static* and *dynamic*. In static scheduling, which is usually done at compile time, the characteristics of a parallel program (such as task processing times, communication, data

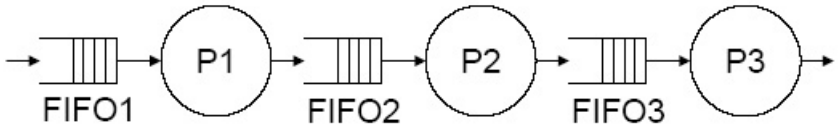


Fig. 2.2: Kahn Process Network

dependencies, and synchronization requirements) are known before program execution. A parallel program, therefore, can be represented by a node- and edge-weighted directed acyclic graph (DAG), in which the node weights represent task processing times and the edge weights represent data dependencies as well as the communication times between tasks.

2.2 Process Networks

We specify media-processing applications as a set of concurrently executing tasks that exchange information solely by unidirectional streams of data. A directed graph with a node for each task and an edge for each data stream represents the structure of the application. Process network, introduced by Kahn in [10], is popularly used as the model of computation (MoC) for specifying these kind of applications. A Kahn process network (KPN) consists of executable *processes* that communicate point-to-point over unbounded FIFO *channels* and synchronize by means of blocking reads. Figure 2.2 shows an example of KPN.

We chose Kahn Process Networks as application model due to the fact that Kahn models nicely fit with the dataflow application domain (to which most of multimedia applications belong) and that they are *deterministic*. The latter means that the same application input always results in the same application output. So, the functionality of a Kahn application is not affected by architectural latencies, i.e. the application behavior is architecture independent.

3. VARIABILITY CHARACTERIZATION OF STREAMING APPLICATIONS

We model streaming applications using the model depicted in Figure 3.1. The application consists of T tasks (or processes), $P_1, P_2, ..., P_T$. Each task P_i has an input buffer B_i which is a FIFO channel of fixed capacity. These buffers are used to store the input stream temporarily until it is consumed by the processor. Each task P_i can be imagined as an infinite loop. At the beginning of each iteration it reads a data item from its input buffer B_i. The body of the loop performs some processing on this data item and finally the output data item is written to the input buffer of the next task i.e B_{i+1}. The output of the last task is written to the *playout buffer* B. A real-time client (RTC) such as a audio or video output device consumes data items from playout buffer at a fixed rate C (e.g. 20 fps for a video device).

For the sake of generality, we consider any multimedia stream to be made up of a sequence of *stream objects*. A stream object might be a bit belonging to a compressed bit stream representing a coded video clip, or a macroblock, or a video frame, or an audio sample—depending on where in the architecture the stream exists. For example, in case of an MPEG-2 decoder the stream objects are nothing but the macroblocks which constitute the frames in the video.

Variability characterization curves (VCCs) are used to quantify best-case and worst-case characteristics of *sequences*. These can be sequences of consecutive stream objects belonging to a stream, sequences of consecutive executions of a task implemented on a PE while processing a stream, or sequences of consecutive time intervals of some specified length. A VCC $\mathcal{V}$ is composed of a tuple

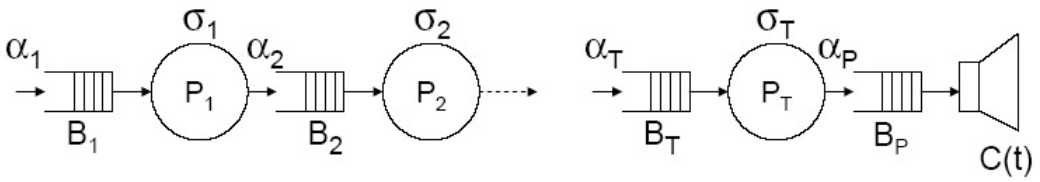


Fig. 3.1: Application Model

$(\mathcal{V}^l(k), \mathcal{V}^u(k))$. Both these functions take an integer k as the input parameter, which represents the *length* of a sequence. $\mathcal{V}^l(k)$ then returns a *lower bound* on some property that holds for *all* subsequences of length k within some larger sequence. Similarly, $\mathcal{V}^u(k)$ returns the corresponding *upper bound* that holds for *all* subsequences of length k within the larger sequence. Let the function P be a *measure* of some property over a sequence $1, 2, \ldots$. If $P(n)$ denotes the measure of this property for the first n items of the sequence (i.e. $1, \ldots, n$), then $\mathcal{V}^l(k) \leq P(i+k) - P(i) \leq \mathcal{V}^u(k)$ for all $i, k \geq 1$.

For every input buffer B_i, let $x_i(t)$ which represent the number of data items arriving in the buffer in an interval of time from 0 to t. We define arrival curve $\alpha_i = (\alpha_i^u, \alpha_i^l)$ for i^{th} task as:

$$\alpha_i^u(\Delta) = sup_{t \geq 0}\{x_i(t+\Delta) - x_i(t)\} \forall \Delta \geq 0$$
$$\alpha_i^l(\Delta) = inf_{t \geq 0}\{x_i(t+\Delta) - x_i(t)\} \forall \Delta \geq 0$$

$\alpha_i^u(\Delta)$ (upper arrival curve) denotes the maximum number of data items arriving in B_i during any interval of length Δ and $\alpha_i^l(\Delta)$ (lower arrival curve) denotes the minimum. Thus using arrival curves we can capture the worst case burstiness of the streams arriving in the buffers.

Similarly with every task P_i we associate $y_i(t)$ which represents the number of data items processed by the task in an interval of time from 0 to t. And the service curve $\sigma_i = (\sigma_i^u, \sigma_i^l)$ is defined as:

$$\sigma_i^u(\Delta) = sup_{t \geq 0}\{y_i(t+\Delta) - y_i(t)\} \forall \Delta \geq 0$$
$$\sigma_i^l(\Delta) = inf_{t \geq 0}\{y_i(t+\Delta) - y_i(t)\} \forall \Delta \geq 0$$

$\sigma_i^u(\Delta)$ (upper service curve) denotes the maximum number of data items processed within any interval of length Δ and $\sigma_i^l(\Delta)$ (lower service curve). Thus using service curves we can estimate worst case service guarantees for an incoming stream.

In this work, we want to ensure that the streaming application produces its expected real-time output even in the presence of the worst case arrival rate α_i^l and worst case service rate σ_i^l. Thus we use only the lower arrival curves and service curves in the rest of the paper. And for the simplicity of notation we use α_i and σ_i instead of α_i^l and σ_i^l respectively.

Using standard results from the theory of network calculus[1], we compute $\alpha_{i+1}(\Delta)$ as:

$$\alpha_{i+1}(\Delta) = (\alpha_i \otimes \sigma_i)(\Delta) \forall i \tag{3.1}$$

where, for any two functions f and g, the min-plus convolution of f and g is given by $(f \otimes g)(t) = \inf_{s:0 \leq s \leq t}\{f(s) + g(t-s)\}$.

The minimum buffer size required to prevent the input buffer of i^{th} task from overflowing can be calculated as:

$$B_i = sup_{\Delta \geq 0}\{\alpha_i(\Delta) - \alpha_{i+1}(\Delta)\} \tag{3.2}$$

And the minimum playout buffer size required to prevent overflow can be calculated as:

$$B_p = sup_{\Delta \geq 0}\{\alpha_p(\Delta) - C(\Delta)\} \tag{3.3}$$

Please refer to [1] and [21] for the detailed derivation of these results. Thus the total on-chip buffer required to prevent any of the buffers from overflowing is computed as:

$$B = \sum_{1}^{T} B_i \tag{3.4}$$

These Variability Characterization Curves (VCCs) have been used [21] in order to identify system level design trade-offs for multimedia processing SoC platforms. We use VCCs to characterize each task when mapped to different processing units, and identify the optimal mapping of tasks to processing units.

4. DESIGN SPACE EXPLORATION

4.1 Problem Description

Streaming applications consist of a sequence of tasks, each of which has a different processing requirements. Some tasks are control intensive for which general purpose processors like MIPS or ARM are most suitable. On the other hand DSPs are more suitable for tasks which are computationally intensive. At a much lower level, each processor is characterized by some microarchitectural parameters like issue and decode width, number of integer and floating point ALUs, instruction and data cache sizes, RUUs, LSQs etc. A task in a streaming application may run more efficiently one setting of these microarchitectural parameters and less for another setting. Further, the designer of a SoC system can choose to run the processors at different frequencies in order to save power and at the same time meet the execution requirements of the application. This gives rise to a design space which has a size n^T, if T is the number of tasks and n the number of different candidate architectures on which they can be mapped to. For the sake of generality, we define each candidate architecture by using the microarchitectural parameters mentioned above, rather than using names of the processors. It should be noted that, almost all commercial processors can be represented using these microarchitectural parameters. For example, Appendix A shows the parameters that describe a processor similar to PowerPC604 (source: simscal [30]). None of the previous works [8, 7, 31] in this direction have considered the microarchitectural details. In Section 5.5 we describe how these microarchitectural properties help us in accurately estimating the costs associated with each candidate architecture.

The goal of this work is to propose a design space exploration method which finds the optimal mapping of T tasks onto any of the n candidate architectures in the architecture library. Potentially there could be more than one optimality criteria, each optimality criteria being one of the objective functions of our multi-objective framework. One of the outcomes of our DSE is a Pareto-Optimal front of solutions, which is nothing but the set of all design points which are not *dominated* by any

other design point. The formal definition of Pareto-optimality is as follows:

If C is a set of all cost vectors $c \in R^k$ where is k is the number of objectives, then a cost vector $c \in C$ dominates $d \in C$, if $c_i \leq d_i,\ \forall\ i \in (0, k-1)$ and $c_j < d_j$ for at least one $j \in (0, k-1)$. A cost vector is Pareto-Optimal if it is not dominated by any other cost vector.

The pareto-optimal front allows a system designer to identify trade-offs between different design objectives like cost, power, on-chip buffer size etc. It should be noted that to be a candidate for inclusion in the pareto-optimal front, the solution has to first satisfy the timing requirements of the application.

The target SoC platforms are assumed to consist of a number of processing elements (PEs) connected by a point-to-point communication network (e.g. RAW [22]) so that the communication conflicts are minimized. The PEs are assumed to have local memories that can fit the input FIFO buffers for the tasks mapped to them thus memory access conflicts are minimized. Thus the mapping problem of tasks to processing units can be addressed independently of the design of communication or memory architecture. Extensive research [12, 24, 14, 11] in the the direction of communication and memory architecture for SoCs shows that these assumptions are safe. They are also in keeping with the latest trends in SoC design methodology which divide the system design in two phases: 1)Mapping of tasks to processing units, and, 2) design of memory and communication architecture, of which we address the first in this work.

In summary, our problem setup consists of an *application* modeled as a Process Network along with the desired output rate C for that application, and *Architecture Library* of all types of processors available for mapping along with the cost of each type of processor. It should be noted that the service curves σ_{ij} for each ($Task_i$, $Processor_j$) pair are available in advance through initial simulation of tasks on processors. Each *candidate solution* is a n-tuple of the form $(M_1, M_2, ..., M_T)$, where the value of each mapping variable M_i points to a candidate architecture in the architecture library, clearly $1 \leq M_i \leq n,\ \forall\ i$. Each candidate solution refers to a leaf node in the design space. We refer to intermediate nodes (non-leaf) as partial solutions. A *partial solution* is essentially a n-tuple of the form $(M_1, M_2, ..., M_T)$ with $1 \leq M_i \leq n,\ \forall\ i \leq t$, for some $t < T$ and $M_i = 0,\ \forall\ i > t$.

VLSI technology dependent factors such as total maximum silicon area (C_{max}) available for system implementation and total maximum power (P_{max}) available to the system, are taken as external

inputs to our framework. We use C_{max} and P_{max} as conditions for pruning certain portions of the design space. Every time we go one step deeper in the design space, we apply these conditions to the partial solution. If the cost of the partial solution exceeds C_{max} or the total power consumption exceeds P_{max} then the search along that path is abandoned. These upper bounds can be used to restrict the search within practical limits of realizing the MPSoC.

4.2 Analytical Framework

Using the result 3.1 recursively we compute α_p, the rate of arrival of data items at the playout buffer, as:

$$\alpha_p(\Delta) = ((...(\alpha_1 \otimes \sigma_1) \otimes \sigma_2)... \otimes \sigma_T)(\Delta) \tag{4.1}$$

The timing constraint for successful execution of this application can be specified as:

$$\alpha_p(\Delta) \geq C.\Delta, \ \forall \ \Delta > 0 \tag{4.2}$$

We define:

- Infeasible Solution: A mapping for which above timing constraint is not satisfied.
- Feasible Solution: A mapping for which above timing constraint is satisfied.
- Optimal Solution: A Feasible solution which has the least cost.

In the above discussion, the service curves depend on the mapping of a task on a processing unit. For example, σ_{11} (in case T_1 is mapped to processor type 1) is not the same as σ_{12} (in case T_1 is mapped to processor type 2). We use binary decision variables x_{ij}, which is true when i^{th} task is mapped to j^{th} processor type. We write the above equations as :

$$\begin{aligned}\alpha_{i+1}(\Delta) &= (\alpha_i \otimes \sigma_{i1})(\Delta).x_{i1} + ... + (\alpha_i \otimes \sigma_{in})(\Delta).x_{in} \\ &= (\alpha_i \otimes (\sigma_{i1}.x_{i1} + ... + \sigma_{in}.x_{in}))(\Delta)\end{aligned}$$

And the rate of arrival of data items in the playout buffer can be expressed as:

$$\begin{aligned}\alpha_p(\Delta) =&((...(\alpha_1 \otimes (\sigma_{11}.x_{11} + \sigma_{12}.x_{12} + ... + \sigma_{1n}.x_{1n}) \\ &\otimes (\sigma_{21}.x_{21} + \sigma_{22}.x_{22} + ... + \sigma_{2n}.x_{2n})) \\ &... \\ &\otimes (\sigma_{t1}.x_{t1} + \sigma_{t2}.x_{t2} + ... + \sigma_{tn}.x_{tn}))(\Delta) \\ &\geq C(\Delta),\ \forall\ \Delta > 0\end{aligned}$$

This serves as a feasibility condition for solutions. Any design space exploration method can use this condition to evaluate candidate solutions from the design space. We discuss the advantages and disadvantages of several search techniques and propose our depth-first-search algorithm.

- Evolutionary Algorithms: They have mainly two parts, a mechanism for choosing potential solutions from the entire population of solutions, and a mechanism that evaluates the 'fitness' of these potential solutions. The feasibility condition mentioned above can serve as the mechanism to evaluate the fitness of a design point. However the mechanism for choosing the potential solutions is ad-hoc and may not lead to the optimal solution.

- ILP: There have been attempts to come up with an ILP formulation for this mapping problem. However the ILP formulation is made possible due to the assumption that the tasks have fixed execution requirement. Also they assume a regular memory access pattern. However the execution requirement for most multimedia applications has a lot of variability. The above feasibility condition accounts for variability through the use of variability characterization curves such as arrival curves and service curves. Due to the non-linear nature of the variability curves, an ILP formulation becomes impossible.

- Proposed Algorithm (DepthFirstDSE): The design space can be represented as a tree of height T (number of tasks), with n (number of different processing units) child nodes per node except the leaf nodes. Every level in the tree represents a mapping decision for a task. Due to the sequential flow of data items through the application, the mapping of a task determines the arrival curve of data items for the next task which in turn decides the mapping for that task and so on. Thus a depth-first approach for searching this design space is the most suitable.

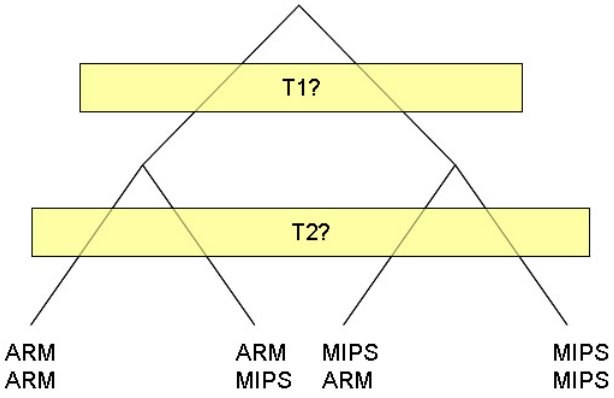


Fig. 4.1: Example Design Space

For example, using $T = 2, n = 2$ in the feasibility condition above and re-arranging the terms, we get:

$$\begin{aligned}\alpha_p(\Delta) = ((\alpha_1 \otimes \sigma_{11} \otimes \sigma_{21}).x_{11}.x_{21} + \\ (\alpha_1 \otimes \sigma_{11} \otimes \sigma_{22}).x_{11}.x_{22} + \\ (\alpha_1 \otimes \sigma_{12} \otimes \sigma_{21}).x_{12}.x_{21} + \\ (\alpha_1 \otimes \sigma_{12} \otimes \sigma_{22}).x_{12}.x_{22})(\Delta) \\ \geq C(\Delta),\ \forall\ \Delta > 0\end{aligned}$$

It can be seen that α_p is made of 4 terms, each corresponding to a path in the tree which also represents a solution in the search space. It is easy to see that, in case of T tasks and n processor types, α_p is made of n^T terms, each corresponding to a unique path in the design space and representing a possible mapping instance. It should be noted that, mapping for a task decides the Arrival Curve of data items for the next task. This indicates the suitability of our depth-first approach for design space exploration.

4.3 Estimating buffer size requirements

Using the results from theory of network calculus, the minimum buffer size required to prevent the input channel of T_i from overflowing can be calculated as:

$$B_i = sup_{\Delta \geq 0}\{\alpha_i(\Delta) - \alpha_{i+1}(\Delta)\} \tag{4.3}$$

where α_i is the arrival rate of streaming objects at the input channel of T_i and it depends on which processor the task is mapped to. And the minimum playout buffer size required to prevent overflow can be calculated as:

$$B_p = sup_{\Delta \geq 0}\{\alpha_p(\Delta) - C(\Delta)\} \tag{4.4}$$

Thus the total buffer cost of a solution is:

$$B = B_p + \sum_{1}^{T} B_i \tag{4.5}$$

This buffer size estimate depends on service rates for the tasks which depend on the type of processor the task is mapped to. This buffer size can used to measure the *goodness* of a mapping, less the buffer size better is the mapping. It is well known that SRAM based components such as caches etc. occupy significant portion of silicon chip in modern processors. In a MPSoC with many processing units on a single die of silicon, it becomes even more important to minimize the buffer sizes. Later in Section 5.5.1 we describe how estimates from Equation 4.5 are used as part of the overall silicon chip-space requirements of the system.

The actual chip area numbers (C_j's) can be obtained from the data sheets of the processors or can be derived from a well defined cost model. We have discussed some of these cost models in Section 5.5. The cost model for our second objective function: power consumption of the system is also discussed in the same section.

4.4 Conditions for pruning design space

As we explore the design space with a depth first approach, it is possible to eliminate some portions of the tree without having to go down to the leaf nodes. We introduced some conditions at each level in the design tree. If these conditions are satisfied at any node in the tree then part of the tree rooted at that node is eliminated. This may improve the running time of the algorithm significantly.

We used mainly two types of conditions at each level in the design space. The first is based on timing properties of the stream. We check if $\alpha(\Delta) < C.\Delta,\ \forall\ \Delta > 0$. This condition implies that the output rates of processors along this path of the design space are too weak to satisfy the overall

performance requirement of the application i.e $\alpha_p(\Delta) \geq C.\Delta, \ \forall \ \Delta > 0$. Thus all the solutions in the part of the design space rooted at this node can be safely eliminated in order to save running time.

The second type of conditions we used were based on certain technology dependent upper bounds on the overall costs of the solutions. Our multiobjective approach considers many types of costs like silicon chip area, power consumption etc. Each of these costs has a upper limit, for example, silicon requirements of the MPSoC are limited by the maximum number of transistors that can fit on a single die of silicon ($max\_silicon\_cost$) or availability of maximum power supply for the system (max_power_cost). At each level in the design tree we check if the costs of path explored so far are greater than the upper bounds and in that case we abandone the search along that path.

As will be discussed in the experimental results, these conditions cause significant improvement in the running time of the algorithm.

5. MPEG2 CASE STUDY

Our experimental setup is as shown in Figure 5.1. We used MPEG2 decoder as an example multimedia application for our experiments. We partitioned the application into 3 tasks: VLD, IQ, and IDCT+MC as the third task. Each task is simulated on different candidate architectures using SimpleScalar simulator and a simulation trace is obtained. For the experimental verification of our technique, we choose different microarchitectural configurations as the candidate architectures, described in more detail in Section 5.1. The *simulation trace* is essentially contains the the number of processor cycles required for each activation of the tasks. It should be noted that, although we need to simulate the tasks on each of the candidate architectures, this has to be done only once. Moreover in the context of industrial development of applications which happens in an incremental fashion with small changes to some of the tasks in the application, this simulation has to be carried out only on the modified tasks. Thus this technique will save valuable simulation time, as compared to other DSE techniques which require full system simulation.

Simulation traces are then passed on to the trace2sigma utility that we implemented which converts the traces into the service curves by counting the number of macroblocks decoded per unit of time. If service curves for different frequencies need to be obtained, then trace2sigma should be run many times with different frequency parameters.

Each candidate architecture is associated with two types of costs - silicon chip space required and the power consumed. These two act as the two objective functions of our multi objective optimization. We used a chip-space estimation model described in [30] to evaluate the first cost function and Wattch [2] to evaluate the second cost function. The designer's goal is to minimize both types of costs. However it may not always be possible to minimize both costs, which results in many trade-offs between the two costs. Our search algorithm keeps track of all these trade-offs and represents them in the form of a Pareto-optimal front. Section 5.5 describes the cost models in more details.

The service curves and the cost numbers from the cost functions are then passed on to the Depth-

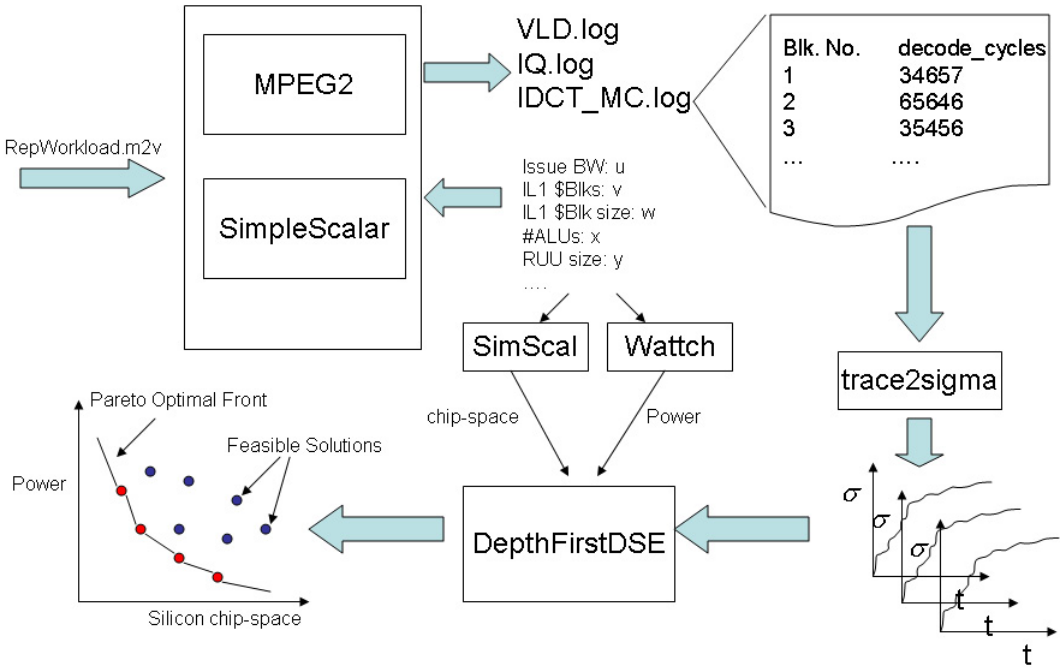


Fig. 5.1: Experimental Framework

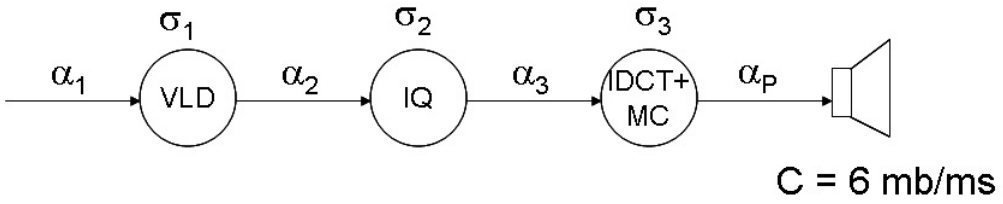


Fig. 5.2: MPEG2 Application Model

FirstDSE algorithm. This algorithm uses feasibility condition mentioned in Section 4.2 to ascertain if one of the n^T possible mappings of tasks on to the candidate architectures satisfies the performance requirements of the application (mpeg2 in this case). Once a mapping is found to satisfy the performance constraint, the costs associated with the mapping are computed by adding the costs of each individual candidate architecture which constitutes the mapping. Each mapping that satisfies the performance constraint is evaluated using multiple objective functions. The algorithm checks for the pareto-optimality of the mapping using the multiple objective functions. The final outcome of the algorithm is the pareto-optimal front which allows one to decide trade-offs between many types of costs associated with the MPSoC system.

All the necessary information about the performance requirements of the application is available in the form of the service curves. The system designer has to provide the information about the cost of each architecture in the architecture library and the desired output rate (C) at the playout buffer. All this information is then passed on to the DepthFirstDSE algorithm, which finds the least cost mapping for each task such that the application as a whole produces output at the desired rate C.

5.1 Candidate architectures

Having decided to use SimpleScalar for our simulations, which allows microarchitectural customizations through a configuration file, simulating different candidate architectures could be easily done by setting different microarchitectural parameters in the configuration file. Some of these parameters are listed below:

- *fetch:ifqsize <size>* - Instruction fetch queue size
- *decode:width <#instructions>* - Decode bandwidth
- *issue:width <#instructions>* - Issue bandwidth
- *ruu:size <size>* - size of the Register Update Unit
- *lsq:size <size>* - size of Load Store Queue
- *cache:il1 <#blks>:<blksize>:<assoc>:<repl_policy>* - Level 1 instruction cache configuration
  - <#blks> : Number of cache blocks
  - <blksize> : Cache block size (bytes)
  - <assoc> : Cache associativity
  - <repl_policy> : Replacement policy (e.g least recently used (LRU))

  [Similarly for il2, dl1, dl2 caches]
- *res:ialu <#ialu>* - number of integer ALUs
- *res:imult <#imul>* - number of integer multiplier/dividers

$Decode : IssueBandwidth$	$Il1Cacheblks$	$Il1blksize$	$IntALUs$	$FPALUs$	$RUUSize$	$LSQsize$
2:2	16	32	2	2	16	8
4:4	32	64	4	4	8	4

Tab. 5.1: Microarchitectural parameters and their different values

- *res:fpalu <#fpalu>* - number of floating point ALUs
- *res:fpmult <#fpmul>* - number of floating point multiplier/dividers

  ...

Many real processors can be emulated by setting appropriate values for the parameters in the configuration file. For example, Appendix A shows a configuration file which emulates PowerPC604 like processor.

From the microarchitectural parameters mentioned above, we chose some parameters which we feel are closely related to the performance of the task. We defined our candidate architectures using different combinations of values for the chosen set of parameters. Table 5.1 shows the parameters and their different values. Allowing two different values for each parameter results in 128 ($2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2$) different microarchitectural configurations. The possible values of parameters were chosen by looking at some existing processor architectures. For example, most superscalars have decode and issue width of 2 or 4 ([22]). Similarly we were conservative in choosing the size and number of I-cache blocks, due to small size of the application in question, the MPEG2 decoder, and the presence of regular loops in the code. We believe that choosing bigger I-cache sizes might result in almost all code being able to fit into the I-cache and if that happens there will not be significant performance difference between two configurations. Number of integer and floating point ALUs and RUU and LSQ sizes were also chosen after looking at configurations of some real processors (Appendix A). Detailed listing of configurations is attached in Appendix B. Appendix C shows an example of a SimpleScalar configuration file. Configuration files for all of the 128 candidates can be reproduced by using data from appendix B. We feel that this design space is big enough for validating the usefulness of our technique. It is worth mentioning here that in [31] only four architectures (ARM, PowerPC, μengine, DSP) were considered. Another work in this direction [7] also considers 4 arbitrary architectures, and does not even mention any of the microarchitectural details.

5.2 Initial Simulation

Any design technique that relies on simulation as a means of performance estimation, is faced with the problem of selecting *representative workload*, which is a well recognized problem in the domain of multiprocessor SoC design. Ideally, each implementation of an application on a MPSoC architecture has to be evaluated for a large number of possible inputs. However, this is an expensive process since the simulation involved for each input might require a considerable amount of time [32]. It is necessary for the designer to able to choose small *representative* set of inputs from a large library. A systematic solution for this is proposed in [20]. This work is based on the hypothesis that all the characteristics of multimedia streams that influence the performance of a MPSoC platform architecture, are related to their *variability*. Given a library of multimedia streams, they classify two streams as *similar* if both of them exhibit the same kind of variability with respect to execution time requirements and input/output rates as mentioned above. Once all similar workloads have been recognized, it is enough only to simulate one of these workloads. Following the rigorous workload selection process mentioned in this work is beyond the scope of this thesis, hence we use a sample video clip for our experiments. However it should be noted that the clip used is long enough and is representative of the behavior of all video clips as far as the worst case properties like service curves are concerned.

5.3 Obtaining service curves from simulation trace

Service curves of each candidate architecture are obtained using the trace2sigma utility that we implemented. It takes as input the simulation trace and produces the number of macroblocks processed per unit of time. Since trace2sigma allows us to specify the processor frequency, the service curves for different frequencies can be derived easily. For example, if frequency specified is 10 MHz then we count the number of macroblocks that are processed in first 10^7 processor cycles, next $2 * 10^7$ cycles and so on (assuming a second as the unit time, however if a ms is used as the unit then trace2sigma samples the trace at every 10^4 processor cycles). This gives us $y_i(t)$ as mentioned in Chapter 3. And the derivation of $\sigma(\Delta)$ is straightforward as described in the same chapter.

We assume that the output device consumes data from the playout buffer at 20 frames per second, which is the value of C in this case. According to MPEG2 standard, each macroblock is of size

16×16 pixels. For a MPEG2 video clip with resolution $W \times H$ pixels, the number of macroblocks per frame is $\frac{W \times H}{16 \times 16}$. Thus an output rate of 20 fps translates to $\frac{W \times H}{16 \times 16} \times 20$ macroblocks per sec. All our experiments were performed using this output rate.

5.4 Overview of Depth First DSE algorithm

Central to our DSE technique is a simple depth-first DSE algorithm that we implemented. Figure 5.3 shows the pseudo-code for this algorithm. This is a recursive depth-first search algorithm which takes as input a *PartialSolution* data structure, which stores the mapping of some or all of the tasks. This data structure is nothing but an array of size equal to the number of tasks. The value of each element in the array indicates the mapping for that tasks. For example, if $x[i] = j$ then i^{th} task is mapped to j^{th} candidate architecture. $x[i] = 0$ means mapping decision for i^{th} task has not been made. First call to DepthFirstDSE is made by the *main* function with $x[i] = 0 \forall i$, which means none of the tasks have been mapped.

DepthFirstDSE first checks if the $PartialSolution\ x$ is a leaf node in the design space (i.e. $x[i] = 0 \forall i$). If it is then it computes the α's at the input of each FIFO buffer by using the σ's corresponding to the candidate architectures chosen by the solution and applying the analytical results mentioned in Chapter 4. Once the α at the input of *playout buffer* is calculated, it checks if the application satisfies the real-time output constraint ($\alpha(\Delta) \geq C \times \Delta,\ \forall\ \Delta > 0$), where C is the rate at which real-time output device consumes streaming objects (macroblocks in this case) from the playout buffer. If this condition is satisfied, which means that this is a feasible solution, then the costs of this solution are computed. A solution is made of mapping of tasks to candidate architectures and each candidate architecture may have different types of costs associated with it. The costs which are specific to candidates are calculated by invoking the cost models for each of them and adding up the results. As described in Section 5.5.2 later, the on-chip buffers used for communication between processing units also contribute to certain costs such as silicon chip space requirements. The estimated buffer cost is computed by function *bufferCost()* and added to the overall silicon cost.

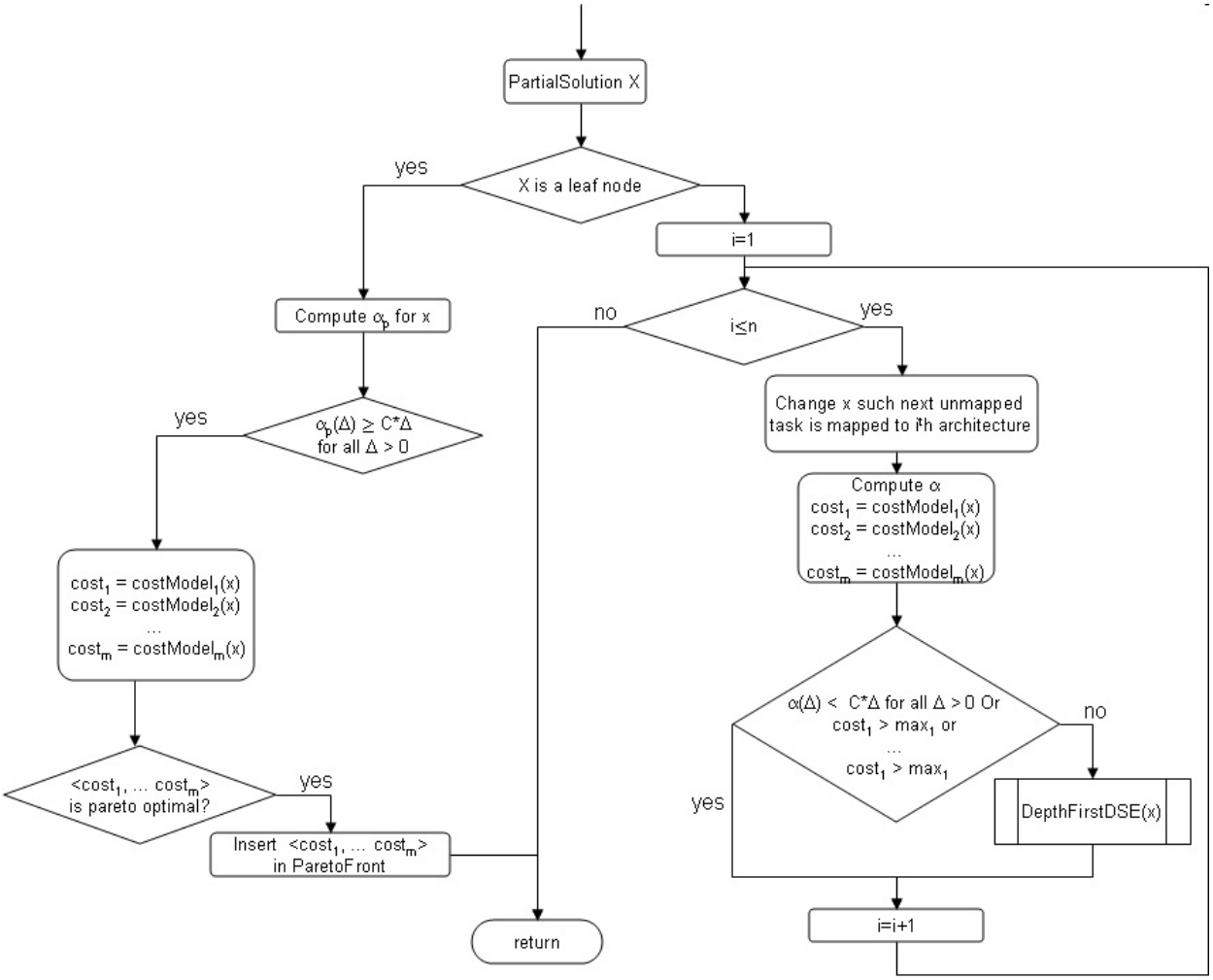
If the $PartialSolution\ x$ is not a leaf node in the design space (not all the tasks are mapped) then it tries to map the next unmapped task onto all possible candidate architectures. For every possible mapping of the next task, first it checks for a timing condition. The function $TimingCondition()$

```
Void DepthFirstDSE(PartialSolution x) {
  If x is a leaf node in the design space {
    // x[j] > 0 for all j
    Compute the arrival rate α at the playout buffer;
        If( α_p(Δ) >= C*Δ for all Δ > 0 ){ // solution satisfies performance constraint
          cost_1 = costModel_1 (x); // invokes costModel1 for x[1], ..., x[T] and returns sum
          ...
          cost_m = costModel_m (x); // invokes costModelm for x[1], ..., x[T] and returns sum
          // above costs depend on costs of individual candidate architectures
          // some other costs like buffer costs depend on mapping of two consecutive tasks
          cost_buff = bufferCost(x);
          // buffers contribute to silicon chip-space requirements
          // assuming cost_1 is silicon chip-space requirement of candidate architectures
          cost_1 += cost_buff;
          if(isParetoOptimal(cost_1, cost_2, ..., cost_m) )
          // checks if the cost vector is pareto-optimal
          // if yes, it is inserted to Pareto Front
                insertIntoParetoFront(x, cost_1, cost_2, ..., cost_m);
        }
  }
  else { // x is a partial solution; x[j] > 0 for 0<=j<=k; x[j] = 0 for k<j<=n
    for(i=1;i<=nArchitectures;i++) {
        update x such that next unmapped task is mapped to ith candidate architecture;
        if(TimingCondition(x))
          ; // this path will not lead to a solution that satisfies the output rate requirements
        else {
          cost_1 = costModel_1 (x); // invokes costModel1 for x[1], ..., x[k] and returns sum
          ...
          cost_m = costModel_m (x); // invokes costModelm for x[1], ..., x[k] and returns sum
          // since x is a partial solution, compute buffer costs for tasks which are mapped
          cost_buff = bufferCost(x);
          // assuming cost_1 is the silicon chip space
          cost_1 += cost_buff;
          if(cost_1 > max_1 OR cost_2 > max_2 ... OR cost_m > max_m) {
                ;  // Costs of solutions along this path exceed some technology dependent
                // upper bounds though these solutions satisfy timing requirements,
                // they can not be implemented using prevalent fabrication technology
          }
          else
                // call recursively with updated solution
                DepthFirstDSE(x);
        }
    }
  }
}
```

essentially implements the check $\alpha(\Delta) < C.\Delta,\ \forall\ \Delta > 0$ mentioned earlier in Section 4.4. Next, the costs of the $PartialSolution$ are computed and compared against some technology dependent upper limits, which is the second condition mentioned in Section 4.4. If any of these costs exceed these upper bounds then search along this path is abandoned. These upper bounds give the designer a way of eliminating solutions which are too costly to be implemented. It should be noted that these upper bounds are external inputs to this algorithm and represent some industrial cost constraints. For example, maximum number of transistors that can fit on a single die of silicon using a particular fabrication technology or maximum power usage allowed for the system. With these two conditions, timing condition and upper bounds on costs, exhaustive exploration of the design space can be avoided. The first condition helps us eliminate solutions that are not powerful enough to match the performance requirements of the application where as the second conditions eliminates solutions that are too costly. Figure 5.4 shows flowchart representation of the same algorithm. As the algorithm explores the design space it finds pareto-optimal solutions and adds them to the pareto optimal front. Figure 5.5 outlines the pseudo-code for two functions, first, $isParetoOptimal()$ which checks if a new solution is pareto-optimal and second, $insertIntoParetoFront()$ which actually inserts the solution into the pareto-front. $insertIntoParetoFront()$ has mainly two parts, first part simply adds the new solution to the existing set of pareto-optimal solution. The second and equally important part checks if there are any solutions in the set which are dominated by the newly added solution. If any such solution is found then it is removed from the set. This is necessary because a solution which is dominated by another solution can no longer be part of the pareto-optimal front.

5.5 Choice of objective functions

Some researchers have formulated the mapping of tasks to processors as an ILP [7] using silicon chip area as the single objective function. This single objective approach may not be adequate since system design for MPSoCs often involves many design objectives. The objectives could be conflicting such that optimizing on one objective compromises the other design objectives. Keeping this in mind we decided to take the multi-objective approach in this thesis. Researchers [31] working in the direction of DSE of MPSoCs have used different objectives like network performance and cost savings. Many more objectives that are critical to the system design are throughput, power usage of the



n	Number of candidate architectures
T	Number of tasks in the application
PartialSolution x	A data structure which stores the mapping of tasks to candidate architectures i.e. $1<=x[i]<=n, \forall\ i \in [0,j]$ for some $j<T$ and $x[i] = 0\ \forall\ i \in [j+1, T]$
Leaf Node	Refers to a complete solution i.e. $1<=x[i]<=n, \forall\ i \in [0,T]$

Fig. 5.4: Depth First DSE program flow

```
struct ParetoSolution {
      PartialSolution *x;
      int cost_1;
      int cost_2;
      …
      int cost_m;
      struct ParetoSolution *next;
};
ParetoFront is the set of pareto optimal solutions;
int isParetoOptimal( <cost_1, cost_2, … cost_m>) {
      ParetoSolution ps;

      If( ParetoFront is empty) // this means any solution is pareto optimal
          return 1;
      ps = first solution in ParetoFront;
      while(there are more solutions in ParetoFront){
            if( ps dominates <cost_1, cost_2, … cost_m>) {
                  //Dominated by some solution already in the pareto-optimal front
                  // this means input vector can not be pareto-optimal
                  return 0;
            }
            ps = next solution in ParetoFront;
      }
      return 1;
}

void insertPareto(PartialSolution x, <cost_1, cost_2, … cost_m>) {
      ParetoSolution ps1, ps2;
      ParetoSolution ps = new ParetoSolution(x, <cost_1, cost_2, … cost_m>);

      Add ps to ParetoFront;

      //New solution inserted to ParetoFront

      // following code removes all solutions in the ParetoFront which are dominated
      // by newly added solution
      // this is necessary to maintain the pareto-optimality of each solution in ParetoFront
      ps1 = first solution in ParetoFront;
      while(there are more solutions in Pareto Front){
            if(ps dominates ps1 ){
                  remove ps1 from ParetoFront;
            }
            ps1 = next solution in ParetoFront;
      }
}
```

Fig. 5.5: Pareto Optimal Front creation

system etc. In order to demonstrate our multi-objective approach we narrowed down to two objective functions: silicon chip-space requirements and the power usage of the system. It should be noted however that our technique in general can handle any number of objectives, subject to the availability of appropriate cost models for those objectives.

5.5.1 Cost model for Chip-space requirements

For one of the objective functions, silicon chip area, we must estimate the chip-space requirements of MPSoC including the processing elements and the communication elements such as the buffers. We considered many ways to estimate the chip-space requirements for processing elements, including information from datasheets of real processors. However it was found that many manufacturers do not mention this information, probably because the chip area varies depending on the fabrication technology used. For example, chip area required for a processor on a 0.5 micron technology is different from area required for the same processor on 0.18 micron technology. We also considered some empirical cost models like SimpleFit [22], which are based on chip areas of some commercial processors. By fitting a quadratic function to the data for real processors, this model defines the silicon cost as a function of issue width. However sophisticated, this model is not of much use because of following reasons: first, its based only on one microarchitectural parameter i.e issue width, and second, the possible values of issue width are limited to 1, 2, or 4 even for latest superscalars.

It should be noted that building a cost model which takes into consideration all the microarchitectural parameters mentioned in Section 5.1 is not a simple task. Among all the components that account for chip area, some are SRAM based and have a well known implementation. For example, data and instruction caches, register sets, etc. It is easy to estimate the chip area for these components using analytical models. Since the chip area for 1 bit of SRAM is known, total area can be easily computed by multiplying it with the memory size of the component and adding some fixed cost for the structural implementation. However, some other components such as functional modules (adders and multipliers etc.) are much complex in implementation. Chip area for these components can only be computed using empirical methods, i.e. by using transistor counts for some well known implementations and extrapolating using word width as a parameter. Naturally these estimations are not as accurate as the analytical models. Another set of components, like buses and arbiters, it is even more

difficult to estimate the chip area because it depends on the actual layout of the other components on the chip. The chip space requirements for these components can best be estimated as a percentage overhead after adding chip space for all other components.

Considering these difficulties a hardware complexity estimation model, SimScal [30] has been developed. SimScal is based on sim-outorder simulator of SimpleScalar. It takes as input SimpleScalar microarchitectural parameters like cache size, decode and issue width, number of integer and floating point ALUs, RUU and LSQ sizes, branch predictors and memory ports, and estimates the transistor count and the chip area required. To the best of our knowledge, this is the only chip space estimator which takes into account such a wide range of microarchitectural parameters and is proved to be fairly accurate.

We estimated the silicon chip-space for each of our candidate architectures using SimScal estimation model. Appendix B summarizes the microarchitectural parameters that define our candidate architectures and Table 5.2 shows the corresponding silicon cost measured in transistor counts.

5.5.2 Chip-space estimation for on-chip buffers

On-chip buffer also contribute significantly to silicon chip-space requirements of an MPSoC. In the context of streaming applications these buffers are first-in-first-out buffers with one processor reading from it and another writing to it. Analytical model for estimating transistor counts of such FIFO buffers can be easily constructed using an approach similar to SimScal's analytical model for I-cache transistor count estimation. SimScal assumes four transistors per SRAM bit of memory ($TransSRBit$ = 4), two transistors per write port ($TransWP$ = 2) and one transistor per read port ($TransRP$ = 1). Assuming one read and one write port per buffer bit, the transistor count per buffer bit can be calculated as: $TransBuffBit = TransSRBit + TransWP + TransRP$.

Buffer size calculated by Equation 4.5 is measured in terms of numbers of data items, for example macro-blocks in case of MPEG2 decoder. The size of MPEG2 pixel is 16×16 pixels. Each macro-block occupies $16 \times 16 \times ColorDepth$ bits of buffer space, where $ColorDepth$ is the number of bits required to represent each pixel. Using this data, the total transistor count for on-chip buffers can be estimated as: $TransBuff = TransBuffBit \times B \times 16 \times 16 \times ColorDepth$, where B is the buffer size calculated by Equation 4.5. Here all the terms except B are constants. The proposed algorithm

$no.$	$Trans.count$	$no.$	$Trans.count$	$no.$	$Trans.count$	$no.$	$Trans.count$
1	2345125	33	2307957	65	2337741	97	2300677
2	2551155	34	2504115	66	2543203	98	2496275
3	2373685	35	2336517	67	2366301	99	2329237
4	2579715	36	2532675	68	2571763	100	2524835
5	2376485	37	2339317	69	2369101	101	2332037
6	2582515	38	2535475	70	2574563	102	2527635
7	2433605	39	2396437	71	2426221	103	2389157
8	2639635	40	2592595	72	2631683	104	2584755
9	2484363	41	2439483	73	2475843	105	2431083
10	2690521	42	2635705	74	2681433	106	2626745
11	2512923	43	2468043	75	2504403	107	2459643
12	2719081	44	2664265	76	2709993	108	2655305
13	2515723	45	2470843	77	2507203	109	2462443
14	2721881	46	2667065	78	2712793	110	2658105
15	2572843	47	2527963	79	2564323	111	2519563
16	2779001	48	2724185	80	2769913	112	2715225
17	2513758	49	2468878	81	2505238	113	2460478
18	2719916	50	2665100	82	2710828	114	2656140
19	2542318	51	2497438	83	2533798	115	2489038
20	2748476	52	2693660	84	2739388	116	2684700
21	2545118	53	2500238	85	2536598	117	2491838
22	2751276	54	2696460	86	2742188	118	2687500
23	2602238	55	2557358	87	2593718	119	2548958
24	2808396	56	2753580	88	2799308	120	2744620
25	2652516	57	2600164	89	2642860	121	2590644
26	2858674	58	2796386	90	2848450	122	2786306
27	2681076	59	2628724	91	2671420	123	2619204
28	2887234	60	2824946	92	2877010	124	2814866
29	2683876	61	2631524	93	2674220	125	2622004
30	2890034	62	2827746	94	2879810	126	2817666
31	2740996	63	2688644	95	2731340	127	2679124
32	2947154	64	2884866	96	2936930	128	2874786

Tab. 5.2: Transistor counts of different candidate architectures

treats $TransBuffBit \times 16 \times 16 \times ColorDepth$ as a scaling factor and uses it to compute the transistor counts for different buffer sizes corresponding to different mappings of tasks to processing units. This transistor count is then added to the sum of transistor counts for each candidate architecture that constitutes a mapping to arrive at the overall transistor count for a mapping.

5.5.3 Estimating Power consumption of candidate architectures

Accurately estimating the power requirements of candidate architectures is necessary to able to differentiate and choose between them. We relied on Wattch [2] power simulator for these power estimates. Wattch is composed of different parameterizable power models for different types of components that make up the processor. It uses per cycle resource usage counts from a cycle-level simulator like SimpleScalar and invokes power models for the resources accessed in each cycle. It generates detailed statistics on power usage by each component as well as the overall power usage.

We used Wattch to find the power consumption by each of the 128 candidate architectures. Among the outputs generated by Wattch, we used the output avg_total_power_insn_cc3 to evaluate the power cost associated with a candidate architecture. This output parameter represents the average power per instruction which is a good measure of the power hungriness of a processor. Table 5.3 summarizes the power costs in units of Watts (Joules/Sec). We assume that the power consumed by the MPSoC is equal to addition of power consumed by each candidate separately. Wattch itself takes a similar approach, as it estimates the power consumed by each component (like ALU, buses etc.) separately and adds them to find total power and it has been demonstrated to be reasonably accurate. Thus in the context of MPSoC system with many processors on a die of silicon, the total power can be approximated by adding power estimates for individual processors. Of course, on-chip communication network also consumes some amount of power and may contribute to some inaccuracy in power estimates. However our goal here is to be able to compare different possible combinations of processors using power as the criteria. Thus the inaccuracies due to on-chip communication network may be assumed to cancel out each other in the comparison. With this second objective in place, the designer can choose design points with low power consumption as well as can identify trade-offs between the two objectives, the silicon chip-space and power requirements.

$no.$	$avg.power/insn$	$no.$	$avg.power/insn$	$no.$	$avg.power/insn$	$no.$	$avg.power/insn$
1	11.97911	33	12.99732	65	14.32321	97	10.917
2	12.65286	34	13.69015	66	12.12976	98	11.23352
3	13.10566	35	10.40271	67	12.35679	99	14.22612
4	10.52442	36	10.74853	68	13.13113	100	15.12567
5	10.70429	37	14.95595	69	13.57614	101	12.65724
6	13.82946	38	15.45465	70	10.69141	102	13.09034
7	14.55188	39	12.93051	71	10.88843	103	13.5684
8	12.25111	40	13.2075	72	15.13805	104	14.3896
9	12.57125	41	13.19031	73	12.04725	105	11.2198
10	13.12747	42	14.11305	74	15.42152	106	10.12687
11	14.57249	43	14.60719	75	13.15984	107	11.59544
12	13.60451	44	11.10565	76	13.26504	108	15.52379
13	10.8265	45	11.22023	77	14.31185	109	15.88443
14	11.06283	46	15.50096	78	14.61354	110	13.70145
15	15.13756	47	16.14119	79	11.48042	111	13.80251
16	15.35552	48	13.33628	80	11.65808	112	14.75102
17	13.30246	49	13.65861	81	15.70674	113	15.24726
18	13.22945	50	14.59685	82	15.98366	114	12.03709
19	14.31436	51	13.89681	83	13.58371	115	12.19327
20	14.59844	52	15.06411	84	12.54377	116	16.09167
21	11.64969	53	11.41189	85	13.59999	117	10.42491
22	11.97501	54	11.54705	86	14.81504	118	16.79948
23	11.67012	55	13.29828	87	15.05845	119	14.13889
24	15.70942	56	13.77497	88	11.80866	120	14.35498
25	16.25589	57	11.73639	89	11.83269	121	15.25696
26	13.73902	58	12.02966	90	13.69041	122	15.90206
27	13.85732	59	12.65886	91	14.25541	123	12.37245
28	14.8207	60	13.13942	92	12.25163	124	12.55795
29	15.11534	61	10.39438	93	12.55374	125	13.28925
30	11.98386	62	11.5992	94	13.09474	126	13.70432
31	12.04513	63	10.71625	95	13.21816	127	11.84595
32	12.5092	64	13.83612	96	13.75536	128	13.72209

Tab. 5.3: Power costs of different candidate architectures

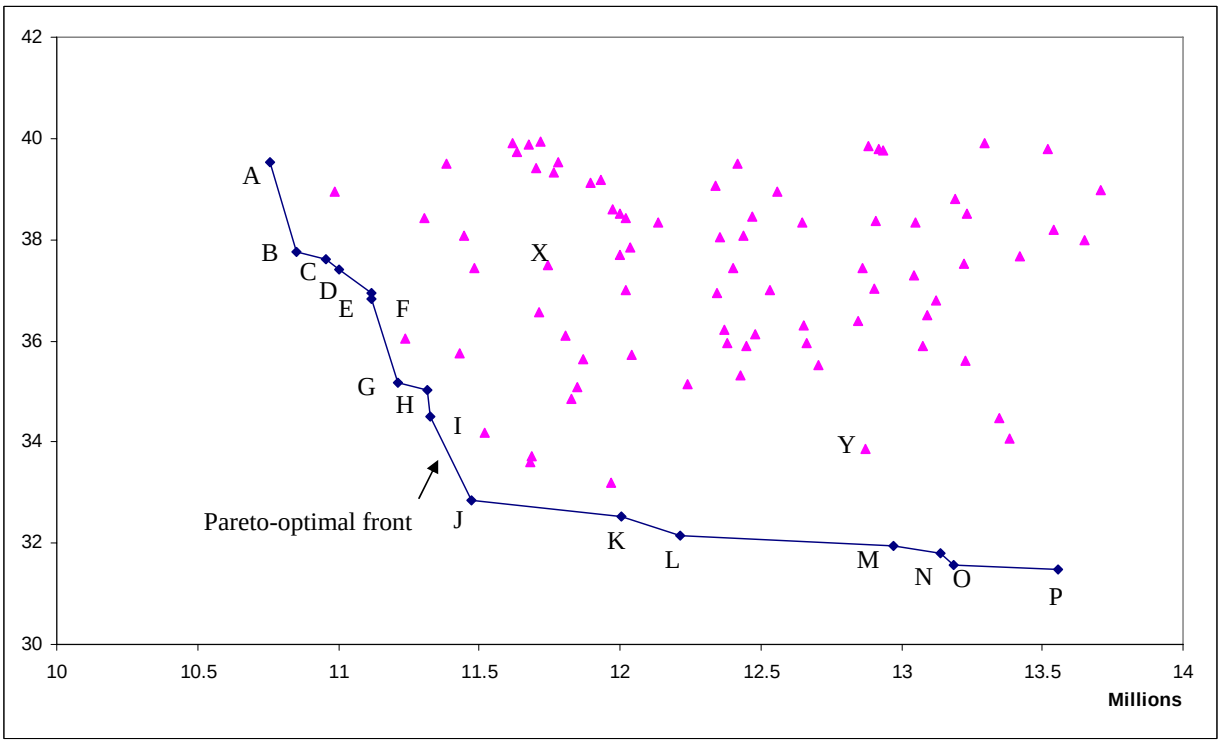


Fig. 5.6: Points connected by line represent the Pareto Optimal front. Other points shown by triangles are other feasible solutions dominated by some solution on the pareto-optimal front. Silicon costs are on horizontal axis and power costs on vertical axis.

5.6 Experimental Results and analysis

The pareto-optimal front generated by a particular run of our algorithm is shown in Figure 5.6. Standard output rate for MPEG2 application is around 20 fps. This translates to around 6 macroblocks/ms for the sample video clip which we used for experiments. The $ColorDepth$ for the video clip was 32. Thus using the analytical model mentioned in Section 5.5.2 the silicon cost of on-chip buffers per macroblock comes to 57344. This scaling factor along with Equation 4.5 was used by the depthFirstDSE algorithm to estimate the silicon costs of the on-chip buffers.

Overall running time of the algorithm was about 18 minutes on Ultra Sparc III CPU (750MHz) running SUN OS 5.8. The design space was explored in a depth first manner with the maximum depth being 3 which equals the number of tasks in the application. The $TimingCondition$ was not satisfied at 22 nodes at depth 1 and 9587 points at depth 2. Each node at depth 1 has 128×128 leaf nodes as its

	$Timing\ Condition$	$Upper\ Bound\ Conditions$
$Nodes\ pruned\ at\ level\ 1$	22	0
$Nodes\ pruned\ at\ level\ 2$	9587	17
$\%\ of\ design\ space\ pruned$	75.7	0.1

Tab. 5.4: Performance statistics for MPEG2 case study

descendants since each node has 128 child nodes. Similarly, each node at depth 2 has 128 leaf nodes as its descendants. Thus the 22 nodes at depth 1 and 9587 nodes at depth 2 represent 75.7% of the design space which is pruned by the $TimingCondition$. The second condition which checks certain upper bounds on the costs contributed to only 0.1% of the pruning since it was not satisfied at only 17 nodes at depth 2 in the design space. However, these upper bounds are external inputs to the algorithm and are either technology dependent or represent some industrial standards, and for the experimental validation we had chosen pessimistic upper bounds ($max\_silicon\_cost = 15\ million\ transistors$ and $max\_power\_cost = 40W$). Nevertheless they demonstrate how search space can be restricted to solutions which are practically feasible to be implemented. In summary, about 76% of the design space was pruned by timing and upper bound conditions. The running time of the same algorithm without these conditions was around 69 minutes which indicates 74% time savings which is almost proportionate with the pruned design space. In figure 5.6 points A to P represent the pareto-optimal front. The points shown by triangles are solutions which are dominated by some point on the pareto-optimal front. For example, it can be seen very easily that solution X is dominated by solution G and solution Y is dominated by solution L. Solution A, which is on pareto-optimal front does not dominate X and O does not dominate Y. However A and O are on the pareto-optimal front since they are not dominated by any other solution (see definition of pareto-optimality in Section 4.1). The pareto front in figure 5.6 can be broadly divided into three segments, solutions A to F in first segment, solutions G to L making the middle segment and M to P constituting the last segment. The average silicon cost of a solution in third segment is 17% higher than that of the other segments whereas solutions in first segment are on average 14.5% more expensive in power as compared to the rest of the solutions. A designer having ample silicon resources can look at third segment for a potential solutions. The first segment can be looked at if the designer can afford higher power consumption. The solutions in middle segment are on average 14% cheaper in terms of silicon costs and 6% more

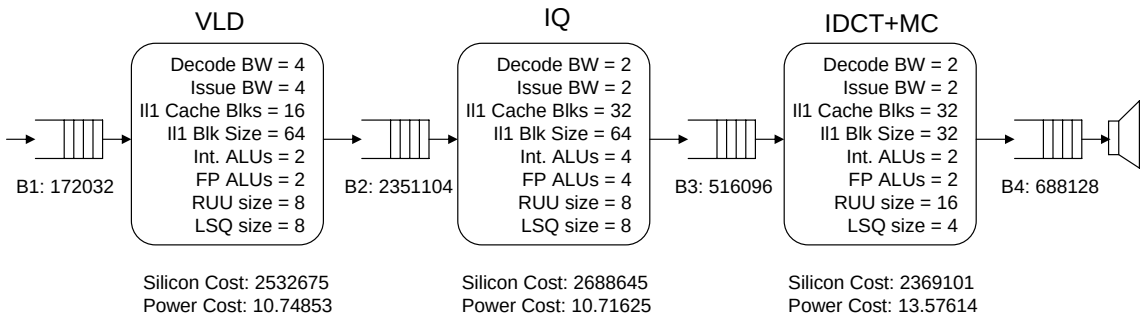


Fig. 5.7: Pareto Optimal Solution H

expensive in power costs as compared to solutions in third segment. A designer can trade-off extra power resources for reduction in silicon costs by switching from segment three to the middle segment. Similar trade-offs can be made between the first two segments since middle segment is 12% cheaper in terms of power costs and 5.5% more expensive in terms of silicon costs as compared to solutions in first segment.

Once a broader segment of the pareto-front has been identified, the solutions could be looked more closely for their microarchitectural details and on-chip buffer requirements. For example, figures 5.7 and 5.8 show the details of solutions H and K respectively. Both solutions choose the same microarchitectural configuration for task IQ but choose different configurations for VLD and IDCT+MC. Solution H chooses a higher decode/issue BW for task VLD as compared to the same by solution K. On the other hand K assigns bigger level 1 I-cache size and RUU size for the same task. It can be seen that though solution K is more expensive in terms of overall silicon costs, it is actually cheaper in silicon costs of only processing units. This could be explained by higher contribution of buffers to the silicon cost (38.2%) of solution K as opposed to only 33% in case of solution H. Appendix F lists other pareto optimal solutions. This analysis clearly suggests that the pareto-optimal front and detailed description of pareto-optimal solutions give us a lot of insight into various aspects of system design and trade-offs between multiple conflicting objectives.

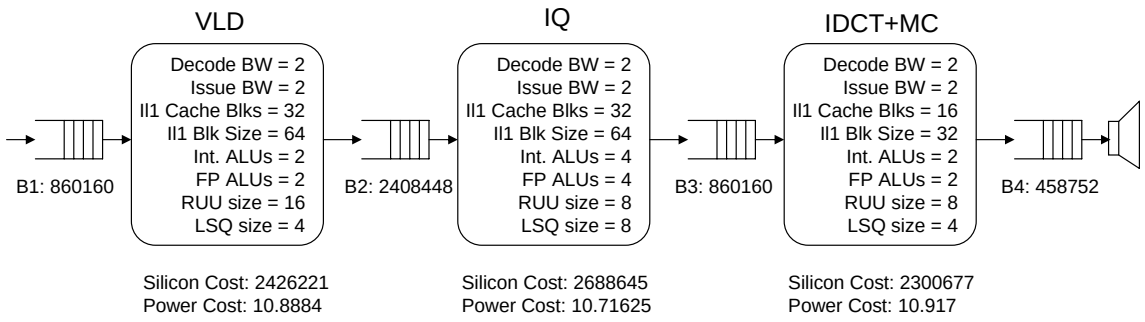


Fig. 5.8: Pareto Optimal Solution K

6. CONCLUSIONS AND FUTURE WORK

In this work we proposed an innovative design space exploration technique for mapping of streaming applications to multiprocessor SoC platforms. All previous work in this direction can be categorized into two parts: 1)Evolutionary techniques that use full/symbolic system simulation, which have the obvious advantage of being accurate in performance estimation, and 2) ILP based techniques which have the advantage that they guarantee optimality of the solutions. Our technique essentially combines the advantages of these two extremes without incurring any of the disadvantages. Evolutionary techniques are heuristic based and can not guarantee optimality. Unlike that, our technique explores the design space in a systematic depth-first manner and prunes parts of the design space only when none of the solutions in that part of the tree can satisfy real-time performance requirements of the application or exceed technology dependent upper bounds. For the MPEG2 case study that we performed, these conditions result in 76% pruning of design space and 74% time savings. Our technique avoids the drawbacks of ILP techniques which assume fixed performance requirements and support only one objective function. Performance estimation technique we used is fairly accurate since we simulate each task on each candidate architecture and model the variability in execution requirements using theory of network calculus which is broadly used in networking domain for performance estimation and modeling.

An important feature of our DSE technique is that it obviates the need for system level simulation which is very time consuming. Instead the process of simulation is broken down to the task level. This gives the designer flexibility of modifying a single task without affecting the entire system. Commercially the applications are developed in incremental fashion, with small changes at a time. Thus our method could prove very useful in real world SoC development because there is no need for a system level simulation every time a small change is performed.

The most useful outcome of the technique we presented is the multi-objective pareto-optimal front. The pareto-optimal solutions represent trade-offs between various design objectives of MP-

SoC design. We demonstrated this technique using two objectives: minimization of silicon costs and minimization of power consumption of the overall application. The pareto-front generated by our depth-first design space exploration algorithm could be used to identify solutions with minimum silicon or power costs. By using the pareto-front, extra silicon resources could be traded for lesser power consumption or excess power availability could be traded for reduction in silicon costs in a systematic way. Other important outcomes of the proposed technique are detailed microarchitectural descriptions of the pareto-optimal solutions as well as minimum buffer sizes required to prevent overflows, which are necessary inputs for implementation of an MPSoC. Though we used only two objectives in our case study of MPEG2 application, more objectives can easily be added provided appropriate cost models for those objectives are available. The depth first DSE algorithm outlined in this work can be easily extended to any other multimedia streaming application. We conclude that the proposed technique is useful for synthesis of multiprocessor architectures for multimedia applications.

Future Work: Although our DSE technique is innovative and efficient, we feel that simulation technique that we use is tedious. For example, the MPEG2 application code that we use is monolithic piece of code written keeping uniprocessor in mind. This makes the process of simulation of individual tasks cumbersome because we have to manually split the application into tasks. If the applications are implemented in a stream programming language then it will greatly simplify the process of initial simulation because these stream programming languages have a clear division between the tasks in the application. In this context we would like to mention that work is already in progress at MIT on implementation of MPEG2 decoder in a stream programming language called StreamIt [9].

BIBLIOGRAPHY

[1] J.-Y. L. Boudec and P. Thiran. *Network Calculus - A Theory of Deterministic Queuing Systems for the Internet*. Lecture Notes in CS, Springer, 2001.

[2] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, pages 83–94. ACM Press, 2000.

[3] J.-M. Chang and M. Pedram. Codex-dp: Co-design of communicating systems using dynamic programming. *IEEE Transactions on CAD*, pages 732–744, July 2000.

[4] B. P. Dave, G. Lakshminarayana, and N. K. Jha. Cosyn: Hardware software cosynthesis of heterogeneous distributed embedded systems. *IEEE Transactions on VLSI Systems*, pages 92–104, 1999.

[5] E. de Kock. Multiprocessor mapping of process networks: a jpeg decoding case study. In *15th international symposium on System Synthesis*, pages 68–73. ACM Press, 2002.

[6] E. A. de Kock. Yapi: Application modelling for signal processing systems. In *37th Design Automation Conference*, pages 402–405. IEEE, 2000.

[7] B. Dwivedi, A. Kumar, and M. Balakrishnan. Synthesis of application specific multiprocessor architectures for process networks. In *17th International Conference on VLSI Design*, page 780. IEEE Computer Society, 2004.

[8] B. K. Dwivedi, A. Kumar, and M. Balakrishnan. Automatic synthesis of system on chip multiprocessor architectures for process networks. In *2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 60–65. ACM Press, 2004.

[9] M. I. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, A. A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe. A stream compiler for communication-exposed architectures. In *10th international conference on Architectural support for programming languages and operating systems*, pages 291–303. ACM Press, 2002.

[10] G. Kahn. Semantics of a simple language for parallel programming. *Proc. of Information Processing*, August 5-10:471–475, 1974.

[11] S. Kim, C. Im, and S. Ha. Schedule-aware performance estimation of communication architecture for efficient design space exploration. In *1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 195–200. ACM Press, 2003.

[12] S. Kim, C. Im, and S. Ha. Efficient exploration of on-chip bus architectures and memory allocation. In *2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 248–253. ACM Press, 2004.

[13] Y.-K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Comput. Surv.*, 31(4):406–471, 1999.

[14] K. Lahiri, A. Raghunathan, and S. Dey. Efficient exploration of the soc communication architecture design space. In *IEEE/ACM international conference on Computer-aided design*, pages 424–430. IEEE Press, 2000.

[15] J. Leijten, J. van Meerbergen, A. Timmer, and J. Jess. Prophid: a data-driven multi-processor architecture for high-performance dsp. In *European conference on Design and Test*, page 611. IEEE Computer Society, 1997.

[16] J. Leijten, J. van Meerbergen, A. Timmer, and J. Jess. Prophid: a heterogeneous multi-processor architecture for multimedia. In *International Conference on Computer Design*, page 164. IEEE Computer Society, 1997.

[17] P. Lieverse, T. Stefanov, P. van der Wolf, and E. Deprettere. System level design with spade: an m-jpeg case study. In *IEEE/ACM international conference on Computer-aided design*, pages 31–38. IEEE Press, 2001.

[18] J. Luo and N. K. Jha. Low power distributed embedded systems: Dynamic voltage scaling and synthesis. In *International Conference on High Performance Computing (HiPC)*, December 2002.

[19] D. Lyonnard, S. Yoo, A. Baghdadi, and A. A. Jerraya. Automatic generation of application-specific architectures for heterogeneous multiprocessor system-on-chip. In *38th Design Automation Conference*, pages 518–523. ACM Press, 2001.

[20] A. Maxiaguine, Y. Liu, S. Chakraborty, and W. T. Ooi. Identifying "representative" workloads in designing mpsoc platforms for media processing. *2nd IEEE Workshop on Embedded Systems for Real-Time Multimedia*, 2004.

[21] A. Maxiaguine, Y. Zhu, S. Chakraborty, and W.-F. Wong. Tuning soc platforms for multimedia processing: identifying limits and tradeoffs. In *2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 128–133. ACM Press, 2004.

[22] C. A. Moritz, D. Yeung, and A. Agarwal. Simplefit: A framework for analyzing design tradeoffs in raw architectures. *IEEE Transactions on Parallel Distributed Systems*, 12(7):730–742, 2001.

[23] T. Parks. *Bounded scheduling of process networks*. PhD thesis, University of California, EECS Dept., Berkeley, CA, 1995.

[24] S. Pasricha, N. Dutt, and M. Ben-Romdhane. Fast exploration of bus-based on-chip communication architectures. In *2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 242–247. ACM Press, 2004.

[25] A. D. Pimentel, L. O. Hertzberger, P. Lieverse, P. van der Wolf, and E. F. Deprettere. Exploring embedded-systems architectures with artemis. *IEEE Computer*, 34(11):57–63, 2001.

[26] M. Rutten, J. van Eijndhoven, E. Jaspers, P. van der Wolf, O. Gangwal, and A. Timmer. A heterogeneous multiprocessor architecture for flexible media processing. *IEEE Design & Test of Computers*, 19(4):39–50, July-August 2002.

[27] M. Rutten, J. van Eijndhoven, and E.-J. Pol. Robust media processing in a flexible and cost-effective network of multi-tasking coprocessors. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2002.

[28] D. Sciuto, F. Salice, L. Pomante, and W. Fornaciari. Metrics for design space exploration of heterogeneous multiprocessor embedded systems. In *10th international symposium on Hardware/software codesign*, pages 55–60. ACM Press, 2002.

[29] D. Shin and J. Kim. Power-aware scheduling of conditional task graphs in real-time multiprocessor systems. In *International Symposium on Low Power Electronics and Design (ISLPED)*, August 2003.

[30] M. Steinhaus, R. Kolla, J. L. Larriba-Pey, T. Ungerer, and M. Valero. Transistor count and chip space estimation of simple-scalar-based microprocessor models. In *Workshop on Complexity-Effective Design*, 2001.

[31] L. Thiele, S. Chakraborty, M. Gries, and S. Kunzli. A framework for evaluating design tradeoffs in packet processing architectures. In *39th Design Automation Conference*, pages 880–885. ACM Press, 2002.

[32] G. V. Varatkar and R. Marculescu. On-chip traffic modeling and synthesis for mpeg-2 video applications. *IEEE Transactions on Very Large Scale Integrated Systems*, 12(1):108–119, 2004.

[33] V. Zivkovic, E.Deprettere, P. van der Wolf, and E. de Kock. Design space exploration of streaming multiprocessor architectures. In *SiPS*, 2002.

[34] V. D. Zivkovic, E. de Kock, P. van der Wolf, and E. Deprettere. Fast and accurate multiprocessor architecture exploration with symbolic programs. In *conference on Design, Automation and Test in Europe*, page 10656. IEEE Computer Society, 2003.

APPENDIX

A. CONFIG FILE PARAMETERS FOR PPC604 LIKE PROCESSOR (SOURCE:SIMSCAL)

Name	Value	Description
seed	0	Random number generator
fetch:ifqsize	4	Instruction fetch queue size
fetch:mplat	3	extra branch mis-perdiction latency
bpred	bimod	branch predictor type (nottaken|taken|perfect|bimod|2lev)
bpred:bimod	1024	bimodal predictor BTB size
bpred:2lev l1size	1	2-level predictor config l1size param
bpred:2lev l2size	1024	2-level predictor config l2size param
bpred:2lev hist_size	8	2-level predictor config hist_size param
decode:width	4	instruction decode B/W (insts/cycle)
issue:width	4	instruction issue B/W (insts/cycle)
issue:inorder	false	run pipeline with in-order issue
issue:wrongpath	true	issue instruction down wrong execution paths
ruu:size	32	register update unit (RUU) size
lsq:size	16	load/store queue (LSQ) size
cache:dl1	512	l1 data cache config
cache:dl1	32	l1 data cache config
cache:dl1	2	l1 data cache config
cache:dl1	l	l1 data cache config
cache:dl1lat	1	l1 data cache hit latency (in cycles)
cache:dl2	1024	l2 data cache config
cache:dl2	64	l2 data cache config
cache:dl2	4	l2 data cache config
cache:dl2	l	l2 data cache config
cache:dl2lat	6	l2 data cache hit latency (in cycles)
cache:il1	1024	l1 inst cache config
cache:il1	32	l1 inst cache config
cache:il1	1	l1 inst cache config
cache:il1	l	l1 inst cache config
cache:il1lat	1	l1 inst cache hit latency (in cycles)
cache:il2	1024	l2 inst cache config
cache:il2	64	l2 inst cache config
cache:il2	4	l2 inst cache config
cache:il2	l	l2 inst cache config
cache:il2lat	6	l2 inst cache hit latency (in cycles)
cache:flush	false	flush caches on system calls
cache:icompress	false	convert 64-bit inst address to 32-bit inst equivalence
mem:lat	18	memory latency first_chunk
mem:lat	2	memory latency inter_chunk
mem:width	8	memory access bus width (in byte)
tlb:itlb	64	instruction TLB config
tlb:itlb	4096	instruction TLB config
tlb:itlb	4	instruction TLB config
tlb:itlb	l	instruction TLB config
tlb:dtlb	64	data TLB config
tlb:dtlb	4096	data TLB config
tlb:dtlb	4	data TLB config
tlb:dtlb	l	data TLB config
tlb:lat	30	inst/data TLB miss latancy
res:ialu	4	total number of integer ALU´s available
res:imult	1	total number of integer multiplier/dividers available
res:memport	2	total number of memory system ports available (to CPU)
res:fpalu	4	total number of floating point ALU´s available
res:fpmult	1	total number of floating point multiplier/dividers available
bugcompat	false	operate in backward-compatible bugs mode (for testing only)

B. MICROARCHITECTURAL CONFIGURATIONS

$No.$	$DecodeBW$	$IssueBW$	$il1CacheBlks$	$il1BlkSize$	$IntALUs$	$FPALUs$	$RUUsize$	$LSQsize$
1	2	2	16	32	2	2	16	8
2	4	4	16	32	2	2	16	8
3	2	2	16	64	2	2	16	8
4	4	4	16	64	2	2	16	8
5	2	2	32	32	2	2	16	8
6	4	4	32	32	2	2	16	8
7	2	2	32	64	2	2	16	8
8	4	4	32	64	2	2	16	8
9	2	2	16	32	4	2	16	8
10	4	4	16	32	4	2	16	8
11	2	2	16	64	4	2	16	8
12	4	4	16	64	4	2	16	8
13	2	2	32	32	4	2	16	8
14	4	4	32	32	4	2	16	8
15	2	2	32	64	4	2	16	8
16	4	4	32	64	4	2	16	8
17	2	2	16	32	2	4	16	8
18	4	4	16	32	2	4	16	8
19	2	2	16	64	2	4	16	8
20	4	4	16	64	2	4	16	8
21	2	2	32	32	2	4	16	8
22	4	4	32	32	2	4	16	8
23	2	2	32	64	2	4	16	8
24	4	4	32	64	2	4	16	8
25	2	2	16	32	4	4	16	8
26	4	4	16	32	4	4	16	8
27	2	2	16	64	4	4	16	8
28	4	4	16	64	4	4	16	8
29	2	2	32	32	4	4	16	8
30	4	4	32	32	4	4	16	8
31	2	2	32	64	4	4	16	8
32	4	4	32	64	4	4	16	8

Tab. B.1: Microarchitectral Configurations

$No.$	$DecodeBW$	$IssueBW$	$il1CacheBlks$	$il1BlkSize$	$IntALUs$	$FPALUs$	$RUUsize$	$LSQsize$
33	2	2	16	32	2	2	8	8
34	4	4	16	32	2	2	8	8
35	2	2	16	64	2	2	8	8
36	4	4	16	64	2	2	8	8
37	2	2	32	32	2	2	8	8
38	4	4	32	32	2	2	8	8
39	2	2	32	64	2	2	8	8
40	4	4	32	64	2	2	8	8
41	2	2	16	32	4	2	8	8
42	4	4	16	32	4	2	8	8
43	2	2	16	64	4	2	8	8
44	4	4	16	64	4	2	8	8
45	2	2	32	32	4	2	8	8
46	4	4	32	32	4	2	8	8
47	2	2	32	64	4	2	8	8
48	4	4	32	64	4	2	8	8
49	2	2	16	32	2	4	8	8
50	4	4	16	32	2	4	8	8
51	2	2	16	64	2	4	8	8
52	4	4	16	64	2	4	8	8
53	2	2	32	32	2	4	8	8
54	4	4	32	32	2	4	8	8
55	2	2	32	64	2	4	8	8
56	4	4	32	64	2	4	8	8
57	2	2	16	32	4	4	8	8
58	4	4	16	32	4	4	8	8
59	2	2	16	64	4	4	8	8
60	4	4	16	64	4	4	8	8
61	2	2	32	32	4	4	8	8
62	4	4	32	32	4	4	8	8
63	2	2	32	64	4	4	8	8
64	4	4	32	64	4	4	8	8

Tab. B.2: Microarchitectral Configurations (Cont.)

$No.$	$DecodeBW$	$IssueBW$	$il1CacheBlks$	$il1BlkSize$	$IntALUs$	$FPALUs$	$RUUsize$	$LSQsize$
65	2	2	16	32	2	2	16	4
66	4	4	16	32	2	2	16	4
67	2	2	16	64	2	2	16	4
68	4	4	16	64	2	2	16	4
69	2	2	32	32	2	2	16	4
70	4	4	32	32	2	2	16	4
71	2	2	32	64	2	2	16	4
72	4	4	32	64	2	2	16	4
73	2	2	16	32	4	2	16	4
74	4	4	16	32	4	2	16	4
75	2	2	16	64	4	2	16	4
76	4	4	16	64	4	2	16	4
77	2	2	32	32	4	2	16	4
78	4	4	32	32	4	2	16	4
79	2	2	32	64	4	2	16	4
80	4	4	32	64	4	2	16	4
81	2	2	16	32	2	4	16	4
82	4	4	16	32	2	4	16	4
83	2	2	16	64	2	4	16	4
84	4	4	16	64	2	4	16	4
85	2	2	32	32	2	4	16	4
86	4	4	32	32	2	4	16	4
87	2	2	32	64	2	4	16	4
88	4	4	32	64	2	4	16	4
89	2	2	16	32	4	4	16	4
90	4	4	16	32	4	4	16	4
91	2	2	16	64	4	4	16	4
92	4	4	16	64	4	4	16	4
93	2	2	32	32	4	4	16	4
94	4	4	32	32	4	4	16	4
95	2	2	32	64	4	4	16	4
96	4	4	32	64	4	4	16	4

Tab. B.3: Microarchitectral Configurations (Cont.)

$No.$	$DecodeBW$	$IssueBW$	$il1CacheBlks$	$il1BlkSize$	$IntALUs$	$FPALUs$	$RUUsize$	$LSQsize$
97	2	2	16	32	2	2	8	4
98	4	4	16	32	2	2	8	4
99	2	2	16	64	2	2	8	4
100	4	4	16	64	2	2	8	4
101	2	2	32	32	2	2	8	4
102	4	4	32	32	2	2	8	4
103	2	2	32	64	2	2	8	4
104	4	4	32	64	2	2	8	4
105	2	2	16	32	4	2	8	4
106	4	4	16	32	4	2	8	4
107	2	2	16	64	4	2	8	4
108	4	4	16	64	4	2	8	4
109	2	2	32	32	4	2	8	4
110	4	4	32	32	4	2	8	4
111	2	2	32	64	4	2	8	4
112	4	4	32	64	4	2	8	4
113	2	2	16	32	2	4	8	4
114	4	4	16	32	2	4	8	4
115	2	2	16	64	2	4	8	4
116	4	4	16	64	2	4	8	4
117	2	2	32	32	2	4	8	4
118	4	4	32	32	2	4	8	4
119	2	2	32	64	2	4	8	4
120	4	4	32	64	2	4	8	4
121	2	2	16	32	4	4	8	4
122	4	4	16	32	4	4	8	4
123	2	2	16	64	4	4	8	4
124	4	4	16	64	4	4	8	4
125	2	2	32	32	4	4	8	4
126	4	4	32	32	4	4	8	4
127	2	2	32	64	4	4	8	4
128	4	4	32	64	4	4	8	4

Tab. B.4: Microarchitectral Configurations (Cont.)

C. BASE CONFIGURATION FILE

```
# instruction fetch queue size (in insts)
-fetch:ifqsize                   4
# extra branch mis-prediction latency
-fetch:mplat                     3
# speed of front-end of machine relative to execution core
-fetch:speed                     1
# branch predictor type {nottaken|taken|perfect|bimod|2lev|comb}
-bpred                        bimod
# bimodal predictor config (<table size>)
-bpred:bimod           2048
# 2-level predictor config (<l1size> <l2size> <hist_size> <xor>)
-bpred:2lev            1 1024 8 0
# combining predictor config (<meta_table_size>)
-bpred:comb            1024
# return address stack size (0 for no return stack)
-bpred:ras                       8
# BTB config (<num_sets> <associativity>)
-bpred:btb             512 4
# speculative predictors update in {ID|WB} (default non-spec)
# -bpred:spec_update         <null>
# instruction decode B/W (insts/cycle)
-decode:width                    4
# instruction issue B/W (insts/cycle)
-issue:width                     4
# run pipeline with in-order issue
-issue:inorder                false
# issue instructions down wrong execution paths
-issue:wrongpath               true
# instruction commit B/W (insts/cycle)
-commit:width                    4
# register update unit (RUU) size
-ruu:size                       16
# load/store queue (LSQ) size
-lsq:size                        8
# l1 data cache config, i.e., {<config>|none}
-cache:dl1             dl1:128:32:4:l
# l1 data cache hit latency (in cycles)
-cache:dl1lat                    1
# l2 data cache config, i.e., {<config>|none}
-cache:dl2             ul2:1024:64:4:l
# l2 data cache hit latency (in cycles)
-cache:dl2lat                    6
# l1 inst cache config, i.e., {<config>|dl1|dl2|none}
-cache:il1             il1:8:8:1:l
# l1 instruction cache hit latency (in cycles)
-cache:il1lat                    1
# l2 instruction cache config, i.e., {<config>|dl2|none}
-cache:il2                     dl2
# l2 instruction cache hit latency (in cycles)
-cache:il2lat                    8
# flush caches on system calls
-cache:flush                  false
# convert 64-bit inst addresses to 32-bit inst equivalents
-cache:icompress              false
```

```
# memory access latency (<first_chunk> <inter_chunk>)
-mem:lat               18 2
# memory access bus width (in bytes)
-mem:width                          8
# instruction TLB config, i.e., {<config>|none}
-tlb:itlb              itlb:16:4096:4:l
# data TLB config, i.e., {<config>|none}
-tlb:dtlb              dtlb:32:4096:4:l
# inst/data TLB miss latency (in cycles)
-tlb:lat                           30
# total number of integer ALU's available
-res:ialu                           4
# total number of integer multiplier/dividers available
-res:imult                          1
# total number of memory system ports available (to CPU)
-res:memport                        2
# total number of floating point ALU's available
-res:fpalu                          4
# total number of floating point multiplier/dividers available
-res:fpmult                         1
# profile stat(s) against text addr's (mult uses ok)
# -pcstat                     <null>
# operate in backward-compatible bugs mode (for testing only)
-bugcompat                     false
```

D. SCRIPTS FOR RUNNING THE SIMPLESCALAR SIMULATIONS

```
echo microconfig 1 >> microconfig_data/timing
date >> microconfig_data/timing
./sim-outorder -fetch:ifqsize 2 -decode:width 2 -issue:width 2
         -cache:il1 il1:16:32:1:l -res:ialu 2 -res:fpalu 2 -ruu:size 16
         -lsq:size 8 -max:inst 100000000 -redir:sim microconfig_data/microconfig1_data
         /home/dinesh/mpeg2/out/mpeg2d in0.m2v
date >> microconfig_data/timing
sleep 5
mkdir microconfig_data/config1
mv ws_met*.* microconfig_data/config1

echo microconfig 2 >> microconfig_data/timing
date >> microconfig_data/timing
./sim-outorder -fetch:ifqsize 4 -decode:width 4 -issue:width 4
        -cache:il1 il1:16:32:1:l -res:ialu 2 -res:fpalu 2 -ruu:size 16
        -lsq:size 8 -max:inst 100000000 -redir:sim microconfig_data/microconfig2_data
         /home/dinesh/mpeg2/out/mpeg2d in0.m2v
date >> microconfig_data/timing
sleep 5
mkdir microconfig_data/config2
mv ws_met*.* microconfig_data/config2

echo microconfig 3 >> microconfig_data/timing
date >> microconfig_data/timing
./sim-outorder -fetch:ifqsize 2 -decode:width 2 -issue:width 2
        -cache:il1 il1:16:64:1:l -res:ialu 2 -res:fpalu 2 -ruu:size 16
        -lsq:size 8 -max:inst 100000000 -redir:sim microconfig_data/microconfig3_data
         /home/dinesh/mpeg2/out/mpeg2d in0.m2v
date >> microconfig_data/timing
sleep 5
mkdir microconfig_data/config3
mv ws_met*.* microconfig_data/config3

...
...

echo microconfig 126 >> microconfig_data/timing
date >> microconfig_data/timing
./sim-outorder -fetch:ifqsize 4 -decode:width 4 -issue:width 4
        -cache:il1 il1:32:32:1:l -res:ialu 4 -res:fpalu 4 -ruu:size 8
        -lsq:size 4 -max:inst 100000000 -redir:sim microconfig_data/microconfig126_data
         /home/dinesh/mpeg2/out/mpeg2d in0.m2v
date >> microconfig_data/timing
sleep 5
mkdir microconfig_data/config126
mv ws_met*.* microconfig_data/config126

echo microconfig 127 >> microconfig_data/timing
date >> microconfig_data/timing
./sim-outorder -fetch:ifqsize 2 -decode:width 2 -issue:width 2
        -cache:il1 il1:32:64:1:l -res:ialu 4 -res:fpalu 4 -ruu:size 8
        -lsq:size 4 -max:inst 100000000 -redir:sim microconfig_data/microconfig127_data
         /home/dinesh/mpeg2/out/mpeg2d in0.m2v
date >> microconfig_data/timing
sleep 5
```

```
mkdir microconfig_data/config127
mv ws_met*.* microconfig_data/config127

echo microconfig 128 >> microconfig_data/timing
date >> microconfig_data/timing
./sim-outorder -fetch:ifqsize 4 -decode:width 4 -issue:width 4
        -cache:il1 il1:32:64:1:l -res:ialu 4 -res:fpalu 4 -ruu:size 8
        -lsq:size 4 -max:inst 100000000 -redir:sim microconfig_data/microconfig128_data
         /home/dinesh/mpeg2/out/mpeg2d in0.m2v
date >> microconfig_data/timing
sleep 5
mkdir microconfig_data/config128
mv ws_met*.* microconfig_data/config128
```

E. SCRIPTS FOR RUNNING THE WATTCH SIMULATIONS

```
echo microconfig 1 >> timing
date >> timing
./sim-outorder -fetch:ifqsize 2 -decode:width 2 -issue:width 2
        -cache:il1 il1:16:32:1:l -res:ialu 2 -res:fpalu 2 -ruu:size 16
        -lsq:size 8 -max:inst 100000000 -redir:sim microconfig1_data
        mpeg2/out/mpeg2d in0.m2v
date >> timing
echo microconfig 2 >> timing
date >> timing
./sim-outorder -fetch:ifqsize 4 -decode:width 4 -issue:width 4
        -cache:il1 il1:16:32:1:l -res:ialu 2 -res:fpalu 2 -ruu:size 16 -lsq:size 8
        -max:inst 100000000 -redir:sim microconfig2_data
        mpeg2/out/mpeg2d in0.m2v
date >> timing
echo microconfig 3 >> timing
date >> timing
./sim-outorder -fetch:ifqsize 2 -decode:width 2 -issue:width 2
        -cache:il1 il1:16:64:1:l -res:ialu 2 -res:fpalu 2 -ruu:size 16 -lsq:size 8
        -max:inst 100000000 -redir:sim microconfig3_data
        mpeg2/out/mpeg2d in0.m2v
date >> timing
echo microconfig 4 >> timing
date >> timing
./sim-outorder -fetch:ifqsize 4 -decode:width 4 -issue:width 4
        -cache:il1 il1:16:64:1:l -res:ialu 2 -res:fpalu 2 -ruu:size 16 -lsq:size 8
        -max:inst 100000000 -redir:sim microconfig4_data
        mpeg2/out/mpeg2d in0.m2v
date >> timing

 ...
 ...

echo microconfig 126 >> timing
date >> timing
./sim-outorder -fetch:ifqsize 4 -decode:width 4 -issue:width 4
        -cache:il1 il1:32:32:1:l -res:ialu 4 -res:fpalu 4 -ruu:size 8 -lsq:size 4
        -max:inst 100000000 -redir:sim microconfig126_data
        mpeg2/out/mpeg2d in0.m2v
date >> timing
echo microconfig 127 >> timing
date >> timing
./sim-outorder -fetch:ifqsize 2 -decode:width 2 -issue:width 2
        -cache:il1 il1:32:64:1:l -res:ialu 4 -res:fpalu 4 -ruu:size 8 -lsq:size 4
        -max:inst 100000000 -redir:sim microconfig127_data
        mpeg2/out/mpeg2d in0.m2v
date >> timing
echo microconfig 128 >> timing
date >> timing
./sim-outorder -fetch:ifqsize 4 -decode:width 4 -issue:width 4
        -cache:il1 il1:32:64:1:l -res:ialu 4 -res:fpalu 4 -ruu:size 8 -lsq:size 4
        -max:inst 100000000 -redir:sim microconfig128_data
        mpeg2/out/mpeg2d in0.m2v
date >> timing
```

F. DETAILS OF PARETO OPTIMAL SOLUTIONS

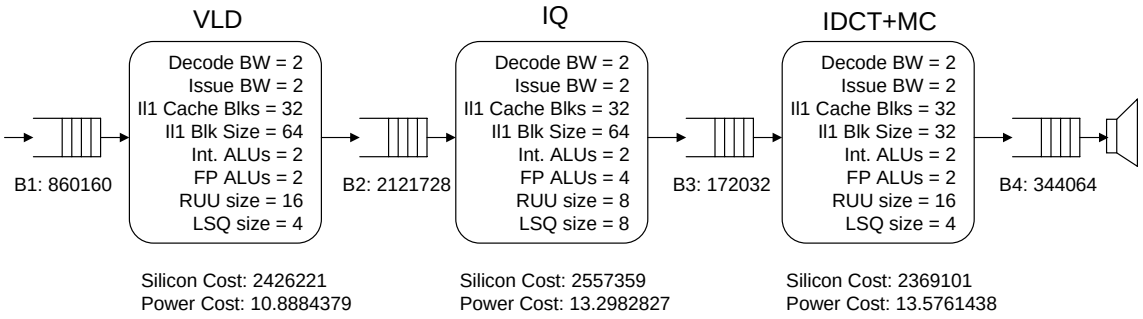


Fig. F.1: Pareto Optimal Solution B

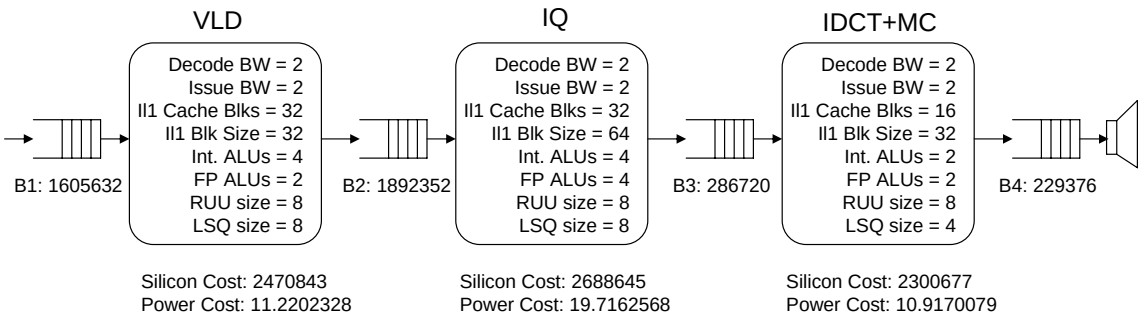


Fig. F.2: Pareto Optimal Solution J

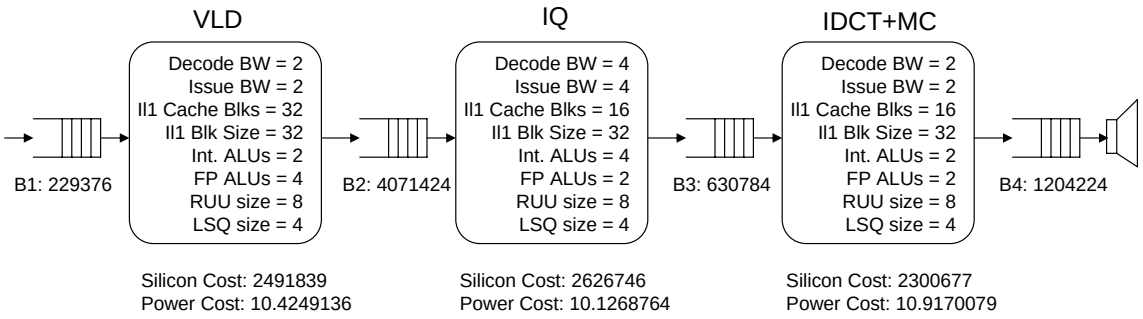


Fig. F.3: Pareto Optimal Solution P