Yield Maximization for System-level Task Assignment and Configuration Selection of Configurable MultiProcessors

Love Singhal
Center for Embeddded
Computer Systems
University of California, Irvine
lsinghal@ics.uci.edu

Sejong Oh
Korea Advanced Institute of
Science and Technology
Daejeon, Republic of Korea
sejong.oh.sayhi@gmail.com

Eli Bozorgzadeh
Center for Embeddded
Computer Systems
University of California, Irvine
eli@ics.uci.edu

ABSTRACT

Configurable multiprocessor system is a promising design alternative because of its high degree of flexibility, short development time, and potentially high performance under constraints and challenges driven by applications. An important design challenge at 45nm for multi-core system is manufacturing process variation. Due to increasing concern of WID variation, designers will have to choose configurations of processing cores that maximize yield of the system while not affecting performance and throughput constraints. Due to interdependency between processor configuration selection and task allocation and its impact on yield and latency constraints, we tackle both problems simultaneously. In this paper, we propose the problem of task allocation and configuration selection for yield optimization. We prove the problem is NP-hard and propose an optimal pseudo-polynomial on Serial-Parallel graphs. We target streaming applications in pipelined reconfigurable multiprocessor systems. We provide a case study of configurable Leon processors as the cores implemented on FPGA. Results show that proposed problem could result in significant improvement of the timing yield of the system by exploiting extra slack on tasks.

Categories and Subject Descriptors: C.4 [**Performance of Systems**]: Reliability, availability, and serviceability; C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems

General Terms: Design, performance, and reliability.

Keywords: configuration selection, delay budgeting, process variation, task allocation, timing yield, and within-die variation.

1. INTRODUCTION

Configurable multiprocessor system is a promising design alternative because of its high degree of flexibility, short development time, and potentially high performance attributed to application specific techniques. This configurability allows designers to change, for example, cache size, cache associativity, pipeline stages, register window, functional units, etc. at later stages of design process [1, 2]. The configurability can also be realized using either fully or partially reconfigurable hardware. Examples are soft core processors [3] such as MicroBlaze processors that are configurable, incorporating custom instructions [4], or co-processors by programming the hardware, etc.

Multi-core system designers today face enormous challenges in selecting right configurations of hundreds of cores on their chip. An important design challenge at 45nm for multi-core system is manufacturing process variation. Due to increasing with-in die (WID) variation, performance yield of a core will change on different parts of a chip. Moreover, performance yield of a core will vary with different configurations of the core, due to difference in number of critical paths [5].

Due to increasing concern of WID variation, designers have to choose configurations of processing cores that maximize yield of the system while not affecting performance and throughput constraints. This paper focuses on configurable multiprocessors and aims to exploit configurability of the cores to achieve the best yield. We specifically target streaming applications and we tune each processor to the configuration which can lead to enhance the yield of overall system. We also consider the impact of configurations on the execution cycle count of the allocated task on the processor. If configuration selection is aware of task allocation, the processor configurations will not affect the throughput of the system.

Due to interdependency between processor configuration selection and task allocation and its impact on yield and latency constraints, we tackle both problems simultaneously and we propose the problem of task allocation and configuration selection for yield optimization. We prove the problem is NP-hard even for task series (task chains). We propose an optimal pseudo-polynomial algorithm which is adapted from discrete budgeting in high level synthesis. The proposed algorithm targets serial-parallel graphs, as widely used to model embedded system benchmarks and data-parallel streaming applications. The algorithm is dynamic programming based and can be simply modified to an approximation algorithm in order reduce the runtime complexity of the algorithm while returning close-to optimal solution.

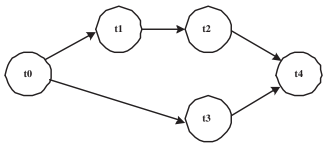
The target architecture as described in the next section is pipelined and consists of a set of configurable processors. In [6–8], researchers have proposed novel variability-aware task allocation and scheduling algorithms . However, the target architecture does not consider pipeline scheme, and the processors in their architecture are not configurable and have a fixed yield at a given clock frequency. Hence, their approach in not applicable to the problem in this paper. In our experiments, we do two sets of experiments. In the first set, we target E3S applications that are run on a multi-core system. In the second set, We use FPGA for implementing configurable Leon-based multi-core systems. We run streaming multimedia benchmarks on this system. Experimental results show that significant improvement in timing yield can be achieved by exploiting extra slack on tasks. Our proposed algorithm is also able to find optimal task allocations and select configurations of processors that give maximum timing yield.

2. ARCHITECTURE MODEL

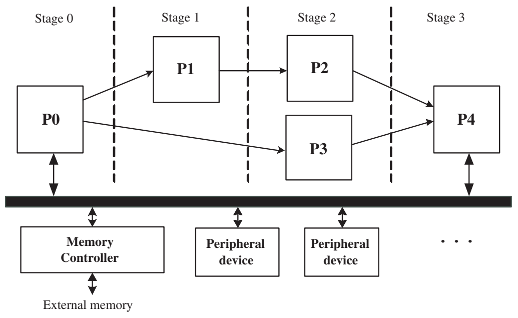
In this section, we show the architecture model used in our paper.

2.1 Target Configurable MultiProcessor System

The system architecture that we target is a multiprocessor platform designed for efficient execution of stream applications. Fig. 1 shows an example task graph and a block diagram of the system architecture for the example task graph. The multiprocessor system consists of homogeneous processors, interconnections, peripheral devices, and memory controller. For simplicity of explanation, tasks are mapped on separate processors without any optimization such as task merging. Processors are named P_i and arrows indicate interconnection between components. Tasks mapped on the processors are executed in the pipeline fashion. Dashed lines in the figure represent how the pipeline is divided and the example system has a four-stage pipeline with five processors. Processors are connected by on-chip interconnection such as shared bus fabric, point-to-point bus, crossbar switches, etc. Tasks accessing external memory or peripheral devices are connected to a shared bus to which external memory controller and peripheral devices are connected. The application throughput is the reciprocal of the execution time of the longest-latency pipeline stage and the application latency is defined as the elapsed time required for the input data to pass all pipeline stages.



(a)



(b)

Figure 1: An example task graph and a block diagram of the system architecture for the example task graph

We assume that processors are homogeneous but they are independently configurable. Each processor has its own configuration options and the degree of configuration is similar to popular configurable processors such as Microblaze processor, Leon processor, and Xtensa processor. The designer can include or exclude some components such as floating-point unit and Multiply-and-Accumulate unit, adjust the number of registers, play the trade-off between latency and clock timing in some components, and so on. For example, the designer can allow additional delay cycles to functional units in processors when running tasks in non-critical paths, which increases the yield by shortening the length of critical paths or reducing the number of critical paths [5].

2.2 Configurability

In our framework, we assume that processors in the system are partially or fully customizable. This class of multi-processor system is called application specific or configurable multiprocessor system. The synthesis of such systems has been a major area of research and many techniques have been proposed to improve performance of the system under throughput or latency constraints [3, 4]. For example, authors in [3] propose implementation of application specific multiprocessor system on field programmable gate arrays (FPGAs) for exploiting parallelism in task graphs. Modern FPGA devices can hold tens of soft processors in a single chip. Thus they can allow significant performance improvement as well as flexibility for application specific adaptations.

Processors in such systems can be customized for application domains in many ways. For example, they can be customized to change register file size and its data transfer ports [2], cache size/associativity and block size [1], the number of pipeline stages, and so on. These features impact the clock frequency, CPI, and the number of critical paths in the designs. Recently, processors are also customized for mitigating the impact of process variations and improving the yield of the system [5].

2.3 Yield Optimization

As technology scales, the effect of manufacturing process variation becomes more predominant. Historically, performance metrics have varied from wafer to wafer or lot to lot. At-speed testing techniques combined with speed-binning has been used to partially compensate for die-to-die (or inter-die) variations. Now, both within-die (WID) and inter-die variations have become strong in the new generations of transistors. As polysilicon gate lengths have decreased below the wavelength of light used in the optical lithography process, the systematic and random within-die variations of channel length have exceeded the die-to-die variations [9]. High within-die variations will lead to variations in the performance parameters of different cores on a multiprocessor system. The multicore system designers have to, therefore, take into account the varying performance of their cores on different parts of the chip.

The *parametric yield* (or simply, *yield*) of the system which is defined as the probability of the system meeting a specified constraint, $Yield = P(Y \leq Y_{max})$, where Y can be performance or power [6], has become an important design metric for the new generation of designs. Like area, wirelength, and power, yield is now an important objective in optimization efforts of new physical synthesis tools. If the timing yield of a system is lower, then the system will fail to meet its timing constraints in most of the chips after manufacturing. The worst-case delay analysis is not feasible anymore as it can be extremely pessimistic of the delay and can severely affect other optimization goals of the synthesis tools. Timing yield, on the other hand, provides a middle path solution to managing variations.

Like other high-performance integrated circuits, Field Programmable Gate arrays (FPGAs) are affected by process variation. However their reconfigurability allows to possibly compensate for the variation by adapting the application circuit based on the measured parameters.

There are many techniques in literature for computing yield of the system. The two most widely used techniques are path-based and block-based statistical timing analysis (SSTA). The path based SSTA is considered more accurate, especially when the low-level circuit information regarding paths is given. For a single path π in a given circuit implementation, with mean delay μ_π and variance σ_π^2, the yield of the path for given clock period T will be [10]:

$$Yield_\pi = D_\pi(T) = \frac{1}{2} + \frac{1}{2} erf(\frac{T - \mu_\pi}{\sigma_\pi \sqrt{2}}) \qquad (1)$$

where erf is the error function. In general, a circuit implementation will have a number of paths which will contribute to yield loss. If there are P paths which have sufficiently little delay slack such that they impact on yield and are 'near-critical' paths, the yield of the circuit is the product of the yields of the paths:

$$Yield = D_\pi(T)^P \tag{2}$$

Equation 2 assumes that the near-critical paths are independent and have the same mean delay and variance, and therefore the same yield.

In multiprocessor systems, the timing yield of each processor can be computed using path based or block based SSTA. After that, the timing yield of the whole system with M blocks/cores running at clock period T is given as follows:

$$Yield_T = \prod_{i=1}^{M} Yield_T(i) \tag{3}$$

In our work, we use these timing yield equations for yield maximization of the system. Next, we propose our task allocation and configuration selection problem on reconfigurable multi-processor systems as described in this section.

3. PROBLEM FORMULATION

In this section, we propose the problem of task allocation for yield maximization suited for streaming applications. For the sake of simplicity, we assume that only one kind of processor is implemented in the multi-core system and the processor can be configured to multiple configurations. The problem can be easily extended to multiple families of processors and as such, our solution will not differ with multiple families of processors.

We assume that we are given a graph of n tasks and execution cycle counts of each task on each configuration of the processor is predetermined. Let P be the set of all possible configurations for the configurable processor(s). The number of possible configurations is equal to p. The cycle count of each task on each configuration is represented by c_{ij} where $1 \leq i \leq n$ and $1 \leq j \leq p$. We also assume that the yield of the processor at a given clock frequency, which is dependent on both its configuration and location on the chip, is also predetermined. This is done by partitioning the area of the chip into multiple regions. In each region, all configurations of the processor are implemented to calculate the yield. Let L be the set of all regions such that $|L| = l$, the number of regions. The yield of a processor at a given clock frequency is represented by y_{jk} where $1 \leq j \leq p$ and $1 \leq k \leq l$.

Each task is mapped to one processor for execution. The configuration and location of the processor will determine the cycle count of the task and yield of the processor. Processor at a certain configuration can provide a better yield but may come at the cost of increase in execution cycle counts of the task. For example, reducing the cache size may lead to increase in cycle counts of some task but provides a better yield [11]. In addition, if a processor is located in the region where the process variation of underlying silicon is negligible, the processor can be configured to provide minimum cycle counts for the task. Hence, there is a trade-off between the yield of a processor and the cycle counts of the tasks, depending on the configuration and location of the processor. On the other hand, we cannot increase the cycle counts of the tasks significantly due to latency and throughput constraints driven by the application. Tuning processor configuration together with task allocation enables to adapt the system to tolerate the variation of underlying silicon of the region while ensuring the increase in cycle count of the task on the configured processor can be tolerated under latency constraints of the application. Hence we introduce the system-level task allocation and configuration selection problem for yield maximization. The problem is defined as follows:

Problem Statement:

- Given a task graph G, cycle counts c_{ij} of each task i in configuration j of a processor, and yield, y_{jk} of the processor with configuration j in region k at a given clock frequency,
- Allocate each task to a processor and select the configuration for each processor such that the total yield of the system is minimized,
- Subject to given latency and throughput constraints for the task graph.

The yield of the total system is the product of the yields of all the processors. The problem defines two types of constraints. The throughput constraint is used for guaranteeing the performance of the system. The throughput of the system is measured by the number of cycles per task. Thus, there is an upper bound on the number of cycles of each task. The latency constraint is the constraint on total number of cycles from start operation of the task graph to the last operation. The total number of cycles defines the total latency of the system. For example, for a streaming MPEG application, the latency constraint determines the delay of the system to process a frame.

At a glance, this problem is related to delay budgeting [12]. Task level delay budgeting problem exploits the extra latency (referred as to *budget*) each task can tolerate under the given latency constraint and distributes the delay budget among the tasks such that the weighted sum of allocated delay budgets is maximized. In our problem, increase in cycle counts can be viewed as delay budget. Each configuration of a processor leads to increase in cycle counts but gain in objective function (i.e., yield) in return. But the gain is not only the function of configuration of the processor but also the region the processor is located at. Location constraint adds another dimension to the problem and increases it complexity. For a given increase in cycle count of a task, there can be up to p different values of yield enhancement depending to which processor it can be allocated to. In delay budgeting, for a given budget, the gain is defined as a single value and it is not location dependent.

This should convince us the problem is not a delay budgeting problem but it has similarities in some ways. For example, under same process variation across all regions, the problem becomes delay budgeting since the location constraint is relaxed. In particular, it is close to discrete budgeting in which increase in delay budget is not regular as the increase in cycle count is not regular as well. In addition, in all the proposed delay budgeting problem, objective function is modeled as a linear function of delay budget. In our problem, the objective is not a linear function and we do not use any approximation technique to linearize it.

LEMMA 1. *Task allocation and Configuration selection problem is NP-hard.*

PROOF. If all the regions (or locations) have same yields, and the graph is made of tasks connected in series, the problem is a discrete budgeting on a directed path. The discrete budgeting problem is shown to be NP Hard even on directed paths by transformation from the subset problem [12]. □

In order to solve this problem on the general case, we propose a pseudo polynomial *optimal* algorithm to solve this problem. We assume the graphs are series-parallel graphs (SP graph). SP graphs are the most common topology for task graphs in embedded systems and data-parallel/streaming applications (such as E3S benchmark suite [13]).

4. ALGORITHM ON SP GRAPH

We provide a pseudo polynomial algorithm for solving this problem when the task graph G is a series-parallel (SP) graph. A Series-Parallel graph SP is defined as a graph which is composed of series or parallel combinations of smaller SP graphs. The SP graphs are fairly common in designs, and are studied extensively in literature for their many interesting properties. Many embedded system applications have component graphs that resemble SP graphs. These include MPEG encoder applications, Software defined radio, and Embedded Microprocessor Benchmark Consortium benchmarks [13].

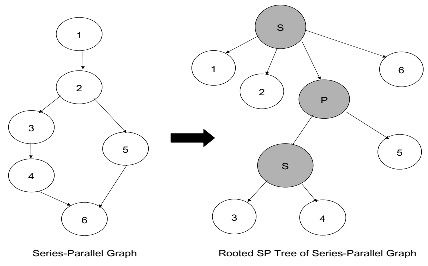


Figure 2: SP tree from series-parallel graph

Our algorithm is based on dynamic programming algorithm presented in [12]. However, it has been extended to include location constraint for task allocation and configuration selection problem. Location constraint enables uniqueness of allocation of each task to one processor and vice versa. As opposed to algorithm in [12] in which the objective function is summation of delay budgets, our objective function is a non-linear function and is a multiplication of the yield at each cycle count and configuration point. We present how to adapt the algorithm to include the the location constraint and non-linear objective function. We show that after this adaptation, the algorithm remains optimal with the same complexity.

We first create an SP tree of the SP graph. An SP tree is a tree whose internal nodes are either S or P nodes, and whose leaf nodes are the nodes of the original task graph. The S node represents a series relationship between the child nodes and P node represents that child nodes are connected in parallel. The creation of SP tree from SP graph can be done in linear time. Figure 2 shows the creation of an SP tree from SP graph.

The idea behind our algorithm is that for every node of the SP tree, we maintain a list of all possible task allocation for the tree rooted at that node. Therefore for all internal nodes in the SP tree, we maintain a list of all possible task allocations for all the child nodes. We also maintain a list of all possible configurations that can be assigned to the child nodes in those task assignments. For each of the task allocation, the total delay and the yield are computed. Hence, for every node i (S, P or leaf node), we maintain a list T_i of tuple (c, y, L), where c is the total delay of the tree rooted at the node based on the assigned configurations, y is the yield of the child nodes and L is the list of all locations used, $L \subset P$. We remove all the inferior task assignments and configurations. The inferior point here is the assignment that has higher delay and lower yield for the same regions used, compared to any other assignment for that node.

Algorithm 1 shows the pseudo code of the dynamic programming algorithm. The algorithm basically computes the list of tuples for all the nodes in the SP tree. Then the algorithm selects a tuple with the highest yield at the root node of the whole SP tree.

Algorithm 2 shows the pseudo code for calculating the list of tuples for any node. The list of tuples is calculated in the depth-first search manner. Once the list of tuple T_i is prepared for a child node, the current list of tuples T_{root} is merged with the child list T_i. This merging is done exhaustively. While merging the two lists, we consider all pairs of tuples from the two lists. We make sure that we only consider the tuples that do not use same resources ($L1 \cap L2 = \phi$)(uniqueness of allocation and location constraints). The merged tuple is added to the new list of tuples of the current node. While the cycle of the merged tuple is either found by addition or maximum of the cycle counts of each tuple, the yield of the merged tuple is found by multiplication of yields of the two tuples (yield function). After all the pairs of two lists are merged, all the inferior points in the lists are removed. Also, all the tuples that do not meet latency constraints are also removed from the list.

Algorithm 1 Pseudo code for the Dynamic programming algorithm

```
1: function DYNAMIC PROGRAMMING ALGORITHM
2:     Form rooted SP tree, $SP$.
3:     Compute the yields of all processor configurations in all regions
4:     Compute cycle counts of all tasks for all processor configurations
5:     Initialize $T_i$, the list of all $(c, y, L)$ tuples, for all the leaf nodes
6:     $T_{root} = ComputeTuples(root)$
7:     return $(c, y_{max}, L) \in T_{root}$ , a tuple with the maximum yield
8: end function
```

Algorithm 2 Pseudo Code for calculating the list of tuples

```
1: function $ComputeTuples(\ root)$
2:     if $root$ is not a leaf node then
3:         $T_{root} = \phi$
4:     end if
5:     for all child node $i$ of $root$ do
6:         $T_i = ComputeTuples\ (i)$
7:         $T_{temp} = \phi$
8:         for all pair of tuples $(c1, y1, L1) \in T_{root}$ and $(c2, y2, L2) \in T_i$ do
9:             if $L1 \cap L2 = \phi$ then
10:                if $root$ is an S node then
11:                    $T_{temp} = T_{temp} \cup \{(c1 + c2, y1 * y2, L1 \cup L2)\}$
12:                else if $root$ is a P node then
13:                    $T_{temp} = T_{temp} \cup \{(max(c1, c2), y1 * y2, L1 \cup L2)\}$
14:                end if
15:            end if
16:        end for
17:        Remove all inferior and infeasible points in $T_{temp}$
18:        $T_{root} = T_{temp}$
19:    end for
20:    return $T_{root}$
21: end function
```

Location constraint is handled in merging process of the tuples, to ensure the resulted tuple does not violate the uniqueness of allocation of each task to one processor at particular region and vice versa. The merging process is more complex than the merging stage in discrete budgeting. Our algorithm is a pseudo polynomial algorithm. It is able to find optimal solution for the task assignment and yield maximization problem. In order to run the algorithm with less runtime complexity, we can bound the number of elements in the list of tuples for a node depending on the degree of accuracy required. [12] has proposed an interesting criteria to provide an approximation algorithm for such dynamic programming when the number of elements in the list of tuples is bounded. This approximation technique can also be applied to our algorithm to decrease the runtime complexity but give results that are close to optimum.

5. EXPERIMENTS AND RESULTS

In this section, we show the results of our technique and our case study on application-specific multiprocessors using Leon processor on FPGA. We implement our algorithm in Java and experiments are conducted in

5.1 Results on E3S benchmarks

In this section, we analyze our proposed technique for task allocation and configuration selection using a set of embedded system synthesis benchmarks from E3S suites [13]. These benchmarks are extracted data provided by the Embedded Microprocessor Benchmark Consortium (EEMBC), and includes real cycle counts of applications on various industry cores. We use AMD K6-2E 400 Mhz processing core for our experiments. We extract the cycle counts of various benchmarks for this core. We assume that there are 5 different configurations of this core, the different configurations affect the CPI by at most 30%. The timing yield of different configurations is assumed to vary from 0.985 to 0.999. This represents a case when all configurations have high probability of meeting timing constraints. Due to within-die variation, we assume that yield of these configurations deteriorate by 0.995 at different locations. We then run our benchmarks for different values of latency constraints. The latency constraints are chosen from the ASAP time (fastest) to the latency constraint of 1.3 times the ASAP time.

Table 1 shows the results of the various E3S benchmarks. The various columns show the yields of the task graphs after task allocation and configuration selection, at various latency constraints. The first 6 benchmarks consists of tasks in series, and the remaining benchmarks have series-parallel topology. We compare the results with the column for *Best cycle*. This column refers to the solution when all tasks are assigned to the processor configurations that minimize their cycle counts. Usually, designers configure the processors that minimizes the cycle counts of the tasks. However, we see that such configuration and assignment can negatively impact the yields of the system as they do not use the budget on tasks. On average, the yield of the $bestcycle$ approach is worse by 12.3% than the solution when timing constraint is 10% more than ASAP time (column $1.1\times$ *ASAP time*) The extra clock cycle budget on the tasks can help improve the yield of the system significantly.

5.2 Target System for Configurable MultiProcessor Case Study

We now discuss our case study of application specific multiprocessor system. Our target multiprocessor system is Leon-MP, Leon multiprocessor system. The Leon-MP is a shared-memory and shared address-space multiprocessor system. It is based on the open source SPARC V8 compliant Leon multiprocessor core. The Leon processor is a synthesizable VHDL model of a 32-bit processor compliant with the SPARC V8 architecture. Its integer unit consists of a 7 stage pipeline; and there are hardware multiply, divide and MAC units and separate instruction and data caches. The processor is particularly designed for system-on-a-chip designs. It is also highly configurable and the designer can configure functional units, register windows, and instruction and data caches. In the Leon-MP system, Leon processors are attached to the shared AMBA high performance bus (AHB) and it provides snoopy protocol mechanism for cache coherence, invalid and update. Peripheral devices are attached to AMBA peripheral bus (APB) bus and they are accessible through AHB/APB bridge.

The Leon-MP system can have a variable number of Leon processors; however, the system with more than four cores are not practical due to the memory bandwidth of the shared bus [14]. Our system architecture may require more than four cores to achieve a high degree of throughput. However, since it is not necessary that all processors need to access external memory in stream applications, interconnections between processors can be separated from the main shared bus. Therefore, we assume that our target system has the main shared bus for external memory access and several shared bus fabrics for interprocessor communication, as shown in Figure 3. The instructions and data for processors are loaded into local memories and each processor has its own instruction and data cache. Since each process has its own caches, the shared bus fabric must provide cache coherence mechanisms and burst transaction to fetch data into cache lines efficiently. In this paper, we assume that interconnection between processors are well-designed and partitioned properly to minimize the communication overhead. In addition, the communication overhead is not taken into account in the proposed algorithm and left for the future research.

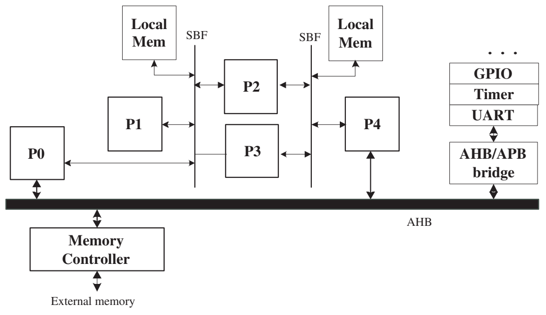


Figure 3: A example block diagram of the target multiprocessor system

5.3 Configurable MultiProcessor Experimental Setup

The experimental flow is divided into three major phases - critical path analysis, performance analysis(execution cycles), the propose algorithm for yield-aware task allocation - as shown in Fig. 4. In the critical path analysis, we synthesize Leon processor with several processor configurations in order to observe the number of critical paths. We use Xilinx Virtex 5 FPGA as the development platform. The performance analysis run all tasks in the benchmarks on a cycle-accurate simulator to measure the number of execution cycles of each kernel for every processor configuration. The proposed algorithm takes as inputs the critical path information and the performance results and perform yield-aware task allocation.

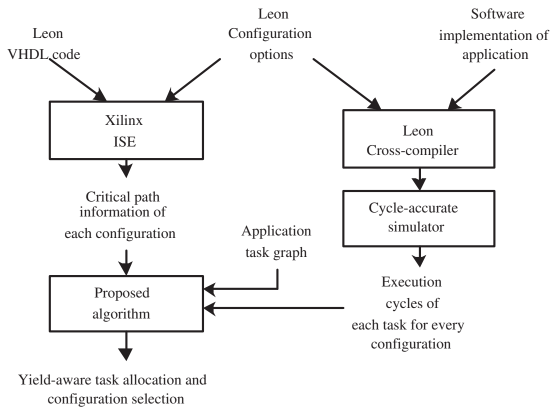

Leon VHDL code
Leon Configuration options
Software implementation of application
Xilinx ISE
Leon Cross-compiler
Critical path information of each configuration
Cycle-accurate simulator
Application task graph
Execution cycles of each task for every configuration
Proposed algorithm
Yield-aware task allocation and configuration selection

Figure 4: Experimental flow of Task Assignment and Configuration Selection on Configurable MultiProcessor System

The input benchmarks are stream applications composed of several image processing kernels and TI image library. The benchmarks are analysis/filter applications and the front part of JPEG compression before Huffman coding. In order to measure the execution cycles, we have developed a cycle-accurate simulator for a Leon-based mono-processor system compatible to the cross compiler and the booting program in BCC system. The simulator provides simulation modes for Leon processor, instruction and data cache, AHB and APB, and UART. Although Gaisler Research provides high-performance simulators of the Leon processors, the simulators are commercial and they don't provide some configuration options of the Leon processor. We run each task in the applications multiple times and measure the average number of execution cycles

Table 1: Yields of various benchmarks on AMD K-6 multi-processor system using various latency constraints

Benchmark	Best Cycle	ASAP Time	$1.1\times$ ASAP time	$1.15\times$ ASAP time	$1.2\times$ ASAP time	$1.25\times$ ASAP time	$1.3\times$ ASAP time
E3S - Telecom0	0.876	0.876	0.956	0.9658	0.9658	0.9757	0.984
E3S - Network1	0.876	0.876	0.956	0.958	0.972	0.977	0.9806
E3S - Network2	0.876	0.876	0.948	0.956	0.9674	0.977	0.984
E3S - Network3	0.876	0.876	0.953	0.963	0.972	0.977	0.984
E3S - Auto1	0.876	0.876	0.956	0.963	0.9658	0.9757	0.984
E3S - Auto3	0.839	0.839	0.932	0.943	0.9544	0.9674	0.971
E3S - Telecom1	0.814	0.8384	0.9417	0.9514	0.9514	0.961	0.969
E3S - Auto2	0.7355	0.7573	0.9124	0.928	0.947	0.9553	0.9553
E3S - Consumer0	0.789	0.789	0.936	0.939	0.949	0.9635	0.9669
E3S - Automation0	0.8396	0.8644	0.943	0.943	0.9543	0.9626	0.967
Average	0.8397	0.8468	0.9434	0.95102	0.95991	0.9692	0.97458

Table 2: Yields of various real benchmarks on Leon multi-processor system using various latency constraints

Benchmark	Best Cycle	ASAP Time	$1.1\times$ ASAP time	$1.2\times$ ASAP time	$1.5\times$ ASAP time	$2\times$ ASAP time	$3\times$ ASAP time
Laplace	0.8537	0.8537	0.897	0.897	0.9157	0.9299	0.9355
Sobel	0.792	0.792	0.8597	0.8597	0.8912	0.897	0.902
Unsharp	0.76	0.76	0.87	0.87	0.888	0.902	0.907
JPEG	0.851	0.851	0.851	0.851	0.8644	0.8778	0.883
Average	0.8142	0.8142	0.8694	0.8694	0.8898	0.9017	0.90687

for a single execution of each task. We also exclude the first few executions of tasks to avoid the transient state of the system.

We pick 5 configurations of Leon processor for our experiments. These configurations are shown in Table 3. All the benchmarks are executed using these configurations to measure execution cycles. All the configurations in Table 3 are implemented on Xilinx Virtex 5 FPGA to measure the yield of configuration. In order to compute the yield of various configurations, we measure the number of critical paths in each configuration. Assuming the yield of a single path to be 0.9985 (the value of $\mu + 3\sigma$), we calculate the yield of the processor using Equation 2. We also assume that due to within-die variation, the yields of various processor configurations deteriorate by 0.995.

5.4 Results of Leon Processor Case Study

Table 2 shows the results of yields of various benchmarks. Table 2 shows that the yields can be improved significantly if the budget of additional cycle counts is provided to the task graphs. The task graphs are able to utilize the extra budget allocated to them to maximize the yield of the system. We do not see significant yield improvement in the JPEG benchmarks because it is made up of lot of parallel paths of same critical path delay. This topology reduced the yield of the system. Also, our algorithm is able to find optimal yield effectively. Thus, dynamic programming can be used to solve problems of task allocation and select configuration of processors.

We also see that if we assign tasks to the processors with the highest yield, then the total latency of the system becomes more than 3 times the ASAP time. This condition may not be acceptable. Hence, budgeting is required to assign tasks to configurable processors.

6. CONCLUSION

Due to increasing concern of WID variation, designers have to choose configurations of processing cores that maximize yield of the system while not affecting performance and throughput constraints. In this paper, we present variation-aware system-level task allocation and configuration selection on reconfigurable multiprocessor systems. We prove the problem is NP-hard and present an optimal pseudo-polynomial algorithm on SP graphs. We focus on streaming applications and use FPGA for implementing configurable Leon-based multi-core systems. We also experiments E3S benchmark on multicore systems. Results show that the proposed solution could result in significant improvement in timing yield by exploiting extra slack on tasks.

Table 3: Leon Processor configurations. NRWIN represents number of register windows. MLat and LDelay represent multiplier latency cycles, and load delay cycles, respectively. The three numbers in the cache configuration represents associativity, the number of blocks, and the line size in bytes in order.

	NRWIN	I-Cache	D-Cache	MLat	LDelay
C1	8	2-8-32	1-8-16	1	1
C2	16	4-4-32	2-4-16	1	1
C3	8	4-4-32	4-4-16	5	2
C4	6	2-4-32	2-4-32	5	2
C5	10	2-16-32	2-8-16	4	1

7. REFERENCES

[1] C. Zhang, F. Vahid, and W. Najjar, “A highly configurable cache architecture for embedded systems,” in *Int. Symp. on Computer Architecture*, 2003, pp. 136–146.

[2] L. Wehmeyer, M. K. Jain, S. Steinke, P. Marwedel, and M. Balakrishnan, “Analysis of the influence of register file size on energy consumption, code size, and execution time,” *IEEE Trans. Computer-Aided Design of Integrated Circuits*, pp. 1329–1337, Nov. 2001.

[3] J. Cong, G. Han, and W. Jiang, “Synthesis of an application-specific soft microprocessor system,” in *ACM FPGA 07*, 2007.

[4] F. Sun, S. Ravi, A. Raghunathan, and N. K. Jha, “Synthesis of application-specific heterogeneous multiprocessor architectures using extensible processors,” in *IEEE VLSID*, 2005.

[5] X. Liang and D. Brooks, “Mitigating the impact of process variations on processor register files and execution units,” in *MICRO*, 2006, pp. 504–514.

[6] F. Wang, X. Wu, and Y. Xie, “Variability-driven module selection with joint design time optimization and post-silicon tuning,” in *IEEE ASPDAC*, 2008.

[7] F. Weng, C. A. Nicopoulos, X. Wu, Y. Xie, and V. Narayanan, “Variation-aware task allocation and scheduling for mpsoc,” in *IEEE ICCAD*, 2007.

[8] J. Jung and T. Kim, “Timing variation-aware high-level synthesis,” in *IEEE ICCAD*, 2007.

[9] S. G. Duvall, “Statistical circuit modelling and optimizations,” in *Intl. Workshop Statistical Metrology*, June 2000, pp. 56–63.

[10] P. Sedcole and P. Y. K. Cheung, “Parametric yield in fpgas due to within-die delay variations: A quantitative analysis,” in *ACM FPGA*, 2007.

[11] X. Liang and D. Brooks, “Microarchitecture parameter selection to optimize system performance under process variation,” in *IEEE ICCAD*, 2006.

[12] S. Ghiasi, K. Nguyen, E. Bozorgzadeh, and M. Sarrafzadeh, “On computation and resource management in networked embedded systems,” in *IPDPS*, 2003.

[13] R. P. Dick, “Embedded systems synthesis benchmarks suite (e3s).” [Online]. Available: http://www.ece.northwestern.edu/~dickrp/e3s

[14] “Leon open-source processor.” [Online]. Available: http://www.embedded-kernel-track.org/2004/papers.html