Software Optimization for MPSoC : A MPEG-2 Decoder Case Study

Eric Cheung, Harry Hsieh
Dept. of Computer Science and Engineering
University of California Riverside
Riverside, CA 92521
{chuncheung,harry}@cs.ucr.edu

Felice Balarin
Cadence Design Systems
San Jose, California 95134
felice@cadence.com

ABSTRACT

Using traditional software profiling to optimize embedded software in an MPSoC design is not reliable. With multiple processors running concurrently and programs interacting, traditional profiling on individual processors cannot capture useful execution information to assist software optimization. A new method to model parallel executions of interacting programs is needed. In this paper, we consider the software optimization problem for throughput-constrained MPSoC designs. We define the "longest delay path" as a sequence of steps leading to a throughput constraint violation and propose an algorithm to build up the path dynamically during simulation. Using an industrial-strength MPEG-2 decoder design in our case study and custom instructions for software optimization, we show that we can optimize the software efficiently in MPSoC designs using frequently executed statement information from the longest delay path.

Categories and Subject Descriptors

D.2.2 [**Design Tools and Techniques**]: Computer-aided software engineering; C.4 [**Performance of Systems**]: Design studies

General Terms

Algorithms, Design, Performance

1. INTRODUCTION

Multiprocessor System-on-a-Chip (MPSoC) has emerged as the most promising architecture for future embedded system designs. MPSoC provides high performance and low power for data-intensive multimedia applications that are not possible in single-processor architecture. An MPSoC design contains multiple interacting embedded programs. Software optimization is difficult because the programs run in parallel instead of in series. Researches on MPSoC design optimization focus on custom hardware [4], interconnects [9], interfaces [16], etc. Software optimization for MPSoC has not been well-studied.

Software optimization techniques based on traditional software profiling [3, 6] are not reliable for MPSoC designs. Traditional software profiling assumes the programs in a design run sequentially, hence reducing the execution time on any part of the programs reduces the overall execution time of the design. Traditional software profiling weights each statement execution equally and tries to find the statements that execute most frequently. However, this assumption does not apply to MPSoC designs because programs run in parallel. Some statement executions are more important than the others for the overall execution time. Therefore, traditional software profiling results on individual processors do not reveal the statements that are critical for the overall execution time of MPSoC designs. A new method to accurately determine the important statements is needed.

In this paper we try to find the statements that are critical for the overall execution time in a throughput-constrained MPSoC design. The design is composed of interacting programs running on separate processors. Programs block and unblock each others during execution, which allow necessary communication and synchronization between the programs. We use a symbolic model to analyze an execution of the design and define the *longest delay path* as the execution path among the programs that leads to a throughput constraint violation. We show that the longest delay path contains very different information from the traditional software profiling. We also propose an iterative algorithm to build up the longest delay path dynamically during simulation with reasonable simulation time overhead. Using custom instructions to speed up the most frequently executed statements in the path, we demonstrate the efficiency of software optimization in an industrial-strength MPEG-2 decoder.

The rest of the paper is organized as follows. In Section 2, we present some related works. In Section 3, we specify the throughput constraints in an MPSoC design. In Section 4, we describe our assumptions and the problem statement. In Section 5, we define the software execution model for the longest delay path. In Section 6, we propose an iterative algorithm to find the path during simulation. In Section 7, we show how the model applies to Kahn Process Network as well as other models of computation. In Section 8, we present a case study on an MPEG-2 decoder design. We conclude the paper in Section 9.

2. RELATED WORK

Traditional software profiling information on individual processors has been extensively used in optimizing software. During simulation, execution frequencies of statements are recorded [3, 6], and the most frequently executed statements (*hotspots*) in the programs are determined for optimization. However, such profiling information is only applicable for single-processor designs where all statements run sequentially. Traditional software profiling does not provide accurate hotspot information for MPSoC.

CODES+ISSS'08, October 19–24, 2008, Atlanta, Georgia, USA.

Traces are normally used in post-simulation analysis for MPSoC designs [12]. Visualization and debugging tools such as Vampir [5] and Paje [8] provide interfaces to visualize the executions of the programs in MPSoC simulation. These tools focus on efficient generation, synchronization and interpretation of multiple traces from multiple processors. They do not create hotspot information for the programs and no automatic analysis has been proposed for software optimization.

Longest path finding in parallel computation is common for hardware designs. Critical paths in logic [1] and gate level [7] have been extensively used to estimate the shortest clock cycle possible in a synchronous digital circuit. Throughput analysis has been applied to Synchronous Data-Flow graphs [2] without the need of simulation. These works rely on well-define semantics of the parallel executing elements that do not exist in many richer models of computation such as Kahn Process Network. We consider MPSoC designs which consist of multiple programs that are large and require long simulation to activate meaningful execution paths.

Custom instructions [15], hardware accelerators [11, 14] and library routines [13] are common techniques to speed up software executions. They provide speedups by replacing the *hotspots* with faster executions in specific software or hardware. Since *hotspot* information for MPSoC is necessary to apply these techniques, correctly identifying *hotspot* information in the programs is crucial to optimize software in limited design time.

3. THROUGHPUT CONSTRAINT

In this paper, we focus on throughput constraints in MPSoC designs. MPSoC is a common implementation platform for multimedia and signal processing applications. Simulation is typically used to examine whether a design satisfies the throughput constraints. The input of the simulation can be an input benchmark for the application, or an input that may violate the constraints constructed by a property checking tool based on the abstract model of the design.

We consider the *system* as the implementation of the design and the *environments* as the external controls. In an MPSoC design, the system consists of multiple processors and interconnects. The environments are the input/output components of the system, such as sensors, networks and monitors. The environments are independent of the execution of the system, and expect inputs and outputs to follow pre-defined performance constraints.

3.1 Buffered Input/Output

To maintain steady flows between the system and the environments, inputs and outputs are buffered as shown in Figure 1. The environments put data to the input buffers and consume data from the output buffers in pre-defined rates. Buffered I/O provides more implementation freedom to the system by allowing the system and the environments to read and write asynchronously.

The time which the environments read and write from the buffers is independent of the system. The system does not have any control over the time which the environments are expected to put input data or consume output data. If the system can affect the reads and writes of the environments, the impact should be modeled inside the system such that the system and the environment are independent. Such independency is very important to define consistent throughput constraints.

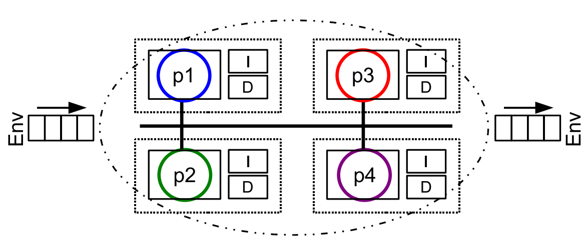

Figure 1: Buffered Input/Output

3.2 Throughput Constraint Violation

A throughput constraint is violated when an input buffer overflows or an output buffer underflows. The input buffers and output buffers are sized in the system such that they withstand the asynchronous reads and writes from the system and the environments. However, if the system is too slow to consume data from the input buffers or produce data to the output buffers in the pre-defined rates, the input buffers will eventually be full of data when the environments write (overflow), or the output buffers will be empty when the environments read (underflow). A throughput constraint is violated when an overflow or an underflow happens. Therefore, the simulation checks the input buffers for overflow and output buffers for underflow to determine whether the MPSoC design satisfies the throughput constraints.

4. PROBLEM STATEMENT

Problem Statement. *Assist the designers in software optimization by automatically narrowing the correct hotspots to a small number of statements.*

If MPSoC simulation shows that a throughput constraint is violated, we try to help the designers to resolve the constraint violation with software optimization by automatically generating correct hotspot information. Similar to traditional software profiling information for single-processor designs, such hotspot information for MPSoC allows designers to focus on small number of statements to optimize their software.

4.1 Assumption

We assume each program in the design runs on one processor in MPSoC. With multiple processors in the system, each processor has a well-defined role and is dedicated to one job. Each program runs on one processor only. We are currently investigating the effect when multiple programs run on a processor.

We further assume the *program steps* in the MPSoC design have strictly precedence relationships. The programs run cooperatively instead of independently. If we consider the execution of a program as a sequence of steps, certain steps have dependencies to other steps in other programs. These steps have to wait for their dependent steps in other programs to finish before they can execute. Such assumption is reasonable for MPSoC designs and applies to many models of computation. We will show the symbolic model for the relationships in Section 5 and how the relationships are applied to Kahn Process Network designs in Section 7.

4.2 Longest Delay Path

To reveal the correct hotspot information in the programs for software optimization, we first need to determine the execution path that leads to the constraint violation. We consider a violation step as the step where the starting time comes late and does not meet the throughput constraint. For an input buffer, the violation step is the step that reads from the input buffer after the buffer overflows. Similarly, for an output buffer, the violation step is the step that writes to the output buffer after the buffer underflows. The violation step is said to be responsible for the constraint violation because the constraint would not have been violated had the step started earlier.

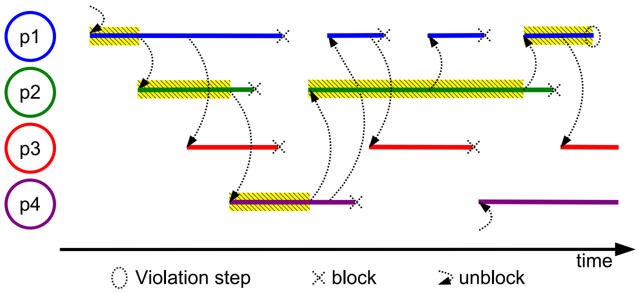

Figure 2: Longest Delay Path

We define the *longest delay path* with respect to a processing step in a program as the sequence of executed statements among all programs that contributes to the earliest starting time of the step. As each program runs sequentially and some steps wait until their dependent steps in other programs finish under the strictly precedence relationships, there is an execution path among the programs that leads to the earliest starting time of the step. Reducing the execution time on any part of the path allows an earlier starting time.

An example of the longest delay path with respect to a violation step is shown in Figure 2. *p1* to *p4* are four programs running on an MPSoC system. In the figure, *block* means a processing step in a program cannot execute right away because of the strictly precedence relationships. When the dependent steps are later finished, the step is *unblocked*. The figure shows the blocking and unblocking between the programs over the execution time. In the example, the input buffer overflows because an input read from *p1* comes too late. The longest delay path of the violation step, as highlighted in the figure, is the execution path among the programs that leads to the earliest starting time of the violation step. According to the path, program *p2* is responsible for majority of the delay that leads to the constraint violation. Therefore, software optimization should focus on the statements on *p2*. On the other hand, optimizing *p3* cannot resolve the constraint violation. Definition of the longest delay path will be shown in Section 5.3 and algorithms to find such path will be shown in Section 6.

5. MPSOC EXECUTION MODEL

In this section, we describe the symbolic model to analyze *an execution* of an MPSoC design. We model each program in an MPSoC system as a step transition, and we use strictly precedence relationships to model interactions between the programs.

5.1 Execution Model

For each program in the MPSoC design, we model it as a step transition. Each non-repeating step σ represents a processing step in a program. $\sigma_i^P \in S$ is the step i in program P and σ_0^P is the beginning of the program. S is the set of all steps in all programs. $\tau_i^P \in R^+$ is the starting time of the step σ_i^P, and $\delta_i^P \in R^+$ is the execution time (delay) of the step σ_i^P.

A step represents execution of a set of statements in the program and its execution history (hence non-repeating). A statement that can be blocked by other programs always start a step, and a statement that can unblock other programs always end a step. Therefore, blocking and unblocking always occur between steps.

Property 1. *Step sequence in a program:*

For all steps $\sigma_i^P, i \in [0,\infty)$ in a program P,

the step sequence is $\sigma_0^P \sigma_1^P \sigma_2^P \ldots$

Property 2. *Starting time restriction on consecutive step:*

For all steps $\sigma_i^P, i \in [1,\infty)$ in program P,

$$\tau_{i-1}^P + \delta_{i-1}^P \leq \tau_i^P$$

Property 1 specifies that each program executes sequentially. Each step σ_i^P is an execution of a set of statements and δ_i^P is the execution time of the statements. A program runs sequentially and subsequent step cannot start before the previous step finishes. Therefore a program is executed following the step sequence and the earliest starting time of each step is restricted by Property 2.

We also define environment events σ_i^E as a sequence of steps from the environments. σ_0^E is an event representing the beginning of the execution. $\sigma_i^E, i > 0$ are steps that read and write from the buffered inputs and outputs. Since the environments are independent of the system, all starting time and execution time of the environment events (τ_i^E and δ_i^E) are pre-defined based on the throughput constraints.

Property 3. *Starting time restriction on precedence relation:*

Step σ_j^Q strictly precedes step σ_i^P

$$\rightarrow \quad \tau_j^Q + \delta_j^Q \leq \tau_i^P$$

Interacting programs have strictly precedence relationships. If a dependency is implied such that the step i in program P cannot start before the step j in program Q finishes, we can specify the dependency as a starting time restriction. Property 3 shows the starting time restriction of a strictly precedence relation. The relation limits the earliest starting time of the steps in addition to the restriction shown in Property 2.

5.2 Earliest Starting Time

With each program runs on a processor, program steps start *as soon as possible* after all restrictions in Property 2 and 3 are satisfied. For a step σ_i^P that depends on steps σ_j^Q and σ_k^R, its starting time is the latest of the finish time for its previous step σ_{i-1}^P in the same program and all the dependent steps.

$$\tau_i^P = max(\tau_{i-1}^P + \delta_{i-1}^P, \tau_j^Q + \delta_j^Q, \tau_k^R + \delta_k^R)$$

Step σ_i^P is called *blocked* when it cannot start immediately after its previous step σ_{i-1}^P. i.e. $\tau_{i-1}^P + \delta_{i-1}^P \neq \tau_i^P$. The step is *unblocked* by one of the dependent steps σ_j^Q when $\tau_j^Q + \delta_j^Q = \tau_i^P$.

We define $Pre : S \rightarrow S$ as the *immediate prior step* relationships. If a step is not blocked, $Pre(\sigma_i^P) = \sigma_{i-1}^P$ because the step immediately follows its previous step in the same program. If it is blocked, $Pre(\sigma_i^P) = \sigma_j^Q$ where σ_j^Q is the step that unblocks σ_i^P. We assume no one step is unblocked by two steps at the same time. Therefore, the immediate prior step of the step σ_i^P is defined as follows:

$$\forall \sigma_i^P, i \in (0,\infty), Pre(\sigma_i^P) = \{\sigma_{i'}^{P'} | \sigma_{i'}^{P'} \text{ unblocks } \sigma_i^P\}$$

Immediate prior step of a step σ_i^P can come from the environments if σ_i^P is blocked by reading from an input buffer when the buffer is empty or by writing to an output buffer when the buffer is full. In such case, $Pre(\sigma_i^P) = \{\sigma_j^E\}$ where σ_j^E is the environment event. The immediate prior step of any environment event σ^E is $\emptyset$.

5.3 Definition of Longest Delay Path

The longest delay path of a processing step can be defined using the symbolic model. The longest delay path is the sequence of statements that contributes to the earliest starting time of the step. The path can also be represented by a set of steps in S. Since the earliest starting time of step σ_i^P depends on the step $Pre(\sigma_i^P)$, we can define the longest delay path of the step σ_i^P as follows.

$$LDP(\sigma_i^P) = \{\sigma_{i'}^{P'} | (\sigma_{i'}^{P'} = \sigma_i^P) \\ \vee (\sigma_{i'}^{P'} = Pre(\sigma \in LDP(\sigma_i^P)))\}$$

The longest delay path of the violation step for a constraint violation is used to determine hotspot information for software optimization. The violation step σ_{iv}^{Pv} is either a read from an input buffer or a write to an output buffer. $LDP(\sigma_{iv}^{Pv})$ represents the execution path among the programs that leads to the constraint violation. Such path provides designers important information on where to optimize their software in order to resolve the violation.

6. ALGORITHM

6.1 Naïve Algorithm

Since the longest delay path has a recursive definition, a naïve algorithm uses traces from simulation and back-tracks the execution path from the step of the constraint violation. Such algorithm can be used alongside with trace-based analysis tools [5, 8]. Starting from the violation step σ_{iv}^{Pv}, we use the simulation traces to back-track the immediate prior step $Pre(\sigma_{iv}^{Pv})$. The immediate prior step is responsible for the lateness of the starting time of the step σ_{iv}^{Pv}. We can build up the longest delay path of the violation step by recursively back-tracking until the immediate prior step is an environment event (i.e. a write in the input buffer or a read from the output buffer). The naïve algorithm is shown in Algorithm 1.

Algorithm 1: Naïve Trace-based Algorithm

```
Input: trace, σ_iv^Pv
Output: LDP(σ_iv^Pv)
LDP(σ_iv^Pv) = {σ_iv^Pv}
σ_i^P = σ_iv^Pv
while Pre(σ_i^P) ≠ σ^E do
    LDP(σ_iv^Pv) = LDP(σ_iv^Pv) ∪ {Pre(σ_i^P)}
    σ_i^P = Pre(σ_i^P)
end
return LDP(σ_iv^Pv)
```

Although the naïve trace-based algorithm allows the longest delay path to be derived from simulation traces, the algorithm relies on complete simulation traces and post-simulation analysis. Generating traces is very expensive in term of simulation speed and disk space. The algorithm is not scalable for long simulation and may generate very large traces that are difficult to analyze. Therefore, we present an iterative algorithm to derive the longest delay path dynamically during simulation that does not require generating any traces.

6.2 Iterative Algorithm

The iterative algorithm comes from the definition that the longest delay path of any step σ_i^P is based on the longest delay path of its immediate prior step $Pre(\sigma_i^P)$. When expanding the definition of the $LDP(\sigma_i^P)$, the path includes the step σ_i^P itself and the longest delay path of its immediate prior step $LDP(Pre(\sigma_i^P))$. Therefore, we can use the following definition to iteratively build up the longest delay path in each step during simulation.

$$LDP(\sigma_i^P) = \{\sigma_i^P\} \cup LDP(Pre(\sigma_i^P))$$

The algorithm used to keep track of the longest delay path in each step dynamically during simulation is shown in Algorithm 2. When each step starts, it copies the longest delay path from its immediate prior step and adds itself to the path. The immediate prior step can be an environment event σ^E if the step is blocked by a read or a write from the environments.

Algorithm 2: Iterative Algorithm

```
Output: LDP(σ_iv^Pv)
forall program P do
    LDP(σ_0^P) = {σ_0^E}
end
repeat
    foreach step σ_i^P starts do
        LDP(σ_i^P) = {σ_i^P} ∪ LDP(Pre(σ_i^P))
    end
until σ_i^P violates a throughput constraint
return LDP(σ_i^P)
```

Only the longest delay path of the currently executing step in each program is needed to be kept to determine the longest delay path of the step for a constraint violation. We do not keep the paths for other steps since we are only interested in the violation step. As the violation step executes after a constraint is violated, we know immediately when the step comes late. Therefore, we only need to keep one longest delay path of the currently executing step in each program and discard those steps that are finished.

6.3 Optimization Methodology

The longest delay path analysis should be repeated after each software optimization is applied. After an optimization is applied, the original longest delay path is shortened and no longer violates the constraints. However, there may exist another path that still violates the constraints and becomes the new longest delay path. Therefore, the analysis should be repeated.

The software optimization methodology is shown in Figure 3. A designer-in-the-loop approach is used to repeat the longest delay path analysis after each optimization is applied to ensure that the subsequent optimization is based on correct hotspot information according to the updated longest delay path. The optimization steps repeat until all throughput constraints are satisfied across all interested inputs.

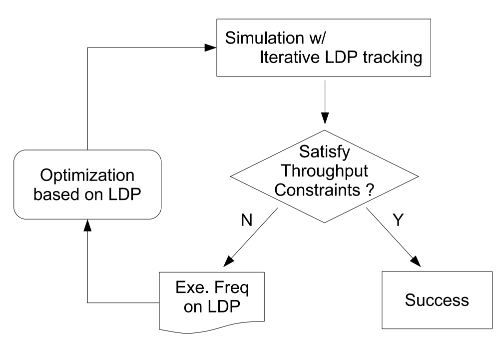

Figure 3: Software Optimization Methodology

7. MODEL OF COMPUTATION

Kahn Process Network (KPN) specification is a common model of computation for MPSoC. A KPN application is modeled as a set of processes that communicate using a set of FIFO. Reads and writes to the FIFO are blocking. A program is blocked when it reads from an empty FIFO or writes to a full FIFO. The program is then unblocked when another process writes to the empty FIFO or read from the full FIFO.

The computation model of KPN complies with our software model with strictly precedence relationships. Each process in KPN is a sequential program that only communicates to other processes using FIFO. A FIFO f is an ordered queue where data is produced and consumed in the same order. A FIFO contains a sequence of data. The i-th data in the FIFO f is denoted as f^i. The $Prod(f,i)$ and $Cons(f,i)$ denote production (writing) and consumption (reading) of the data f^i.

The semantics of a FIFO restrict the starting time of production and consumption of the data. Specifically, the production of the data f^{i+1} must happen after the production of the data f^i (i.e. $\tau_{Prod(f,i)} \leq \tau_{Prod(f,i+1)}$), and the consumption of the data has to be in the same order (i.e. $\tau_{Cons(f,i)} \leq \tau_{Cons(f,i+1)}$). In addition, the data must be produced before it can be consumed, i.e. $\tau_{Prod(f,i)} \leq \tau_{Cons(f,i)}$. If a process tries to consume the data from the FIFO before the data is produced, the process has to wait until the data is available. This is commonly referred to as "blocking reads". For a FIFO f with size N, the production of the data f^{i+N} cannot occur before the consumption of the data f^i. As a result $\tau_{Cons(f,i)} \leq \tau_{Prod(f,i+N)}$. This is commonly referred to as "blocking writes". These restrictions can be represented in our software model as strictly precedence relationships.

	blocking	unblocking
shared variable	spinwait	write
message-passing	blocking read/write	write/read
handshaking	synchronize	synchronize

Table 1: Blocking Mechanism

Blocking mechanism is also very common in other multiprocessor models of computation. As shown in Table 1, blocking and unblocking are commonly used in MPSoC to synchronize multiple asynchronously executing programs. In symmetric multiprocessing, blocking can be achieve by a spinwait on a shared variable, and the spinwait is unblocked with a proper write to the variable. Although such blocking and unblocking mechanisms are not as explicit as in KPN, they can still be modeled for the longest delay path analysis.

8. MPEG-2 DECODER CASE STUDY

An MPEG-2 decoder design is used to demonstrate the effectiveness of software optimization using the hotspot information from the longest delay path. The MPEG-2 decoder is manually designed such that high-level parallelism of the application is explicitly defined. The decoder is developed in KPN with complex controls and operations. The MPEG-2 decoder design consists of 9 processes and 63 FIFO. The process network is shown in Figure 4. The controller process controls the dataflow of the MPEG-2 stream through the decoding stages. The stream is first parsed with variable length decoding into frames and macro-blocks. Each of them is then decoded through inverse scan, inverse quantization and inverse discrete cosine transform. Prediction processes predict the frames and motion compensation. Output processes combine the results and produce a raw video stream.

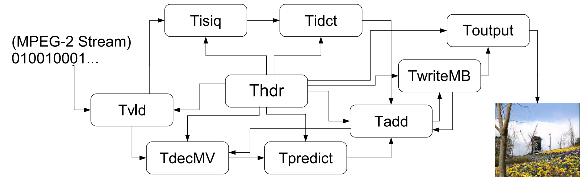

Figure 4: MPEG-2 Decoder Design

In the experiments, we use Tensilica's Xtensa LX2 processors with a typical configuration. The processors are extensible such that we can design custom instructions and integrate them into the processor datapaths using *Tensilica Instruction Extension language* (TIE) [15]. Processors in the MPSoC design become heterogeneous by adding custom instructions into the processors.

8.1 Statement Execution Frequency

In the first experiment, we compare the hotspot results between the traditional software profiling and the longest delay path. To introduce a constraint violation, we set a very tight constraint to decode one group of pictures of an MPEG-2 stream. Each group of pictures contains half a second of video.

program	line #	profiling (%)	LDP (%)	diff (%)
Twritemb	299-300	9.96%	0.02%	-99.80%
Tpredict	400-401	7.56%	14.66%	+93.92%
Toutput	401-402	6.51%	0.71%	-89.09%
Tidct	203-237	5.79%	10.51%	+81.52%
Tpredict	382-390	5.48%	14.31%	+161.13%
Tadd	266-268	5.30%	0.59%	-88.87%
Tadd	278-285	4.59%	0.20%	-95.64%
Tpredict	367-369	2.72%	5.07%	+86.40%
Tpredict	338-339	2.17%	3.93%	+81.11%
Tpredict	351-357	2.01%	5.28%	+162.69%
Tidct	147-181	1.90%	3.79%	+99.47%
Tpredict	296-299	1.87%	4.64%	+148.13%

Table 2: Profiling vs. Longest Delay Path

Table 2 shows the execution frequencies of the statements in the traditional software profiling and the longest delay path. We compare the frequencies of the 14 most frequently executed statements in the traditional software profiling. These 14 statements are responsible for more than 50% of the total execution time in both results. As shown in the table, although some statements take a long time to execute (i.e. line 299-300 in *writemb*), the statements do not contribute to the longest delay path. Optimizing these statements does not provide any throughput improvements in the MPSoC design. On the other hand, some statements (i.e. line 382-390 in *predict*) show more significance in the longest delay path. These statements are important in the MPSoC design and optimizing these statements will provide substantial throughput improvements.

The overhead to keep track of the longest delay path in each step dynamically during simulation increases the simulation time by about 70%. Such overhead is comparable to traditional software profiling in single-processor simulation and several times faster than generating simulation traces for off-line analysis.

8.2 Software Optimization Result

Based on the hotspot information from both results, we apply software optimization to the processes in the MPEG-2 decoder. We use Tensilica's *XPRES* compiler [10] to generate custom instructions for the most frequently executed statements. We direct the XPRES compiler to optimize the statements according to the hotspot information. We use the default options to combine multiple instructions in the original programs into a lesser number of complex instructions. Table 3 shows the speedups of the custom instructions on the statements we used in the experiments. The generated custom instructions reduce the execution time of the statements themselves by 36% to 55%. The table also shows the numbers of gates required to implement the custom instructions.

#	program	line #	gate	runtime
1	Twritemb	299-300	12,683	-42%
2	Tpredict	400-401	7,421	-38%
3	Toutput	401-402	12,693	-37%
4	Tidct	203-237	27,295	-36%
5	Tpredict	382-390	8,727	-50%
6	Tadd	266-268	12,779	-42%
7	Tadd	278-285	5,356	-38%
8	Tpredict	351-357	8,247	-47%
9	Tpredict	367-369	5,218	-55%
10	Tpredict	296-299	4,747	-40%
11	Tpredict	338-339	4,927	-55%
12	Tidct	147-181	17,861	-47%

Table 3: Speedup for Custom Instruction

We compare the throughput improvements using the hotspot information between the traditional software profiling and the longest delay path. For the traditional software profiling, we apply the custom instructions in the order of execution frequencies shown in the column *profiling* in Table 2. For the longest delay path, we iteratively apply custom instructions to the most frequently executed statements shown in the longest delay path and follow the described optimization methodology in Section 6.3. We limit the area for custom instructions to 90K gates.

The software optimization results are shown in Figure 5. Using the longest delay path, we can correctly determine the important statements that can speed up the MPSoC design. Therefore, a throughput improvement can be observed in every custom instruction we applied. On the other hand, traditional software profiling does not reveal the statements that are important in the MPSoC design. With imprecise hotspot information, designers will waste their time optimizing an unimportant part of the programs and can only discover later that the optimization does not show any throughput improvements in the simulation. As a result, the software optimization using the longest delay path offers 50% better throughput improvement than using the traditional software profiling with 90K gates. In the scenario where 10% throughput improvement is required to meet the throughput constraints, custom instructions using information from the longest delay path take 16K gates, while custom instructions using information from traditional profiling take 69K gates. Our longest delay path analysis provides designers correct hotspot information and allows designers to optimize the software efficiently.

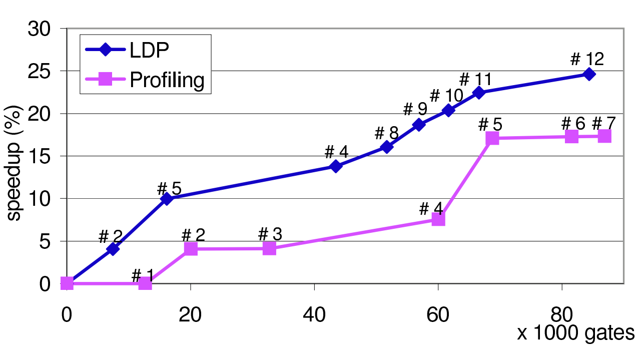

Figure 5: Software Optimization Result

9. CONCLUSION

In this paper, we present a software optimization case study on an MPSoC design. We define the longest delay path as the execution path that is important to optimize in order to resolve a throughput constraint violation. We present an iterative algorithm to derive the path dynamically during simulation with reasonable simulation time overhead. We show that the longest delay path correctly identifies the hotspots for efficient software optimization in MPSoC.

10. REFERENCES

[1] V. D. Agrawal. Synchronous path analysis in mos circuit simulator. In *DAC '82: Proceedings of the 19th conference on Design automation*, pages 629–635, Piscataway, NJ, USA, 1982. IEEE Press.

[2] F. Balarin, L. Lavagno, et al. Scheduling for embedded real-time systems. *IEEE Des. Test*, 15(1):71–82, 1998.

[3] T. Ball and J. R. Larus. Optimally profiling and tracing programs. *ACM Trans. Program. Lang. Syst.*, 16(4):1319–1360, 1994.

[4] R. Banakar, S. Steinke, et al. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In *CODES '02: Proceedings of the tenth international symposium on Hardware/software codesign*, pages 73–78, New York, NY, USA, 2002. ACM.

[5] H. Brunst, D. Kranzlmüller, and W. Nagel. Tools for scalable parallel program analysis - vampir ng and dewiz. pages 93–102. 2005.

[6] P. P. Chang, S. A. Mahlke, and W. mei W. Hwu. Using profile information to assist classic code optimizations. *Softw. Pract. Exper.*, 21(12):1301–1321, 1991.

[7] H.-C. Chen, D. H. C. Du, et al. Critical path selection for performance optimization. In *DAC '91: Proceedings of the 28th conference on ACM/IEEE design automation*, pages 547–550, New York, NY, USA, 1991. ACM.

[8] J. C. de Kergommeaux and B. de Oliveira Stein. Paje: An extensible environment for visualizing multi-threaded programs executions. In *Euro-Par '00: Proceedings from the 6th International Euro-Par Conference on Parallel Processing*, pages 133–140, London, UK, 2000. Springer-Verlag.

[9] G. de Micheli and L. Benini. Networks on chip: A new paradigm for systems on chip design. In *DATE '02: Proceedings of the conference on Design, automation and test in Europe*, page 418, Washington, DC, USA, 2002. IEEE Computer Society.

[10] D. Goodwin and D. Petkov. Automatic generation of application specific processors. In *CASES '03: Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*, pages 137–147, New York, NY, USA, 2003. ACM.

[11] R. K. Gupta and G. D. Micheli. Hardware-software cosynthesis for digital systems. pages 5–17, 2002.

[12] G. Martin. Overview of the mpsoc design challenge. In *DAC '06*, pages 274–279, New York, NY, USA, 2006. ACM Press.

[13] A. Peymandoust, G. D. Micheli, and T. Simunic. Complex library mapping for embedded software using symbolic algebra. In *DAC '02: Proceedings of the 39th conference on Design automation*, pages 325–330, New York, NY, USA, 2002. ACM.

[14] G. Stitt, F. Vahid, and S. Nematbakhsh. Energy savings and speedups from partitioning critical software loops to hardware in embedded systems. *Trans. on Embedded Computing Sys.*, 3(1):218–232, 2004.

[15] A. Wang, E. Killian, et al. Hardware/software instruction set configurability for system-on-chip processors. In *DAC '01: Proceedings of the 38th conference on Design automation*, pages 184–188, New York, NY, USA, 2001. ACM.

[16] M.-W. Youssef, S. Yoo, et al. Debugging hw/sw interface for mpsoc: video encoder system design case study. In *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 908–913, New York, NY, USA, 2004. ACM.