Rapid Runtime Estimation Methods for Pipelined MPSoCs targeting Streaming Applications

Haris Javaid   Sri Parameswaran

University of New South Wales, Australia
{harisj,sridevan}@cse.unsw.edu.au

Technical Report
UNSW-CSE-TR-1004
February 2010

Abstract

The pipelined Multiprocessor System on Chip (MPSoC) paradigm is well suited to the data flow nature of streaming applications, specifically multimedia applications. A pipelined MPSoC is a system where processors are connected in a pipeline. To balance the pipeline for high throughput and reduced area footprint, Application Specific Instruction set Processors (ASIPs) are used as the building blocks. Each ASIP in the system has a number of configurations which differ by instruction sets and cache sizes. The design space of a pipelined MPSoC is all the possible permutations of the ASIP configurations. To estimate the runtime of a pipelined MPSoC with one combination of ASIP configurations, designers typically perform cycle-accurate simulation of the whole pipelined MPSoC. Since the number of possible combinations of ASIP configurations (design points) can be in the order of billions, estimation methods are necessary.

In this paper, we propose two methods to estimate the runtime of a pipelined MPSoC, minimizing the use of slow cycle-accurate simulations. The first method performs cycle accurate simulations of individual ASIP configurations rather than the whole system, and then utilizes an analytical model of the pipelined MPSoC to estimate its runtime. In the second method, runtimes of individual ASIP configurations are estimated using an analytical processor model. These estimated runtimes of individual ASIP configurations are then used in pipelined MPSoC's analytical model to estimate its runtime. By evaluating our approach on four benchmarks, we show that the maximum estimation error is 5.91% and 13.21%, with an average estimation error of 2.28% and 5.91% for the first and second method respectively. The time to cycle-accurately simulate the whole design space of a pipelined MPSoC is in the order of years, as design spaces with at least 10^{10} design points are considered in this paper. However, the time for cycle-accurate simulations of individual ASIP configurations (first method) is days, while the time to simulate a subset of ASIP configurations and estimate their runtimes (second method) is only several hours. Once these simulations are done, the runtime of each design point can just be estimated by using the pipelined MPSoC's analytical model's estimation equation.

1 INTRODUCTION

Estimation is an important and critical part of most design space exploration methodologies. Used in conjunction with exploration algorithms which search for some global minima or maxima, estimation provides quick values to guide the exploration through the vast design space. Whether it be a simulated annealing algorithm, a genetic algorithm or a heuristic, the process of getting performance values for evaluation of design points is important. In the case of hardware, accurate performance evaluation necessitates the creation of a chip. While this is not a feasible option, cycle accurate simulation is the next best thing to estimate performance. However, such a simulation can be time consuming for processor based systems, which contain millions of instructions for just a few milliseconds of execution. This is particularly pertinent when millions or even thousands of design points have to be evaluated. Thus, an estimation process which relies purely on cycle accurate simulations for processor based systems is not feasible for creating systems in short design times.

Recently, a new Multiprocessor System-on-Chip (MPSoC) paradigm of pipelined MPSoC was shown to achieve high performance for streaming applications, specifically multimedia applications [1, 2, 3, 4]. A pipelined MPSoC is a system where processors are connected in a pipeline. The input data stream is read by the processor(s) in the first pipeline stage, which is then processed by processor(s) in each pipeline stage, and finally the output data stream is written by the processor(s) in the last pipeline stage. In a pipelined MPSoC, at a given instant, each stage processes different data in a pipelined fashion, thus providing high throughput. For example, JPEG encoder application can be partitioned in to five standalone tasks: Reading and color space conversion; Discrete Cosine Transform; Quantization; Huffman encoding; and, Writing the output file. These standalone tasks can then be allocated to processors in different stages. In such an implementation of JPEG encoder application, while the last stage is busy writing a data block to output file, other stages will be processing data blocks further in the data stream in a typical pipelined fashion.

To balance a pipelined MPSoC system for high throughput with reduced area footprint and power consumption, the notion of ”application specific system” is adopted, whereby each processor in a stage is tuned according to the specific tasks allocated to it. Hence, Application Specific Instruction set Processors (ASIPs) such as Xtensa, Nios and ARC 600 and 700 core families [5, 6, 7] are used, where each ASIP has a number of configurations. Typically ASIP configurations differ by the instruction set architecture (ISA), and instruction and data cache sizes. The design space of a pipelined MPSoC is then the number of all the possible combinations of ASIP configurations (design points). Exploration of such a large design space typically requires a fast performance estimation methodology, augmented with cleverly designed exploration algorithms. Various exploration algorithms exist for pipelined MPSoCs [1, 2, 3, 4], however, not much attention has been paid to fast performance estimation techniques for pipelined MPSoCs.

In this paper, two performance estimation methods for a pipelined MPSoC are presented. In brute force, all the possible combinations of ASIP configurations (design points) need to be simulated, which is infeasible given the slow nature of cycle-accurate system simulation. Thus, an analytical equation is proposed which calculates the runtime of a pipelined MPSoC by utilizing the runtimes of individual ASIP configurations, eliminating the need for cycle-accurate simulation of the system.

In the first method, the runtimes of individual ASIP configurations are obtained through cycle accurate simulations. Thus for every ASIP, simulations are performed for each configuration. For example, an ASIP configuration with instruction set X_1, with cache configuration Y_1, is simulated and the runtime is recorded. Then, simulation for configuration with X_1 and Y_2 is performed, and so forth until configuration with X_N and Y_M has been simulated, where N and M are the maximum number of instruction sets and cache configurations available. The analytical equation is then able to calculate the runtime of the whole pipelined system with differing instances of the individual ASIP configurations.

In the second method, the runtime of each individual ASIP configuration is estimated using an analytical processor model, which is based on very basic ISA information, and cache statistics. The runtime of a configuration with X_1 and Y_1 can be estimated using ISA information of X_1 and cache statistics for Y_1. Thus, simulations are only performed with configurations X_1 with Y_1, X_2 with Y_1, until X_N with Y_1, excluding the exploration of cache configurations, to extract ISA information of each X. This is in contrast to the first method where every combination of X with Y is simulated. For cache configurations, we use SuSeSim tool [8] (similar to Dinero IV [9], but much faster), to obtain the cache statistics for all the cache configurations under consideration (Y_1 to Y_M) based upon the trace of the program. Using these cache statistics and ISA information, the performance of all the ASIP configurations (X_1 with Y_1, X_1 with Y_2 until X_N with Y_M) is estimated using the analytical processor model. These estimated runtimes of the individual ASIP configurations are then used to estimate the runtime of the pipelined MPSoC. By utilizing these two methods, we reduce the number of simulations which significantly reduces the simulation time, making it possible for a designer to explore larger design spaces. As obvious, the penalty is paid in terms of the accuracy of the estimated runtime of the system compared to brute force method.

The rest of this paper is organized as follows. Section 2 will present an overview of existing methodologies for estimating the performance of a processor and a pipelined MPSoC. Section 3 presents the underlying concepts of a pipelined MPSoC. Section 4 presents the two performance estimation methods. Section 5 and Section 6 provide the experimental setup and results, with the conclusion presented in Section 7.

2 RELATED WORK

Performance estimation techniques for processors typically use two different methods: first, processor simulation; and second, processor modeling.

In the simulation domain, cycle-accurate processor simulators are used for performance estimation. Several cycle-accurate simulators, such as PTLSim for x86 architecture [10], RealView ARMulator ISS [11], Xtensa Instruction Set Simulator (ISS) [5], etc. are available for various architectures. The disadvantages of cycle-accurate simulators are their slow speed and the large amount of output they generate.

Processor modeling involves the creation of analytical timing models to describe the processor and estimate the runtime of an application. The advantage of a processor model is that it is less expensive to run compared to a full cycle-accurate processor simulator. The price paid for the speed is in the accuracy of the model. The authors of [12] proposed a MonteCarlo based model for predicting the performance of Itanium-2 processor. The model breaks down the execution time of a processor in terms of net time to execute instructions and the stalls due to data dependencies and cache misses. However, they do not take into account cache exploration as only hardware performance counters are used.

To take into account the effect of different cache configurations on processor performance, trace-based simulation can be used to obtain cache miss statistics. Trace-based cache simulation tools include SuSeSim [8], Dinero IV [9], and CRCB1 and CRCB2 [13]. Cache simulation is a fast and efficient method to accurately obtain the cache miss statistics of an application trace. However, the disadvantage of trace-based simulation is that the cache statistics does not contain sufficient information to obtain the processor stalls that occur due to cache misses.

Singleton et al. [14] exploited the use of cache statistics to predict the runtime of tasks running on a processor. However, these cache measurements were used in Dynamic Voltage and Frequency Scaling (DVFS) techniques to reduce the energy consumption of the processor.

Lee et al. [15] and Joseph et al. [16] proposed a linear regression based model for predicting the performance and power of a processor. Their work is orthogonal to our approach as their methods can be used to further refine the simple model proposed here. We focused on reducing the number of simulations that have to be performed to simulate different cache configurations for a fixed processor. In this case, a wide range of predictors dependent on the microarchitecture of the processor are not required, as used in [15] and [16]. A simple model, based on cache statistics, which is proposed in this paper is enough to predict the runtime of a given application on a given processor with reasonable accuracy.

The authors in [17] modeled an out-of-order superscalar processor at a very detailed level of microarchitecture, which takes into account the effects on the Clock cycle per Instruction (CPI) of the ISA, branch missprediction, the commit and reorder buffer in out-of-order execution, and instruction and data cache misses. In contrast, our approach uses a simple and reasonably accurate estimation technique for performance estimation of individual ASIP configurations in a pipelined system to reduce the simulation time. The concepts introduced in [17] can be used to further improve the accuracy of our estimation method at the cost of more complex analysis of the processor microarchitecture.

Pipelining is a well known technique for high throughput systems, and has been deployed at different levels of design. Various uses of pipelining at system level include exploitation of loop pipelining and pipelined scheduling of tasks on multiprocessor systems to speed up applications [18, 19, 20, 21, 22, 23]. However, none of these works considered ASIPs connected in a pipeline (pipelined MPSoC) which recently has emerged as a viable platform for reduced area footprint and high throughput implementation of streaming applications, specifically multimedia applications [1, 2, 3, 4].

In design space exploration of pipelined MPSoCs, performance estimation is typically done using a mixture of cycle-accurate simulators and system models. There has been no prior work in fast runtime estimation of pipelined MPSoCs. The few works done in the past on pipelined MPSoCs [1, 2, 3, 4, 24] are more focused on design of such systems instead of runtime estimation methods. All these works either propose less accurate estimation methods [2, 24] or assumed that the proposed equations work correctly [3, 4]. The analytical equation proposed in this paper is based on similar concepts, however, we also provide an insight and evaluate our proposed equation rigorously to validate its applicability. Furthermore, runtime estimation for the individual ASIP configurations in the pipelined system is proposed, which is the very first work of its kind in the context of pipelined MPSoCs.

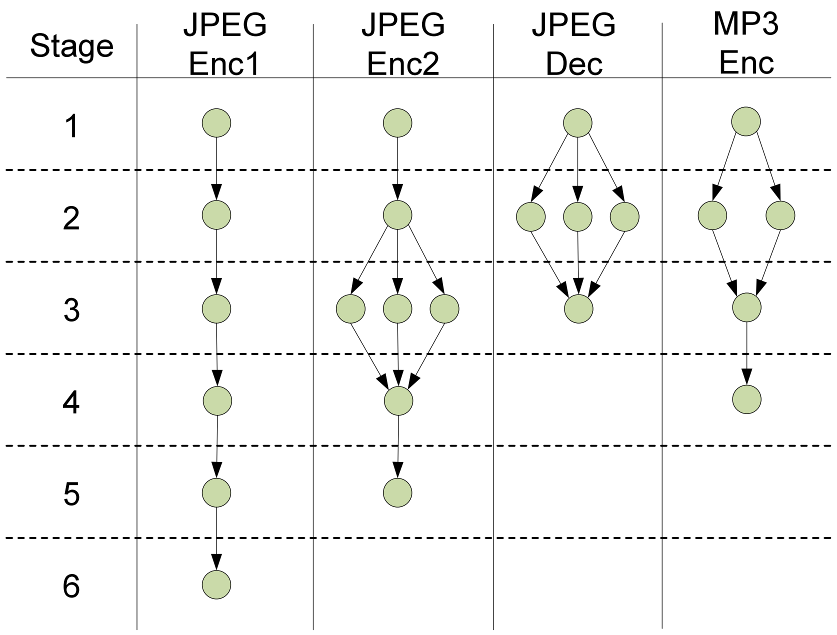


Figure 3.1: Benchmark applications

2.1 Our Contribution

In this paper, we propose a performance estimation methodology for a pipelined MP-SoC system. The presented methodology uses two analytical models: one for the pipelined MPSoC and the other for the ASIP configurations in the pipelined MPSoC. These analytical models are combined with cycle-accurate and trace-based simulation to estimate the performance of a pipelined MPSoC. Using our methodology, the simulation time can be reduced by several orders of magnitude, which allows faster exploration of large design spaces. To the best of our knowledge, this is the first work which targets performance estimation of pipelined MPSoCs with the help of analytical models, and cycle-accurate and trace-based simulation.

3 BACKGROUND

In a pipelined multiprocessor system, processors are connected in a pipeline via queues which implement First In First Out (FIFO) protocol. The typical contention exhibited in a shared bus architecture is avoided through the use of these FIFOs, which allow communication at a much higher bandwidth, increasing the throughput of the system. A typical pipelined system consists of various pipeline stages, where each stage can contain one or more processors. These processors execute some part of the application which is mapped on to them. Hence, the use of ASIPs further increases the throughput of a pipelined system as each processor can be tuned according to the task mapped onto it. By carefully crafting the processors, the overall throughput of the system can be increased (by balancing the stages of the pipeline), while minimizing the area of the system.

Various topologies are possible in pipelined MPSoC systems. A few are shown in Figure 3.1, where each node represents a processor in the pipelined system. We use the

Stage	JPEG Enc1	JPEG Enc2	JPEG Dec	MP3 Enc
1	Read & RGB to YCbCr	Read & RGB to YCbCr	Read & Entropy Decoding	Read File
2	Level Shifting	Level Shifting	Dequantiz. & IDCT for Y, Cb, Cr & Level Shift	Polyphase Filtering
3	DCT	DCT & Quantiz. for Y, Cb and Cr	Color Space Conversion & Write Back	MDCT
4	Quantiz.	Huffman Encoding		Quantiz. & Encoding & Write Back
5	Huffman Encoding	Write Back		
6	Write Back			

Figure 3.2: Partitioned tasks of the applications shown in Figure 3.1

following assumptions for the pipelined MPSoCs considered in this paper:

1. Two processors in series make up two separate pipeline stages. Thus, the depth of a stage is one in terms of the number of processors. However, processors in parallel are considered to be in the same pipeline stage.

2. A processor in stage k can only communicate with processors in stage $k + 1$.

With these assumptions, a pipelined system can be used to implement a streaming application. The data flow nature of streaming applications is well suited to pipelined architectures, which provide efficient implementation platforms [1, 2, 3, 4]. A streaming application contains a kernel which is run several times on the input data stream. The number of times a kernel is executed is referred to as the number of iterations of the application. The operations within the kernel are usually independent of each other, making it possible to execute them in a pipelined fashion. For example, a JPEG decoder kernel can be broken down into smaller operations: Reading file and entropy decoding; Dequantization; Inverse DCT; and, Color space conversion and writing the final bitstream. These operations are standalone tasks, which can be allocated to separate processors, while the communication between these tasks is achieved through the intermediate FIFOs in the pipelined system. Implementations of a few streaming applications are shown in Figure 3.1, where each node represents a standalone task of the entire application. The arrows show the data dependencies between these tasks, which will be mapped onto FIFOs in the pipelined system. Figure 3.2 shows the names of the corresponding tasks for the applications illustrated in Figure 3.1. RGB to YCbCr correspond to color space conversion from RGB to YCbCr, while Quantiz., Dequantiz. and ITransform stand for Quantization, Dequantization and Inverse Transform respectively. As illustrated in Figure 3.1 and 3.2, stage 3 of JPEGEnc2 perform DCT and Quantization of Y, Cb and Cr components of a macro block in parallel.

In these implementations, we assumed that the feedback loops in an application are contained within a single node (this is achieved by partitioning the application at a higher level), simplifying the structure of the application graphs. This simplification helped us to develop intuitive analytical equations as part of fast performance estimation techniques, yet being able to implement several important multimedia applications on pipelined MPSoCs as illustrated in Figure 3.1. In future, we will extend the proposed analytical equations for systems with feedback loops. It should also be noted that a new Model of Computation (MoC) such as Kahn Process Network (KPN) [25] or Signal Data Flow Graph (SDF) [26] for modeling multimedia applications is not being proposed here, and is beyond the scope of this paper. The application graphs shown in Figure 3.1 can be considered as a sub-class of KPN or SDF.

Once the standalone tasks of an application are mapped onto ASIPs in a pipelined system, the resulting system can be optimized with respect to some cost function. Minimizing area or power consumption are examples of such cost functions. The variants in the system are the different ASIP configurations that can be used to execute these standalone tasks. Optimization of pipelined systems has been addressed in [1, 2, 3, 4] with the help of standard techniques such as Integer Linear Programming and various heuristics. However, these techniques require runtime estimates to evaluate alternative design choices, which are typically obtained through cycle accurate simulations. The estimation methods shown in this paper minimize the use of cycle accurate simulations to reduce simulation time, which enables the quick availability of such runtime estimates. The following section describes the two runtime estimation methods in detail.

4 RUNTIME ESTIMATION METHODS

We target ASIP based pipelined MPSoC systems in this paper. Each ASIP in the pipelined MPSoC has a number of configurations. A pipelined MPSoC can be implemented using one of a possible combination of the ASIP configurations.

Let us examine how the pipelined MPSoC works through the example shown in Figure 4.1. Annotations around each processor show the iterations of the task being run on that processor, the latency of each iteration, and the number of bytes transferred in each iteration of the task. For example, (10, 500, 64) means the task is repeated ten times, while latency of each iteration is 500 clock cycles and 64 bytes are transferred in each iteration. Since the last processor is writing out to file, 0 bytes will be transferred. Assuming that there are no stalls between the processors and a byte transfer takes a single clock cycle, the latency of the first processor for each iteration will be 564 clock cycles.

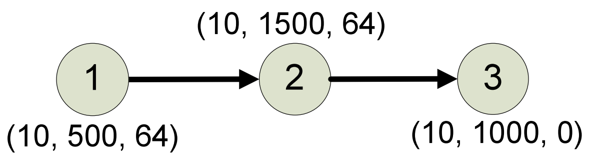


Figure 4.1: Pipelined MPSoC Example

The first iteration of each processor corresponds to the filling of the pipeline. Thus, time to fill the pipeline in Figure 4.1 is 3,256 clock cycles, where processor 1 accounts for 564 clock cycles (500 + 64), processor 2 accounts for 1,628 clock cycles (64 + 1500 + 64), and processor 3 accounts for 1,064 clock cycles (64 + 1000). After 3,256 clock cycles, the first output is available. Subsequent outputs from the pipelined system will be available after every 1,628 clock cycles, as processor 2 is the critical processor in this example. The stalls on processor 1 and processor 3 are hidden in the latency of processor 2, and thus these stalls need not be considered. However, for the stalls to be hidden in the latency of the critical processor (processor 2), we assume that the buffers between the processors are able to accommodate the output of at least one iteration. For example, 64 bytes are transferred between processor 1 and 2, thus the size of the FIFO should be at least 64 bytes. Otherwise, processor 2 (which is the critical processor) will be stalled due to the limited data space in the FIFO, and the critical processor should not be stalled due to the non-critical processors. Thus, assuming the availability of sufficiently sized buffers, the total time will be $3,256 + (10 - 1) \times 1,628 = 17,908$ clock cycles. This simple concept is extended to derive an estimation equation which uses latencies of each ASIP to estimate the runtime of a pipelined MPSoC.

The code of each standalone task on an ASIP in the pipelined system is divided into three portions. The first portion refers to the non-kernel tasks performed before the kernel operations start. Thus, it is named initialization time. The second portion of code refers to the kernel iterations and the third portion refers to the code to execute the final non-kernel operations. The time taken to execute the first iteration of the kernel is named 'First Latency' (FL) while the time taken to execute the rest of the kernel iterations is named the 'Average Latency' (AL). AL is averaged over the total number of kernel iterations, except the first one which is referred to as FL. Due to cold cache start, there will be more instruction and data cache misses during the first iteration (FL) compared with the second iteration. Thus, FL will be significantly higher than AL. Hence, separating FL from AL makes the runtime calculation more accurate. The time taken to execute the third portion is named finalization time. The runtime of a pipelined system can then be calculated as follows:

$$\mathbb{R} = \mathrm{R}^{init}(s_1) + \sum_{i=1}^{M} \mathrm{L}^1(s_i) + (I - 1) \times \mathrm{L}(s_{critical}) + \mathrm{R}^{final}(s_M) \quad (4.1)$$

$$\begin{aligned} where \quad \mathrm{R}^{init}(s_1) &= \max_{1 \le j \le N_0} \{\mathrm{R}^{init}(p_{1,j})\} \\ \mathrm{R}^{final}(s_M) &= \max_{1 \le j \le N_M} \{\mathrm{R}^{final}(p_{M,j})\} \\ \mathrm{L}^1(s_i) &= \max_{1 \le j \le N_i} \{\mathrm{L}^1(p_{i,j})\} \\ \mathrm{L}(p_{i,j}) &= \textstyle\sum_{h=2}^{I} \mathrm{L}^h(p_{i,j})/(I-1) \\ \mathrm{L}(s_i) &= \max_{1 \le j \le N_i} \{\mathrm{L}(p_{i,j})\} \end{aligned}$$

$R^{init}(p_{i,j})$, $R^{final}(p_{i,j})$ and $L^h(p_{i,j})$ refer to initialization time, finalization time and latency of h-th iteration of processor j in pipeline stage i respectively. Thus, $L^1(p_{i,j})$ and $L(p_{i,j})$ refer to FL and AL of processor j in pipeline stage i respectively. In the equation, s_i stands for pipeline stage i, while $s_{critical}$ refers to the critical stage. The critical stage has the worst AL amongst all the stages in the pipelined system. *I,* N_i and M refers to the number of iterations of the application's kernel, the number of processors in the pipeline stage i and the total number of pipeline stages respectively.

The equation generalizes the concept of runtime calculation of the example shown in Figure 4.1. The runtime of the pipelined system is calculated by summing up the various factors that contribute to the runtime. It sums up initialization time of the first stage ($\mathrm{R}^{init}(s_1)$), the time to fill the empty pipeline ($\sum_{i=1}^{M} \mathrm{L}^1(s_i)$), after initial filling of the pipeline, the time spent by the critical stage ($(I-1) \times \mathrm{L}(s_{critical})$), and finally the finalization time of the last stage ($\mathrm{R}^{final}(s_M)$). FL is used to calculate the time to fill the pipeline while AL is used for the critical stage, because the pipeline is already filled. Only the initialization time of the first stage and the finalization time of the last stage are used in the equation. This is because the first stage is responsible for initialization while the last stage performs the epilogue operations. The max functions in the equation are used to account for the parallel pipeline stages, that is, the stages with more than one processor in parallel, as the latency of the processor with the worst latency in the parallel pipeline stage will hide the latency of other processors in that stage. The latencies of each processor used in this equation include the computation and net communication time of each processor, omitting the communication stalls. The communication stalls of the processors do not contribute to the runtime of the pipelined system when sufficiently sized buffers are used. This is because slower processors stall for the critical processor in the pipelined system and the communication stalls are hidden in the latency of the critical processor. For streaming applications, amount of data to be transferred between processors is usually known a priori, and thus buffer sizes can be computed at design time. For example, for a typical JPEG encoder application, processing is done at macroblock level. Thus, assuming that 8×8 macroblock is processed in each iteration, and each pixel in a macroblock is represented with 24 bits (8 bits for each of R, G and B components), buffer size will be $8 \times 8 \times 24 = 1536$ bytes. It should be noted that use of non-sufficient buffers will degrade the performance and hence will compromise the throughput of a pipelined MPSoC. Thus, we believe that it is reasonable to assume the availability of sufficiently sized buffers. In addition, Equation 4.1 assumes that the FIFOs are implemented as dedicated hardware buffers, and separate instruction and data memories are used for each processor.

4.1 Method One

Given Equation 4.1, the runtime of a pipelined MPSoC with one combination of ASIP configurations can be calculated with the aid of the latencies of the individual ASIP configurations. Thus, there is no need for cycle accurate simulation of the whole pipelined MPSoC with that particular combination of ASIP configurations. However, a methodology to obtain the latencies of each ASIP configuration is required. To record the different execution times of each ASIP configuration, the pipelined system is simulated with the first available configuration of each ASIP. Next time, the next available configuration of each ASIP is used to simulate the pipelined system. In this setting, all the ASIP configurations are simulated at least once, by only running K_{max} simulations of the pipelined system, where K_{max} is the maximum number of configurations for an ASIP from amongst all the ASIPs in the pipelined system. For example, assume that the three processor pipelined MPSoC in Figure 4.1 has 10, 20, and 15 configurations for processor 1, 2 and 3 respectively. In the brute force method, 3,000 (10×20×15) simulations are required. However, with our methodology, only K_{max} = 20 simulations are required as illustrated in Figure 4.2. The results obtained from the simulation of the pipelined system with one set of ASIP configurations are used to record the different latencies of each ASIP configuration used in that particular simulation. This is possible because the computation time of an ASIP configuration is separated from its inter-processor communication stalls. These recorded values can then be used to calculate the runtime of the pipelined system for any combination of the ASIP configurations using Equation 4.1.

System Simulation	P1's Configuration	P2's Configuration	P3's Configuration
1	1	1	1
2	2	2	2
⋮	⋮	⋮	⋮
9	9	9	9
10	10	10	10
11	10	11	11
⋮	⋮	⋮	⋮
20	10	20	15

Figure 4.2: Pipelined MPSoC simulation methodology for the example shown in Figure 4.1

4.2 Method Two

In the last subsection, we were able to reduce the number of simulations of the pipelined system to only K_{max} simulations, which provides significant speed up in the simulation time. However, all the ASIP configurations still need to be simulated at least once to record their latencies. These cycle accurate simulations may increase as the number of ASIP configurations increase, hence limiting the maximum design space that can be targeted. Thus, in this section, we propose another estimation method which can be used to estimate the runtime of a task on an ASIP configuration, minimizing the use of cycle accurate simulations.

ASIP configurations differ by the addition of extensible instructions, functional units and instruction and data cache configurations. The additional instructions and/or functional units are combined with the base instruction set to generate a new ISA. These new ISAs can be combined with different instruction and data cache configurations. For example, the total number of configurations for an ASIP with 10 ISAs and 40 cache configurations will be 400. The idea here is to simulate some configurations of an ASIP to extract performance parameters, which are then used to predict the runtime for the other configurations of the same ASIP.

The runtime of a task being executed on a processor can be broken down into two parts: the time to fetch the instructions and data, t_f; and the net time to execute the fetched instructions, t_{ne}. The fetching time of instructions and data depends on the memory hierarchy of the processor. The time to execute the fetched instructions depends on the underlying microarchitecture, data dependency and the total number of instructions in the program. We assume a processor with a classical 5-stage pipeline, in-order issue, separate L1 instruction and data caches, and separate instruction and data memories (local memories of each processor in the pipelined MPSoC). A write-through cache policy is assumed.

Using this model, Equation 4.2 can be used to estimate the runtime of a task on an ASIP configuration. L_{IM} refers to instruction memory read latency while L_{IH} is the latency to read an instruction from instruction cache in case of instruction hit. L_{DMR} and L_{DMW} refer to data memory read latency and data memory write latency respectively. $C_{IM}, C_{IH}, C_{DMR}, C_{DMW}$ represent instruction cache miss count, instruction cache hit count, data cache read miss count and data cache write miss count respectively.

$$\begin{aligned} \mathbb{R} &= t_f + t_{ne} \qquad (4.2)\\ &= (1 + L_{IM}) \times C_{IM} + L_{IH} \times C_{IH} \\ &\quad +(1 + L_{DMR}) \times C_{DMR} + (1 + L_{DMW}) \times C_{DMW} \\ &\quad +\mathit{NCPI} \times N_I \end{aligned}$$

The first four factors provide an estimate of the memory fetch time for both instruction and data in the whole program. In a typical 5 stage pipeline processor, instructions are fetched at stage 1 (Instruction Fetch stage), while data fetches are processed at stage 4 (Memory stage). Thus, instruction and data fetches can be overlapped. The two factors, $(1 + L_{IM}) \times C_{IM}$ and $L_{IH} \times C_{IH}$, account for instruction fetching in case of instruction cache miss and hit. The instruction miss latency, L_{IM}, is added with one to account for the clock cycle needed to access the instruction cache to check for cache hit or miss. The data hits are not included in the equation, because we assume data hits will be overlapped with the instruction hit or miss latency due to the pipeline in the processor. However, the data misses may not be perfectly overlapped with instruction hits and misses. Furthermore, the latency to fetch data in case of a miss may be different from instruction miss latency as separate memories are used. Hence, data misses are included in the runtime estimation. To make the estimation more accurate, the data misses for read and write are included separately as $(1 + L_{DMR}) \times C_{DMR}$ and $(1 + L_{DMW}) \times C_{DMW}$ in the equation.

Once the instruction and data fetch time is obtained, the rest of the time is due to the execution of the fetched instructions. The last factor calculates the net time to execute all the instructions by multiplying the net CPI ($NCPI$) by the total number of instructions (N_I) in the program. Note that $NCPI$ is not the actual CPI of the processor, but it is the net time to execute the instructions once they have been fetched and the corresponding data has been fetched as well. Thus, in this equation, $NCPI$ accounts for the overlapping between the data misses and instruction hits and misses, and the stalls caused due to data dependencies in the program. This value remains fairly constant for a given program on a given ISA with different cache configurations. The effect of different cache configurations is taken into account by the instruction and data caches' hit and miss counts. The major reason for fluctuations in the value of NCPI across the same ISA, but with different cache configurations, will be due to the effects of overlapped fetches of missed data and instructions. To accurately model such overlapped fetches, one needs to perform a cycle accurate simulation or use data flow analysis techniques to extract data dependencies and then estimate the runtime. However, to keep the model simple, these details are omitted, but the results show that the model is still quite accurate in predicting the runtime of a particular program on a given processor configuration (Section 6). This is because runtime of an application is mostly dominated by the time spent in fetching instructions and data from the memory, and that has been modeled in Equation 4.2. The rest of the hardware events are taken into account by the $NCPI$ parameter.

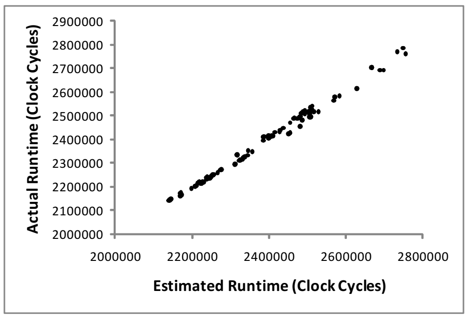
Given instruction hit and miss counts, data read and write miss counts, and $NCPI$ of a given program for one particular ISA and a cache configuration, the runtime of the program can be estimated using Equation 4.2. However, the equation assumes that the value of $NCPI$ and the cache statistics are available. Any of the tools presented in [8, 9, 13] can be used to obtain the cache statistics for all the different cache configurations. To find the value of $NCPI$, Equation 4.2 can be rearranged as:

$$NCPI = \frac{\mathbb{R} - t_f}{N_I} \tag{4.3}$$

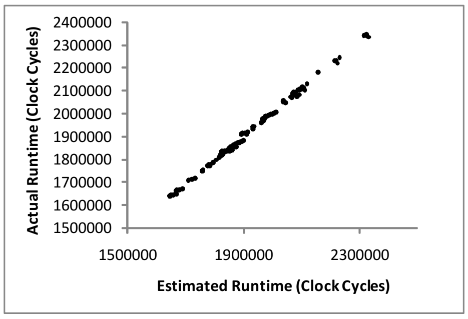
Using the actual runtime of a program on a given ISA and a cache configuration, and the cache hit and miss counts for that particular cache configuration (to calculate the value of t_f), the value of NCPI can be calculated. Several cycle accurate simulations can be run to calculate NCPI values for an ISA with several cache configurations, in order to calculate an average NCPI value for that particular ISA. The average NCPI value can then be used for the rest of the cache configurations, with their given hit and miss counts to predict the runtime for those cache configurations with the same ISA.

As part of preliminary analysis, the results of runtime estimation for the ASIP in stage 1 of each benchmark (shown in Figure 3.1) is presented here. For JPEGEnc1, the first processor was responsible for reading the raw image and performing 'color space conversion' from RGB to YCbCr. We had 4 different ISAs and changed instruction and data cache sizes from 1KB to 32KB. Thus, in total 144 configurations were created for this ASIP. We only simulated each ISA with identical instruction and data cache sizes from 1KB to 32KB. Thus, ISA 1 with both 1KB instruction and 1KB data cache, ISA 1 with 2KB instruction and 2KB data cache, and so on was simulated. As a result, 6 cache configurations from ISA 1 were simulated. The choice for identical instruction and data cache sizes is based on the analysis in [15], which shows that an application's performance on baseline configurations is the most significant predictor for its performance on other processor configurations. The rest of the ISAs were also simulated with only 6 cache configurations. In total, 24 simulations were performed to calculate average NCPI values for each of the 4 ISAs. Using these NCPI values, the runtimes for the rest of the 120 processor configurations were predicted. Figure 4.3(a) shows the actual runtime plotted against the estimated runtime. The linear relationship between the estimated runtime and the actual runtime suggests that they are closely correlated. The maximum estimation error across all 144 configurations was 1.36%, while the average estimation error was 0.46% only. Despite the simplicity of the model, these results show that the estimations are quite accurate. It should be noted that very basic information of the processor ISA is used, which means that the same equation can be used for other processors, given the latencies of the memory hierarchy are known. The savings in terms of simulation time were significant. It took 11 hours to simulate all 144 configurations. The simulation time for 24 configurations was 2 hours, including the time to obtain the cache statistics for all the different cache configurations. This speed up will be further improved when more cache configurations are considered.

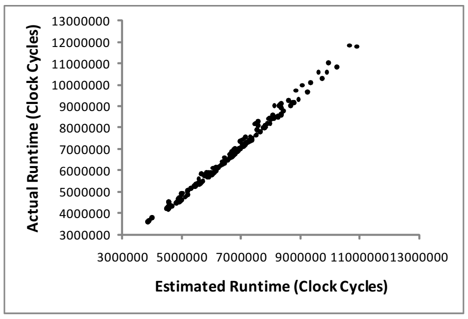
Same methodology was used for the ASIP in stage 1 of JPEGEnc2, JPEGDec and MP3Enc as well, and the results are shown in Figure 4.3(b), 4.3(c) and 4.3(d) respectively. The details of the number of configurations, and maximum and minimum errors



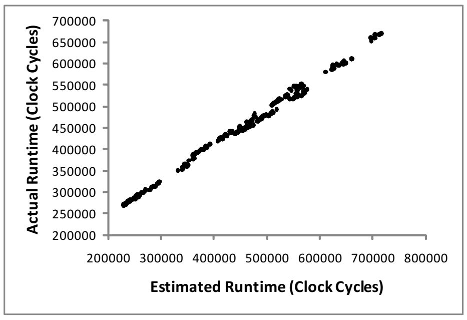
(a) JPEGEnc1



(b) JPEGEnc2



(c) JPEGDec



(d) MP3Enc

Figure 4.3: Preliminary analysis of runtime estimation using Equation 4.2 (Note that both the axes start at the same value in all the figures)

are reported in Section 6. However, in these graphs, the linear relationship between actual runtime and estimated runtime illustrates close correlation. Furthermore, the graphs are well spread out, signifying the fact that the proposed analytical equation predicted the runtime with reasonable error (see Section 6) for a wide range of ASIP configurations. The analysis on all the other ASIPs of each example benchmark of Figure 3.1 revealed similar findings.

5 EXPERIMENTAL SETUP

To evaluate the proposed estimation methodology, we used a commercial processor, Xtensa LX2 [5] to create the pipelined MPSoCs presented in Figure 3.1. The Xtensa LX2 family of processors provides an extensible processor platform for creation of ASIP configurations, and comes with the Xtensa RB-2007.1 toolset which includes C/C++ compiler, Instruction Set Simulator (ISS), Xtensa PRocessor Extension Synthesis (XPRES) and XTensa Modeling Protocol (XTMP).

XPRES is used to generate processor configurations directly from the C/C++ code, for a given base processor. XPRES analyzes the C code and automatically generates application specific additional instructions, which may consist of a combination of fused operations, FLIX instructions [27], specialized operations [28] and vector operations. XPRES can also generate different sets of additional instructions, reflecting different ISAs for the given application. These additional instructions are output in Tensilica Instruction Extension (TIE) language, and are compiled through TIE compiler for seamless integration, as C/C++ compiler will automatically incorporate the new instructions without the need for modification of the C code.

ISS is used to simulate a given processor configuration cycle accurately. ISS produces profiling data such as total clock cycles, cache statistics, etc. XTMP is a multiprocessor simulation environment, which is used to instantiate multiple processors, and to connect them in a pipelined fashion using queues. These queues implement a FIFO protocol. FIFO interface for each processor in XTMP includes POP and $PUSH$ functions. These functions are used by the connected processors to read from and write to the FIFO. A pop from an empty FIFO and a push to a full FIFO stalls the processor. These stall cycles are recorded as global stalls, which can be used to calculate net communication time of each processor. Using ISS, XTMP generates clock cycle information for the pipelined system. This information is used to record different latencies of each processor in the pipelined system.

For cache statistics, we used the tool from [8] which uses a trace based simulation methodology. For a given trace, the tool outputs the cache hit and miss counts for different instruction and data cache configurations. Parameters for cache configurations include cache size, associativity, and line size.

All the experiments were conducted on a quad core machine running at 2.15 GHz with 8Gb RAM.

6 RESULTS & ANALYSIS

The results are presented in two parts. First, we present the results of the processor estimation technique (Equation 4.2). Second, we compare the effectiveness of the runtime estimation techniques for the pipelined system with the actual cycle accurate system simulation in XTMP.

The benchmarks shown in Figure 3.1 are used for all the experiments in this paper, which were created manually (adhering to the standards of JPEG encoder, JPEG decoder and MP3 Encoder). Table 6.1 shows the number of ASIP configurations for each of the ASIPs in the pipelined implementation of each benchmark. Columns 2-5 show the names of the benchmarks, while each row represents the number of ASIP configurations for a given benchmark in a particular stage. For example, row 1 shows that $4 \times 36 = 144$ configurations are available for the ASIP in stage 1 of JPEGEnc1 benchmark. The first number, 4 in this case, is the number of different ISAs while the second number, 36 in this case, is the number of cache configurations for each of the ISA. In case of JPEGDec, in stage 2, the three numbers show the ASIP configurations for each of the three ASIPs in that stage. Since JPEGDec had only three pipeline stages, rows 4-6 contain no data. Both the instruction and data cache sizes were changed from 1KB to 32KB, accounting for 36 cache configurations. This setting was used to generate a reasonable number of configurations for each processor (and is not a limitation of our approach). Associativity of the caches could also have been changed, increasing the processor configurations further.

Stage	**JPEGEnc1**	**JPEGEnc2**	**JPEGDec**	**MP3Enc**
1	4×36	5×36	8×36	7×36
2	4×36	5×36	8×36, 8×36, 8×36	7×36, 7×36
3	11×36	7×36, 7×36, 7×36	7×36	9×36
4	4×36	7×36	-	9×36
5	7×36	4×36	-	-
6	4×36	-	-	-

Table 6.1: ASIP Configurations

Table 6.1 suggests that the number of cache configurations is typically greater than the different ISAs available for an ASIP. Thus, increase in cache configurations will explode the total number of ASIP configurations. This will increase the simulation time as cycle accurate simulations are slow. Thus, the effectiveness of Equation 4.2 can be seen here. For each different ISA available for an ASIP, we simulated it cycle accurately with only 6 different cache configurations (those with equal instruction and data cache sizes) as explained in Section 4.2. Using the results from these initial simulations, the NCPI value for each ISA is calculated. During these cycle accurate simulations, separate trace files are also generated for each ISA. These traces are then used by the cache simulator to produce cache hit and miss counts for all the cache configurations. Using the cache statistics and the calculated NCPI values, runtime for the rest of the ASIP configurations is predicted.

Table 6.2 shows the results of the prediction using Equation 4.2. The columns show the benchmark, the processor number in that benchmark, and the average error and the maximum error in runtime estimation for each processor respectively. For example, row 10 shows the results for the first ASIP in stage 3 of JPEGEnc2 benchmark, thus named 'p3a'. The error in runtime estimation using Equation 4.2 is calculated by comparing the estimated runtimes with the cycle accurate runtimes obtained through simulation. The average error shown in the table is calculated from error in runtime estimation for all the configurations of an ASIP. For example, Table 6.1 shows that 144 configurations are used for p1 of JPEGEnc1. Thus, the average is calculated over all these processor configurations for p1 of JPEGEnc1. The average error in estimation

Benchmark	Processor	Avg. Error (%)	Max. Error (%)
JPEGEnc1	p1	0.46	1.36
	p2	0.15	0.50
	p3	0.23	3.06
	p4	0.70	0.73
	p5	0.37	1.74
	p6	0.48	2.15
JPEGEnc2	p1	0.46	1.20
	p2	0.16	0.96
	p3a	0.80	3.53
	p3b	0.83	4.00
	p3c	0.83	4.00
	p4	0.58	3.17
	p5	0.34	1.38
JPEGDec	p1	3.83	9.94
	p2a	6.05	14.79
	p2b	7.15	13.91
	p2c	6.09	14.23
	p3	0.71	3.07
MP3Enc	p1	5.56	15.02
	p2a	3.07	8.07
	p2b	2.80	9.13
	p3	2.39	11.49
	p4	0.86	4.23

Table 6.2: Error in Runtime Estimation using Equation 4.2

for all the processors used in our benchmarks is less than 7.15%, while the maximum error is less than 15.02% as highlighted in Table 6.2. This shows that the estimation method proposed in Equation 4.2 is reasonably accurate, and can be used to estimate the runtime of a given task on an ISA with a cache configuration by utilizing the cache statistics, and the NCPI parameter only. This eliminates the need for cycle accurate simulations of all the ASIP configurations.

Now, we show the effectiveness of Equation 4.1 in estimating the runtime of the whole pipelined system. Due to large design space (at least 10^{10} design points) for each benchmark, the estimated runtime for all the design points cannot be compared with the actual cycle accurate runtimes (obtained through XTMP). Thus, we compare K_{max} design points for each of the benchmarks, where $K_{max} = 396, 252, 288$ and 324 for JPEGEnc1, JPEGEnc2, JPEGDec and MP3Enc respectively (from Table 6.1). These design points include all the ASIP configurations at least once. More points can be compared at the cost of increased simulation time.

Two methods can be used to estimate the runtime of the pipelined system using Equation 4.1. Method 1 (Section 4.1) uses the latencies of the individual ASIP configurations, which are obtained through cycle accurate simulations. Method 2 (Section 4.2), on the other hand, uses Equation 4.2 to estimate the latencies of the individual ASIP

Benchmark	Method 1		Method 2	
	Avg. Error (%)	Max. Error (%)	Avg. Error (%)	Max. Error (%)
JPEGEnc1	2.28	5.91	5.00	9.11
JPEGEnc2	0.69	2.16	5.91	11.41
JPEGDec	0.21	1.29	5.09	13.21
MP3Enc	3.83	6.96	2.53	10.37

Table 6.3: Error in Runtime Estimation using Equation 4.1

configurations, reducing the number of cycle accurate simulations. Obviously method 2 will be faster than method 1, at the cost of less accurate estimation. Table 6.3 illustrates the results for the pipelined system's runtime estimation using both methods. The three major columns represent the benchmark, runtime estimation using method 1 and 2 respectively. For JPEGEnc1, the average estimation error is 2.28% and the maximum error is 5.91% using method 1. In method 2, both the average and maximum error increased to 5.00% and 9.11% respectively. This increase in error is expected as estimated runtimes of the individual processors are used in method 2, where as cycle accurate simulations are used in method 1. The reason for a decrease in the average error of method 2 compared to method 1 for MP3Enc benchmark in Table 6.3 is that all the design points are not used for the calculation of average error. The 324 design points which we used may have resulted in combinations of ASIP configurations that reduced the average error. As explained before, more design points could have been used for the calculation of average and maximum errors at the cost of more cycle-accurate simulations. To summarize, in all the benchmarks, the worst average error and maximum error for method 2 is 5.91% and 13.21% respectively as highlighted in Table 6.3.

The advantage in method 2 is the reduction in simulation time (due to reduced number of simulations), which is shown in Table 6.4. The second column shows the total number of design points (all possible combinations of the ASIP configurations – obtained through Table 6.1) for each benchmark. The third column, titled 'Pure Simulation', shows that it is infeasible (requiring many years) to simulate all the design points in the pipelined system to record their accurate runtime. Columns 4 and 5 show the simulation time needed for method 1 and 2 respectively. Using method 1, the simulation results can be obtained within a day. However, as the design space grows with an increase in ASIP configurations or a complex benchmark such as MP3Enc is used, method 1 will require days to obtain the simulation results, making it an impractical choice. The number of required simulations are reduced in method 2 with the help of Equation 4.2, in turn reducing the simulation time. The results show significant speed up for method 2 (column 5) when compared with method 1 (column 4) in Table 6.4,

Benchmark	Design Space	Pure Simulation	Method 1	Method 2
JPEGEnc1	4.2×10^{13}	$\sim$ yrs	19 hrs	1.8 hrs
JPEGEnc2	2.35×10^{16}	$\sim$ yrs	15 hrs	2 hrs
JPEGDec	1.73×10^{12}	$\sim$ yrs	13 hrs	1.5 hrs
MP3Enc	1.68×10^{12}	$\sim$ yrs	2 days	16 hrs

Table 6.4: Simulation Time Results

Benchmark	**Method 1**		**Method 2**	
	Avg. Error (%)	Max. Error (%)	Avg. Error (%)	Max. Error (%)
JPEGEnc1	2.89	6.55	4.47	8.48
JPEGEnc2	1.06	2.56	5.62	11.01
JPEGDec	0.36	1.56	4.97	12.90
MP3Enc	19.60	23.03	15.15	22.53

Table 6.5: Error in Runtime Estimation using the Equation proposed in [2]

reducing the simulation time to only 16 hours.

We also compared the estimation error of Equation 4.1 with the estimation error of the equation proposed in [2], which will be referred to as Shee's equation. Shee's equation uses the initialization time of the first stage, finalization time of the last stage and the critical stage's average latency to calculate the runtime of a pipelined system, ignoring first latencies which correspond to filling of the pipeline. The estimation error results for Shee's equation are shown in Table 6.5 in the same format as Table 6.3. A comparison of the second major column of Table 6.5 with the second major column of Table 6.3 shows that the error introduced by Shee's equation is slightly higher than Equation 4.1, except for MP3Enc benchmark. Since Shee's equation considers steady state of the pipeline, the predicted runtime will be close to Equation 4.1's prediction when the number of iterations is large and first latencies have low magnitude. For MP3Enc benchmark, the first latencies had high magnitudes because of the complexity of the encoding process and number of iterations was small, resulting in a significant increase in the estimation error (19.60% and 23.03%) as highlighted in Table 6.5. Except for MP3Enc benchmark, the error results for method 2 in Table 6.5 (Shee's equation) showed a slight unexpected decrease for both average and maximum error when compared to Table 6.3. This is because only a limited number of design points were used for such calculations due to slow cycle-accurate system simulation and huge size of the design spaces (at least 10^{10} design points). This also explains the decrease in both average and maximum errors of method 2 compared to method 1 for MP3Enc benchmark, in contrast to an obvious increase for the other benchmarks.

6.1 Further Discussion

An estimation equation can be evaluated by calculating its absolute accuracy and/or by calculating its fidelity. In absolute accuracy, absolute error of each estimated design point is calculated from the corresponding actual design point, and then averaged over all the design points. On the other hand, fidelity measures the correlation between the ordering of the estimated points and the ordering of the actual design points. An estimation equation with 0% absolute error will also have a fidelity of 100%. In some cases, such as design space exploration, ordering of the estimated design points is more important than their absolute errors for proper guidance of the exploration algorithms [29, 30].

Table 6.5 reported average and maximum error of 19.60% and 23.03% respectively for MP3Enc benchmark when using Shee's equation for method 1. These values suggest that all the estimated design points had almost the same estimation error, and hence most likely to be in the same order as the actual design points. On the other hand, the estimation error results for Equation 4.1 in Table 6.3 show that the average error (3.83%) is almost half the maximum error (6.96%) for MP3Enc benchmark. This suggests that some design points had higher estimation error than the others, increasing the probability of disordering the estimated points with respect to actual design points. Thus, even though Equation 4.1 demonstrated higher absolute accuracy compared to Shee's equation, it may not have higher fidelity. We aim to measure the fidelity of the proposed analytical equations in future to further validate their accuracy.

7 CONCLUSION

In this paper, two methods are presented to estimate the runtime of a pipelined MPSoC for a given combination of ASIP configurations. Brute force simulation is infeasible due to the size of the design space, which is at least 10^{10} design points. The presented estimation methods reduce the number of simulations by using analytical estimation equations. Thus, these methods can be used to speed up the process of acquiring performance measures, so that even larger design spaces can be explored. Our results show that the worst estimation error is 13.21% in all the benchmarks for both the estimation methods, while the simulation time is reduced from years to only several hours.

Bibliography

[1] S. L. Shee, A. Erdos, and S. Parameswaran, "Heterogeneous multiprocessor implementations for jpeg:: a case study," in *CODES+ISSS '06: Proceedings of the 4th international conference on Hardware/software codesign and system synthesis*, (New York, NY, USA), pp. 217–222, ACM, 2006.

[2] S. L. Shee and S. Parameswaran, "Design methodology for pipelined heterogeneous multiprocessor system," in *DAC '07: Proceedings of the 44th annual conference on Design automation*, (New York, NY, USA), pp. 811–816, ACM, 2007.

[3] H. Javaid and S. Parameswaran, "Synthesis of heterogeneous pipelined multiprocessor systems using ilp: jpeg case study," in *CODES/ISSS '08: Proceedings of the 6th IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis*, (New York, NY, USA), pp. 1–6, ACM, 2008.

[4] H. Javaid and S. Parameswaran, "A design flow for application specific heterogeneous pipelined multiprocessor systems," in *DAC '09: Proceedings of the 46th Annual Design Automation Conference*, (New York, NY, USA), pp. 250–253, ACM, 2009.

[5] Tensilica, "Xtensa Customizable Processor." http://www.tensilica.com.

[6] Altera, "Nios Processor." http://www.altera.com.

[7] ARC, "ARC 600 and 700 Core Families." http://www.arc.com.

[8] M. S. Haque, A. Janapsatya, and S. Parameswaran, "Susesim: a fast simulation strategy to find optimal l1 cache configuration for embedded systems," in *CODES+ISSS '09: Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, (New York, NY, USA), pp. 295–304, ACM, 2009.

[9] J. Edler and M. D. Hill, "Dinero iv trace-driven uniprocessor cache simulator." http://www.cs.wisc.edu/markhill/DineroIV/, 2004.

[10] M. Yourst, "PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator," in *Performance Analysis of Systems & Software, 2007. ISPASS 2007. IEEE International Symposium on*, pp. 23–34, April 2007.

[11] ARM, "RealView ARMulator ISS." http://www.arm.com.

[12] R. Srinivasan, J. Cook, and O. Lubeck, "Performance modeling using monte carlo simulation," *Computer Architecture Letters*, vol. 5, pp. 38–41, Jan.-June 2006.

[13] N. Tojo, N. Togawa, M. Yanagisawa, and T. Ohtsuki, "Exact and fast l1 cache simulation for embedded systems," in *ASP-DAC '09: Proceedings of the 2009 Conference on Asia and South Pacific Design Automation*, (Piscataway, NJ, USA), pp. 817–822, IEEE Press, 2009.

[14] L. Singleton, C. Poellabauer, and K. Schwan, "Monitoring of cache miss rates for accurate dynamic voltage and frequency scaling," in *Proceedings of the Multimedia Computing and Networking Conference (MMCN)*, 2005.

[15] B. C. Lee and D. M. Brooks, "Accurate and efficient regression modeling for microarchitectural performance and power prediction," in *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, (New York, NY, USA), pp. 185–194, ACM, 2006.

[16] P. Joseph, K. Vaswani, and M. Thazhuthaveetil, "Construction and use of linear regression models for processor performance analysis," in *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*, pp. 99–108, Feb. 2006.

[17] T. S. Karkhanis and J. E. Smith, "A first-order superscalar processor model," *SIGARCH Comput. Archit. News*, vol. 32, no. 2, p. 338, 2004.

[18] T. Kodaka, K. Kimura, and H. Kasahara, "Multigrain parallel processing for jpeg encoding on a single chip multiprocessor," in *IWIA '02: Proceedings of the International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems (IWIA'02)*, (Washington, DC, USA), p. 57, IEEE Computer Society, 2002.

[19] S. Banerjee, T. Hamada, P. Chau, and R. Fellman, "Macro pipelining based scheduling on high performance heterogeneous multiprocessor systems," *Signal Processing, IEEE Transactions on*, vol. 43, no. 6, pp. 1468–1484, 1995.

[20] J. Jeon and K. Choi, "Loop pipelining in hardware-software partitioning," in *Asia and South Pacific Design Automation Conference*, pp. 361–366, 1998.

[21] J. DeSouza-Batista and A. Parker, "Optimal synthesis of application specific heterogeneous pipelined multiprocessors," *Application Specific Array Processors, 1994. Proceedings., International Conference on*, pp. 99–110, 22-24 Aug 1994.

[22] S.-R. Kuang, C.-Y. Chen, and R.-Z. Liao, “Partitioning and pipelined scheduling of embedded system using integer linear programming,” in *ICPADS ’05: Proceedings of the 11th International Conference on Parallel and Distributed Systems - Workshops (ICPADS’05)*, (Washington, DC, USA), pp. 37–41, IEEE Computer Society, 2005.

[23] S. Bakshi and D. D. Gajski, “Partitioning and pipelining for performance-constrained hardware/software systems,” *IEEE Trans. VLSI Syst.*, vol. 7, no. 4, pp. 419–432, 1999.

[24] I. Karkowski and H. Corporaal, “Design of heterogenous multi-processor embedded systems: applying functional pipelining,” in *PACT ’97: Proceedings of the 1997 International Conference on Parallel Architectures and Compilation Techniques*, (Washington, DC, USA), p. 156, IEEE Computer Society, 1997.

[25] G. Kahn, “The semantics of a simple language for parallel programming,” pp. 471–475, 1974.

[26] E. A. Lee and D. G. Messerschmitt, “Synchronous Data Flow,” *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.

[27] Tensilica, “Flix: Fast relief for performance-hungry embedded applications.” http://www.tensilica.com/pdf/FLIX_White_Paper_v2.pdf, 2005.

[28] Tensilica, “XPRES Generated Specialized Operations.” http://tensilica.com/pdf/XPRES%201205.pdf, 2005.

[29] P.-K. Huang, M. Hashemi, and S. Ghiasi, “System-level performance estimation for application-specific mpsoc interconnect synthesis,” in *SASP ’08: Proceedings of the 2008 Symposium on Application Specific Processors*, (Washington, DC, USA), pp. 95–100, IEEE Computer Society, 2008.

[30] J. T. Russell and M. F. Jacome, “Architecture-level performance evaluation of component-based embedded systems,” in *DAC ’03: Proceedings of the 40th annual Design Automation Conference*, (New York, NY, USA), pp. 396–401, ACM, 2003.