Rapid Runtime Estimation Methods for Pipelined MPSoCs

Haris Javaid  Andhi Janapsatya  Mohammad Shihabul Haque  Sri Parameswaran
School of Computer Science and Engineering, University of New South Wales, Sydney, Australia
{harisj, andhij, mhaque, sridevan}@cse.unsw.edu.au

ABSTRACT

The pipelined Multiprocessor System on Chip (MPSoC) paradigm is well suited to the data flow nature of streaming applications. A pipelined MPSoC is a system where processing elements (PEs) are connected in a pipeline. Each PE is implemented using one of a number of processor configurations (configurations differ by instruction sets and cache sizes) available for that PE. The goal is to select a pipelined MPSoC with a mapping of a processor configuration to every PE. To estimate the runtime of a pipelined MPSoC, designers typically perform cycle-accurate simulation of the whole pipelined system. Since the number of possible pipelined implementations 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 estimates the runtime of the pipelined MPSoC, by performing cycle accurate simulations of individual processor configurations (rather than the whole pipelined system), and then utilizing an analytical model to estimate the runtime of the pipelined system. In the second method, runtimes of individual processor configurations are estimated using an analytical processor model (which uses cycle-accurate simulations of selected configurations, and an equation based on ISA and cache statistics). These estimated runtimes of individual processor configurations are then used to estimate the total runtime of the pipelined system. By evaluating our approach on three benchmarks, we show that the maximum estimation error is 5.91% and 16.45%, with an average estimation error of 2.28% and 6.30% for the first and second method respectively. The time to simulate all the possible pipelined implementations (design points) using cycle-accurate simulator is in the order of years, as design spaces with at least 10^11 design points are considered in this paper. However, the time to simulate all processor configurations individually (first method) takes tens of hours, while the time to simulate a subset of processor configurations and estimate their runtimes (second method) is only a few hours. Once these simulations are done, the runtime of each pipelined implementation can be estimated within milliseconds.

1. INTRODUCTION

Estimation is an important and critical part of most design exploration methodologies. Used in conjunction with exploration algorithms which search for some global minima, 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 thousands or even millions 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 [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 pipelined MPSoC, at a given instant, each stage processes different data in a pipelined fashion, thus providing high throughput. Furthermore, each processor in a stage is tuned according to specific tasks allocated to that stage. Hence, it is important to find a performance estimation methodology which is faster than pure cycle accurate simulation. In this paper, a performance estimation methodology for a pipelined MPSoC is presented. Two methods are shown in this paper. In the first method, an analytical equation is proposed which calculates the runtime of a pipelined MPSoC, utilizing the runtimes of individual ASIP configurations. These runtimes of individual ASIP configurations and the whole system 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 performance 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 \( X_i \) and \( Y_1 \) can be estimated using ISA information of \( X_i \) and cache statistics for \( Y_1 \). Thus, simulations are only performed with configurations \( X_1 \) with \( Y_1 \), \( X_2 \) with \( Y_1 \), until \( X_i \) 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 a tool named SuSeSim [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 \( X_1 \) 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 show that the estimation technique can significantly reduce simulation time, which will help system designers to explore larger design spaces. However, the penalty is paid in terms of the accuracy of the estimated runtime of the
systems. Clearly, the first method is more accurate than the second, but the second method reduces the simulation time significantly.

The rest of this paper is organized as follows. Section 2 will present an overview of existing methodologies for processor performance estimation and pipelined MPSoC estimation. Section 3 presents the underlying concepts of a pipelined MPSoC. Section 4 presents the two performance estimation methods. Section 5 and Section 6 presents 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 up is in the accuracy of the model. The authors of [12] proposed a Monte Carlo based model for predicting the performance of a single in-order processor. The model breaks down the execution time of a program 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 SuS- eSim [8], Dinko IV [9], and CRCE1 and CRCE2 [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 do not contain sufficient information to obtain the processor stalls that occur due to cache misses.

Singleton et al. [14] used 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] model 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 misprediction, 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.

For 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] are more focused on design of such systems instead of runtime estimation methods. All these works either propose less accurate estimation methods [2, 18] 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 (Section 4.2), which is the very first work of its kind in the context of pipelined MPSoCs.

2.1 Our Contribution

In this paper, we propose a performance estimation methodology for a pipelined MPSoC 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, 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 multiprocessor systems. A few are shown in Figure 1, where each node represents a processor in the pipelined system. We use the 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

<table>
<thead>
<tr>
<th>Stage</th>
<th>JPEG</th>
<th>JPEG</th>
<th>JPEG</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Enc1</td>
<td>Enc2</td>
<td>Dec</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 1: Benchmark Applications
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 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. Due to limited space, the names of standalone tasks are not shown.

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 often of each other, making it possible to execute them in a pipelined fashion. Thus, assum-

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:

\[
R = R^{init}(s_1) + \sum_{i=1}^{M} L^1(s_i) + (I - 1) \times L(s_{critical}) + R^{final}(s_M)
\]

where

\[
R^{init}(s_1) = \max_{1 \leq j \leq N_0} \{R^{init}(p_{1,j})\}
\]

\[
R^{final}(s_M) = \max_{1 \leq j \leq N_M} \{R^{final}(p_{M,j})\}
\]

\[
L(s_i) = \max_{1 \leq j \leq N_i} \{L^i(p_{i,j})\}
\]

\[
R^{init}(p_{1,j}), R^{final}(p_{j}), L^i(p_{i,j}) \text{ and } L(s_i) \text{ refer to initialization time, finalization time, FL and AL of processor } j \text{ in pipeline stage } i \text{ respectively. In the equation, } s_i \text{ stands for pipeline stage } i, \text{ while } s_{critical} \text{ refers to the critical stage. The critical stage has the worst AL amongst all the stages in the pipelined system. } I, N, \text{ and } M \text{ refers to the number of iterations of the application’s kernel, the number of processors in the pipeline stage } x \text{ and the total number of pipeline stages respectively.}
\]

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 2. 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 of the task. Thus, assuming that there are no stalls between the processors, the latency of the first processor for each iteration will be 564 clock cycles (assumes a byte transfer takes a single clock cycle).

The first iteration of each processor corresponds to the filling of the pipeline. Thus, time to fill the pipeline in Figure 2 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 there are enough buffers between the processors to accommodate the output of 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:

\[
R = R^{init}(s_1) + \sum_{i=1}^{M} L^1(s_i) + (I - 1) \times L(s_{critical}) + R^{final}(s_M)
\]

where

\[
R^{init}(s_1) = \max_{1 \leq j \leq N_0} \{R^{init}(p_{1,j})\}
\]

\[
R^{final}(s_M) = \max_{1 \leq j \leq N_M} \{R^{final}(p_{M,j})\}
\]

\[
L(s_i) = \max_{1 \leq j \leq N_i} \{L^i(p_{i,j})\}
\]

\[
R^{init}(p_{1,j}), R^{final}(p_{j}), L^i(p_{i,j}) \text{ and } L(s_i) \text{ refer to initialization time, finalization time, FL and AL of processor } j \text{ in pipeline stage } i \text{ respectively. In the equation, } s_i \text{ stands for pipeline stage } i, \text{ while } s_{critical} \text{ refers to the critical stage. The critical stage has the worst AL amongst all the stages in the pipelined system. } I, N, \text{ and } M \text{ refers to the number of iterations of the application’s kernel, the number of processors in the pipeline stage } x \text{ and the total number of pipeline stages respectively.}
\]

The equation generalizes the concept of runtime calculation of the example shown in Figure 2. 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 \(R^{init}(s_1)\), the time to fill the empty pipeline \(\sum_{i=1}^{M} L^1(s_i)\), after initial filling of the pipeline, the time spent by the critical stage \((I - 1) \times L(s_{critical})\), and finally the finalization time of the last stage \(R^{final}(s_M)\). 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 initial-ization 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 because slower processors stall for the critical processor in the pipelined system. The communication stalls of the processors do not contribute to the runtime of the pipelined system as the latency of the critical processor hides those stalls.

4.1 Method One

Given Equation 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. The 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 only 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 2 has 10, 20, and 15 configurations for processor 1, 2 and 3 respectively. In the brute force method, 3,000 $(10 \times 20 \times 15)$ simulations are required. However, with our methodology, only $K_{max} = 20$ simulations are required. 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.

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 speedup 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 2 can be used to estimate the runtime. $L_{IM}$ 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.

\[
    t_f = t_f + t_{ne} = (1 + L_{IM}) \times C_{IM} + L_{IH} \times C_{IH} + (1 + L_{DMR}) \times C_{DMR} + (1 + L_{DMW}) \times C_{DMW} + NCPI \times N_I
\]

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 instruction 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 their 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 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 2. However, the equation is not sensitive to the value of $N_{I}$ 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 2 can be rearranged as: $NCPI = \frac{t_f}{t_{ne}}$. 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.

To show the preliminary results, the analysis of the first ASIP in stage 1 of JPEGEncl benchmark shown in Figure 1 is presented here. This 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 sim-
parameters for cache configurations include cache size, associativity, stalls, which can be used to calculate net communication time of each processor. These stall cycles are recorded as global statistics. ISS produces profiling data such as total clock cycles, cache access statistics for all the different cache configurations. This speed up will be further improved when more cache configurations are considered.

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 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 XTens 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 [19], specialized operations [20] and vector operations. XPRES can also generate different sets of additional instructions, reflecting different ISAs for the given application.

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 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 1 are used for all the experiments in this paper, which were created manually (adhering to the standards of JPEG encoder and decoder). Table 1 shows the number of ASIP configurations for each of the ASIPs in the pipelined implementation of each benchmark. Columns 2-4 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 × 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 ISAs 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.

![Figure 3: Runtime Estimation for the first ASIP in stage 1 of JPEGEnc1 (Note that both the axes start at the same value)](image)

Table 1: ASIP Configurations

<table>
<thead>
<tr>
<th>Stage</th>
<th>JPEGEnc1</th>
<th>JPEGEnc2</th>
<th>JPEGDec</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4,36</td>
<td>5,36</td>
<td>8,36</td>
</tr>
<tr>
<td>2</td>
<td>4,36</td>
<td>5,36</td>
<td>8,36</td>
</tr>
<tr>
<td>3</td>
<td>11,36</td>
<td>7,36</td>
<td>7,36, 7,36</td>
</tr>
<tr>
<td>4</td>
<td>4,36</td>
<td>7,36</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>7,36</td>
<td>4,36</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>4,36</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Now, we show the effectiveness of Equation 1 in estimating the runtime of the whole pipelined system. Due to large design spaces (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, and 288 for JPEGEnc1, JPEGEnc2 and JPEGDec respectively (from Table 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 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 2 to estimate the latencies of the individual ASIP 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 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 4.89% and 10.26% 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. In all the benchmarks, the worst average error and maximum error for method 2 is 6.30% and 16.45% respectively, which occurred for JPEGDec.

The advantage of method 2 is the reduction in simulation time (due to reduced number of simulations), which is shown in Table 4. The second column shows the total number of design points (all possible combinations of the ASIP configurations – obtained through Table 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, 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 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 4, reducing the simulation time to only 2 hours.

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 (at least $10^{10}$ number of design points). The presented estimation methodologies reduce the number of simulations by using analytical estimation equations. Thus, these methodologies 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 16.45% in all the three benchmarks for both the estimation methods, while the simulation time is reduced from years to only 2 hours.

8. REFERENCES


This research was supported by Australian Research Council’s Discovery Projects funding scheme (Project Number DP0986091).