Multi-Threaded Processor Design for Embedded Systems

Ran Zhang[1] Hui Guo[2]

[1] University of New South Wales, Australia
cliffran87@hotmail.com
huig@cse.unsw.edu.au

Technical Report
UNSW-CSE-TR-201310
April 2013

THE UNIVERSITY OF
NEW SOUTH WALES

School of Computer Science and Engineering
The University of New South Wales
Sydney 2052, Australia

Abstract

Multi-threaded processor design enables high performance of a single processor core by exploiting both the thread-level and instruction-level parallelism. This performance gain is, however, at the cost of increasing energy consumption, which is not desirable to embedded systems. This paper investigates multi-threaded designs of varied thread number and under two different thread switching schemes: fine-grained and coarse-grained. Based on our experiment with a 6-stage PISA processor, we found that in terms of energy efficiency the coarse-grained based designs are better than the fine-grained designs. And for the coarse-grained design, the thread number for an optimal design is closely related to the memory access delay; when the memory access latency is small, the low-thread processor appears more energy efficient than the processor with a high thread number, but when the memory delay increases the high-thread processor becomes superior.

1 Introduction

Performance and energy consumption are critical design issues in embedded systems. To improve performance, parallel processing techniques are often used. Parallel processing can be applied at different design levels with different parallelism being exploited: at the instruction level, independent instructions can be executed in parallel; At the data level, an operation can be performed simultaneously on an array of data elements; and at the thread level, multiple instruction streams can be concurrently executed.

To support the parallel processing, various of processor architectures have been proposed. The pipelined processor is one of the basic designs to exploit instruction-level parallelism. Multi-threaded processor designs [1] can even improve the pipeline performance by exploiting thread level parallelism. When one thread stalls the processor execution due to dependency, the processor can switch to other thread so that the pipeline stall is avoided.

Compared to other typical processor architectures with parallel processing capabilities, such as superscalar and vector processor, the multi-threaded processor design possess some advantages. Its design is much simpler than the superscalar and its application area is more broader than the vector processor.

Since the performance improvement from parallel processing will inevitably be at the cost of energy. How well does each unit of extra energy spent on the performance improvement in the multi-threaded processors? This is the question we want to answer in this paper.

Our main contributions are as following:

- we proposed an evaluation scheme for energy efficient multi-threaded designs, where energy consumption per instruction is used for easy comparison among different designs;
- We developed profiling based estimation models for the CPI and power consumption of a multi-threaded processor; Our experiments show that our models provide values consistent with the results given by the commercial design tools. The estimation models effectively enable exploration of different multi-threaded designs
- We demonstrated that given a baseline processor, there is an optimal design with high energy efficiency and it can be identified with the estimation models.

The rest of paper is organized as follows. In Section 2, we review the related work on the multi-threaded processor design. Our energy estimation models for the processor evaluation are discussed in Section 4. The experimental verification of the models is given in Section 5, followed by the evaluation of different processor designs. The paper is concluded in Section 6.

2 Related Work

The application of multi-threaded processor can be traced back to a MIMD (Multi-Instruction Multi-Data) architecture, HEP (Heterogeneous Element Processor), proposed in [2]. HEP consists of several Process Execution Modules

(PEMs) connected to a single data memory. Each PEM is a fine-grained multi-threaded processor. The HEP system can execute up to 16 tasks simultaneously. A task may have up to 64 processes (threads).

In [3], authors proposed a multi-threaded processor architecture for parallel symbolic computing (MASA), where a large set of procedures/routines are frequently performed for basic symbol operations, such as expression simplification, differentiation and polynomial factorization. Those basic operations (tasks) are arranged in multiple context frames (threads). A fine-grained multi-threaded scheme is applied to switch the execution between frames in MASA.

Both HEP and MASA implement the fine-grained threading on the scalar processor. The fine-grained multi-threaded has also been applied to VLIW (Very Long Instruction Word) architectures. The Horizon machine [4] and Tera computer [5] are two typical examples. With their designs, four levels of parallelism are exploited for high performance: system level parallelism, with multiple processors; thread-level parallelism, with multi-thread interleaved execution within individual processors; instruction level parallelism, with the overlapped instruction execution in each pipeline; and operation level parallelism with the multi-operations in each horizontal (wide) instruction.

For multi-thread interleaved execution, separate registers are needed for different threads, which lead to a huge register file. If implemented with latches or flip-flops, as a common case in the traditional design, the big register file would consume a large chip area. To reduce the hardware cost, in [4], the register file is implemented with the cheap but slow memory.

For the fine-grained thread interleaving, sufficient number of threads are needed to retain a high throughput, which may not be always possible. To handle this situation, in Tera Computer [5], an Explicit-Dependence lookahead approach is used. Several bits are added to each instruction to specify the number of successive independent instructions that can be executed consecutively in the pipeline.

In [6], authors proposed a multi-computer model, called M-Machine, in order to achieve high performance for both individual threads and the overall system. The M-Machine consists of a set of computing nodes interconnected by a bidirectional 3-D mesh network. A node consists of a multi-ALU (called MAP) chip. Each MAP chip contains four clusters and one cluster is a three-issue, pipelined processor consisting of four ALUs. Two mechanisms are used for intra-node concurrency: Vertical Thread (V-thread) and Horizontal Thread (H-thread). A V-thread consists of up to four H-threads. A H-thread is a 3-instruction wide stream which is statically scheduled and executed on a single MAP cluster. The H-threads within the same V-thread can communicate via registers, while the H-threads of different V-threads must communicate through memory. The four H-threads within a V-thread can be either independently or collectively scheduled by the compiler to achieve an instruction level parallelism of 12-instructions.

The designs discussed above exploit the thread parallelism mainly at the fine-grained level, where the execution switches to other thread on each instruction.

In contrast, the coarse-grained multi-threaded processor consecutively executes a string of instructions for a thread before switches to other thread.

In the designs proposed in [7]-[9], the load and branch instructions are chosen as the static thread switching points since those instructions often cause the pipeline stall.

A design technique for fast thread switching is also discussed in [7]. In a general five-stage pipeline, a thread switching can often be performed in the second stage after the instruction is decoded. Therefore, two cycles will be wasted for a thread switching. To reduce such an overhead, the authors [7] used a dedicate bit in the instruction to explicitly code the thread switching operation. Switching can start before the instruction decoding, hence one clock cycle can be saved. To further improve the performance, they introduced a thread switch buffer to hold the addresses of most-recently-used load instructions. If the address of the next instruction matches with an address in the thread switch buffer, a thread switching is performed immediately. In this case, the switching overhead is reduced to zero.

In [8], the authors implemented a coarse-grained multi-threaded system (MSparc) on a Sparcle processor architecture. MSparc is a large-scale multiprocessor system with the distributed memory. The multi-thread implementation helps reduce the impact of the long remote memory delay on the overall system performance and at the same time achieve high speed of the single thread execution. In their design, a cache-miss monitor unit is used to dynamically detect the thread switching event.

A similar design approach can be found in [10], where other conditions, such as synchronization failure, are also taken into consideration in the thread switching control.

The dynamic thread switching requires a complex switching condition detect unit, which may increase the clock cycle time. In addition, since the event that triggers a thread switching can often be detected at a later stage (after some calculations), any instructions in the pipeline after this stage should be discarded when the thread switching occurs, which leads to a high performance penalty per thread switching. It is reported in [10] that a thread switching can take about 11 cycles.

In [9], the authors proposed a design that combines the synergy of both fine-grained (low switching overhead incurred) and coarse-grained threading (small number of threads required) approaches. With their design, the processor works as a fine-grained multi-threaded processor until a long-latency instruction is encountered, to achieve high performance with a small number of available threads.

The NIGARA [11] multi-threaded processor, is another example of exploiting both the fine-grained and coarse-grained multi-thread execution. In this design, the processor pipeline is pre-fixed with two extra stages: Fetch and Thread Select. In the Fetch stage, instructions from all threads are fetched. Then the *Thread Select* stage selects the instruction to be executed in the pipeline. The extra two stages pre-determine the target for the thread switching; hence it takes zero clock cycle for the pipeline execution switches from one thread to another thread. In addition, a dynamic thread selection policy is used so that the processor execution can be thread-interleaved at the fine-grained level, mixed fine-grained/coarse-grained level, or coarse-grained level, according to the number of available threads.

The above multi-threaded designs are mainly for high performance. Different issues were also addressed in some other works. In [12], the authors investigated the power, performance and energy of multi-threaded processors.

A separate comparison between the multi-threaded design, single thread execution, and superscalar out-of-order execution is presented in paper [13]. The

paper shows that multi-threaded single pipeline in-order execution can achieve a similar speedup as the out-of-order instruction execution in the superscalar, but has a better energy efficiency. To facilitate the exploitation of the parallel processing offered by the multi-threaded processor, the authors extend the C langauge with explicit parallelism constructs. With these parallelism constructs, the compiler is able to identify parallel threads, recognize the shared variables and ensure the correct communications between threads.

In this paper, we investigate both fine-grained and coarse-grained thread interleaving designs for the single pipeline, and try to find the optimal design with high energy efficiency.

3 Overview of Target System

We target a system with multiple input execution threads; all threads are independent from each other. Figure 3.1 shows the structure of the multi-threaded system. It consists of a plain pipelined processor with no components for hazard treatment, a thread switching control unit (**SCU**) to switch instruction execution between threads, and a reconfigurable forwarding/stall unit to mitigate the impact of pipeline hazards due to the dependency between instructions on the pipeline. The reconfigurable forwarding/stall unit can be customized for different input threads and can be reduced from full forwarding connections to nil forwarding paths.

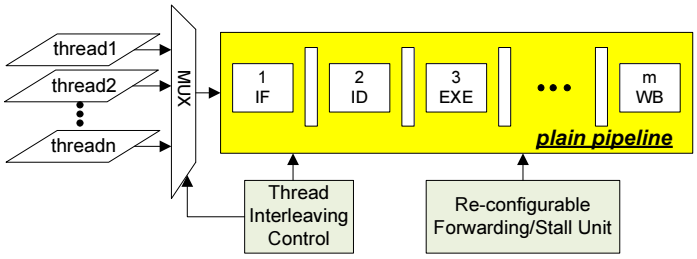


Figure 3.1: Multi-Threaded Processor System

Different number of input threads and different switching control schemes will have different multi-threaded processor designs. Here we assume the processor has a fixed m pipeline stages and takes n independent threads, and the thread switching can be either fine-grained (**FG**) or coarse grained (**CG**).

For the FG thread interleaving execution, the switching control logic is very simple. Switching is deterministic on each clock.

Full pipeline speedup can be achieved if there are sufficient independent threads. With the full speed execution, there are no pipeline hazards. Therefore, the forwarding and hazard detection unit can be totally removed from the pipeline, hence reducing the hardware costs, hence the energy consumption. The FG based design becomes less effective when the speed gap between the processer and memory increases or the pipeline become deeper with many stages.

The CG thread interleaving can reduce the impact of long memory access delay and execution flow change on performance. The CG designs interleave

execution between threads on a chunk/block basis, where instructions in a block are consecutively executed.

4 Evaluation Model

We evaluate the processor design based on its energy efficiency. The energy is the product of power consumption and execution time. It may vary significantly from application and application. It is not straightforward to see whether a processor is energy efficient by looking at the amount of energy the processor consumed for an application execution. Instead, the **energy per instruction** is more indicative. Therefore, we use the following formula for energy consumption:

$$E = P \times CPI/f \tag{4.1}$$

where CPI is the **average clock cycles per instruction**, f the clock frequency, and P the power consumption.

For a given baseline processor, we assume f is fixed. Our focus is then the power consumption and CPI, which are discussed in the next two sub-sections.

4.1 Power Consumption

As shown in Figure 3.1, our multi-threaded processor is based on a baseline pipeline processor. A processor can be abstracted as a connection of components, including the register file and forwarding unit. The power consumption of the processor is sum of the power consumed by each component. Some components are rarely scaled with the thread number, but for the register file and forwarding unit, their size and complexity can change greatly with different number of threads in the design. We separate them from other components, and the power consumption of the baseline processor (for single thread execution) can be represented as

$$P_{base} = p_0 + p_{rf} + p_{fw}, \tag{4.2}$$

where p_{rf} and p_{fw} are the power consumed by the register file and forwarding unit, respectively; and the power consumed by rest components is denoted as p_0, which is deemed as constant.

When the baseline processor is extended for the multi-threaded execution, the power consumption is increased. The power consumption for n-thread design (i.e. **n-threaded processor**) can be represented as

$$P_{nt} = p_0 + P_{RF} + P_{FW} + P_{SCU}, \tag{4.3}$$

where P_{RF} and P_{FW} are the power consumption of the altered register file and forwarding unit, and P_{SCU} the power consumed by the switching control unit.

If each thread uses the same size of register file as in the baseline processor, the total registers required for the n-threaded processor is linearly scaled with the thread number. Therefore, the related power consumption can be written as

$$P_{RF} = n \times p_{rf}, \tag{4.4}$$

where p_{rf} is the power consumption of the baseline register file.

Note that the register file for each thread can be reduced with some customization techniques, which have been investigated in other research work [].

Here we look at the power consumption of the other two components (forwarding unit and SCU). Both components differ between the fine-grained design and the coarse-grained design. Therefore their power estimations are different, and discussed in detail in the sections below.

Power Consumption in the Fine-Grained Threaded Processor

To get a general idea of power distribution in the processor, we have conducted an experiment. Table 4.1 shows the power consumption of different components, generated by the Synopsys Design Compiler[18] for a six-stage pipelined processor. The component power consumptions for the baseline design are given in Columns 2-4, and the power consumptions for the 2-thread switching control and 3-thread switching control are shown in Columns 5&6. The relative power consumptions as compared to the baseline design are presented in the last row. As can be seen from the experiment, the forwarding unit consumes a considerable amount of the processor power and should be targeted for reduction.

	Plain (p_0)	Reg File	FW Unit	FG.SCU (2thrd)	FG.SCU (3thrd)
Power(mw)	1.016	0.494	0.235	0.073	0.103
Rel.Pow(%)	57.51	27.95	13.28	4.14	5.83

Table 4.1: Power Consumption of Different Components

With the FG threading design, the instruction dependency in the pipeline is decreased. Therefore, the forwarding unit can be reduced. Here we first discuss the forwarding unit reduction, then based on which derive the power consumption for the FG multi-threaded processor.

In terms of the forwarding unit design, the stages in the processor pipeline can be divided into five groups: (1) stages that generate values to be stored in the register file, we refer to such a stage as **source stage**; (2) stages in which the values generated in the pipeline are saved to the register file, we call such a stage **destination stage**; (3) stages that take values in the register file as their computing operands, we thus call such stages **user stage**; (4) stages that pass the register values in the pipeline to the user stage, we call such a stage **forwarding stage**; and (5) the rest stages that are irrelevant to forwarding, and are grouped into **irrelevant stage**. Any stage, from the source stage and before the destination, can be a forwarding stage. Figure 4.1 illustrates an m-stage pipeline. Three stages, with their nickname displayed in the pipeline, give an example of different stage types. Stage k_2 ($ex1$ stage) is an execution stage that computes on register values and generates results to be stored in the register file in stage k_3, wb stage. Stage k_1 ($brch$ stage) deals with the condition checking on register values for branch instructions. Stage k_3 is a destination stage, stage k_2 is a source stage as well as a forwarding stage, and both k_1 and k_2 are user stages. A forwarding path is required if a register value used by a user stage, has been generated by the source stage, but yet arrived to the register file in the destination stage.

We can group the forwarding paths according to the number of stages they pass backwards in the pipeline, and we call the number of passed stages the **length** of the path. We use FW_i to represent all forwarding paths that are of length i, as abstracted in Figure 4.2. If the maximal distance of dependent

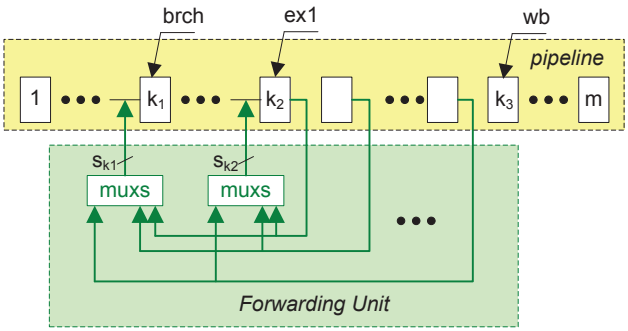


Figure 4.1: Forwarding Paths

instructions is d, the forwarding unit may contain components from FW_1 up to FW_d.

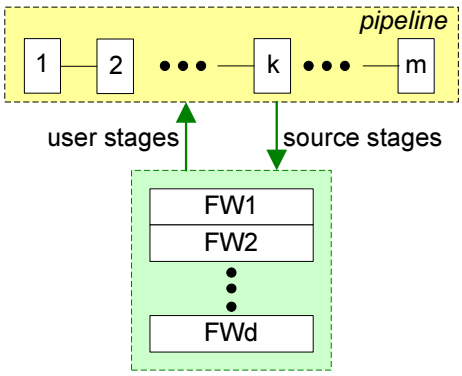


Figure 4.2: Forwarding Unit

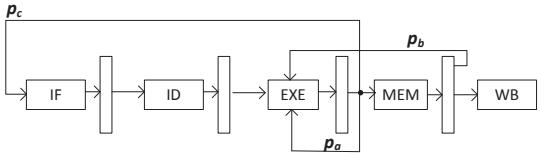
If there are no dependent instructions with distance $i(1 \leq i \leq d)$, then component FW_i can be totally removed from the forwarding unit.

In general, given the longest forwarding path d, n threads will reduce the number of forwarding groups from d to $\lfloor \frac{d}{n} \rfloor$, which gives the required forwarding paths as $\sum_{i=1}^{\lfloor \frac{d}{n} \rfloor} FW_{i \times n}$, where $\lfloor x \rfloor$ represents the floor function, which gives the largest integer not greater than x.

To explain, we take a five-stage pipeline as an example, its full forwarding paths are shown in Figure 4.3(a). In this pipeline, the source stage is EXE, destination stages are WB, the forwarding stages are EXE and MEM, and the user stages are IF and EXE. There are three forwarding paths, P_a, P_b, and P_c. The length of the longest forwarding path is 3 (P_c). Figure 4.3(b) summarized the needed forwarding components for 1-thread, 2-thread and 3-thread FG interleaved pipelines.

With the forwarding unit reduced, the power of forwarding unit in an FG n-thread processor can be approximated as:

$$P_{FW} = \frac{\lfloor \frac{d}{n} \rfloor}{d} \times p_{fw}, \tag{4.5}$$



(a)

	FW1	FW2	FW3
1-thread	Pa	Pb	Pc
2-thread	0	Pb	0
3-thread	0	0	Pc

(b)

Figure 4.3: Forwarding Reduction Example

where p_{fw} is the power consumption of the full forwarding unit, d the longest forwarding length, and n the number of threads in the design.

Like the register file consumption, the power consumption of SCU also increases with the number of threads, as given below:

$$P_{FG.SCU_n} = P_{FG.SCU_0} + P_{FG.SCU_1} \times (n-1) \quad (n \geq 2), \tag{4.6}$$

where $P_{FG.SCU_0}$ is the base cost of SCU design and $P_{FG.SCU_1}$ the extra SCU power consumption per thread for the FG design.

Finally, the total power consumption of the FG n-thread pipeline can be estimated as

$$P_{FG_n} = \begin{cases} p_0 + p_{rf} + p_{fw} & \text{if } n = 1 \\ p_0 + n \times p_{rf} + \frac{\lfloor \frac{d}{n} \rfloor}{d} \times p_{fw} + & \\ +P_{FG.SCU_0} + P_{FG.SCU_1} \times (n-1) & \text{if } n > 1 \end{cases} \tag{4.7}$$

Formula (4.7) shows that increasing number of threads leads to more power consumption in the register file and at the same time, reduces the power consumption by the decreased forwarding unit in the fine-grained threading design.

Power Consumption in the Coarse-Grained Threaded Processor

In the CG multi-threaded processor, the SCU controls to fetch instructions from threads that are ready for execution. A set of registers to hold the execution state for each thread and the related logic circuit are required for checking and updating the state of each thread, and for finding the target for a thread switching.

Compared to FG design, the SCU is more complicated in the CG multi-threaded processor and takes more power consumption, as illustrated by the experiment data given in Table 4.2. The table lists the power consumption of SCU in the 2-thread and 3-thread FG designs (Column 2 and 4, respectively), and the related power consumptions in the CG designs (Columns 3 & 5). The last row gives their relative values as compared to the baseline processor design. As can be seen from the table the power consumption of the SCU in the CG design is about doubled as compared to that in the FG design.

	2-thread		3-thread	
	FG	CG	FG	CG
Power (mW)	0.073	0.145	0.103	0.199
Relative Power	4.14%	8.18%	5.83%	11.23%

Table 4.2: Comparison of SCU Power Comparison

For the CG SCU design, each thread requires an equal size of registers for thread switching control. The power consumption of the SCU increases with the thread number, and the n-thread SCU power consumption can be approximated as:

$$P_{CG.SCU_n} = P_{CG.SCU_0} + P_{CG.SCU_1} \times (n-1) \quad (n \geq 2), \tag{4.8}$$

where $P_{CG.SCU_0}$ is the base cost of SCU design and $P_{CG.SCU_1}$ the extra SCU power consumption per thread for the CG design.

Since a block of instructions from a thread will be consecutively executed in the CG design, data dependency between instructions in the block requires the full forwarding unit in place. Therefore, the power consumption of the CG based n-threaded processor can be approximated as

$$P_{CG_n} = \begin{cases} p_0 + p_{rf} + p_{fw} & \text{if } n = 1 \\ p_0 + n \times p_{rf} + p_{fw} + & \\ +P_{CG.SCU_0} + P_{CG.SCU_1} \times (n-1) & \text{if } n > 1 \end{cases} \tag{4.9}$$

where p_{fw} is the power consumption of the full forwarding unit as in the baseline processor design. Formula (4.9) shows that the power consumption from the CG design will increase when the thread number increases.

4.2 CPI

In the pipelined processor, the execution time of an application consists of two types of clock cycles: **execution clock cycles** that are actually used to execute the application, and the **idle clock cycles** when the pipeline is stalled due to the hazards in the pipeline. We use U to denote the actual clock cycles used for execution and W the total waste clock cycles when the pipeline is in the idle state. For a typical pipelined processor, without pipeline stall one instruction takes one clock cycle to complete. Therefore, U can be regarded as the number of instructions executed. The average clock cycles per instruction can, therefore, be calculated as

$$\begin{aligned} CPI &= (U+W)/U \\ &= 1 + R_{idle}, \end{aligned} \tag{4.10}$$

where $R_{idle} = W/U$ is the **pipeline idle rate**.

With the multiple thread execution, the idle clock cycles of one thread can be taken to execute instructions from other threads. Therefore, the idle clock cycles in the pipeline can be reduced, hence the CPI is decreased.

Assume there are k types of hazards ($h_1, \cdots, h_k$) that can happen in the pipeline[1], and the idle time incurred by the h_i hazard is ι_{h_i}; We call the idle time **hazard latency**. We use p_{h_i} to represent the frequency of hazard h_i over the whole executed clock cycles. The idle rate is closely related to the hazard

[1]The hazard types may vary from one pipeline design to another. Here our multi-threaded processors are built on the baseline design. They have a fixed set of hazard types.

frequencies and their latencies, which will be discussed in detail in the next two subsections.

CPI in the Fine-Grained Multi-Threaded Processor

With the FG threading design, the processor switches to different threads on each clock cycle. Its execution is illustrated in Figure 4.4(a), where a block denotes a clock cycle and the thread being executed in the cycle is given in the block. There are n threads, and each thread gets executed every n clock cycles.

If the latency of a hazard is larger than the number of threads, the hazard will result in pipeline stall. Take the execution of 4-thread processor given in Figure 4.4(b), as an example. If thread A causes a hazard with the latency of 10 clock cycles, the pipeline will stall for the next 2 clock cycles when the thread gets its turn to execute.

In general, the FG threading design can reduce the pipeline idle time, related to hazard h_i, from ι_{h_i} to $\lfloor \frac{\iota_{h_i}}{n} \rfloor$.

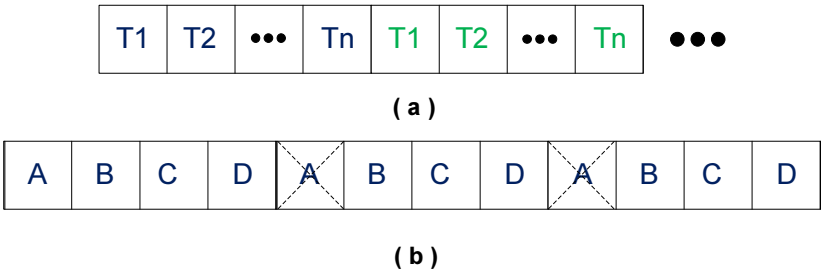


Figure 4.4: Fine-Grained Multi-Thread Execution

The total idle time of the n-thread processor can be calculated as

$$W = U \times \sum_{j=1}^{k} (p_{h_j} \times \lfloor \frac{\iota_{h_j}}{n} \rfloor), \tag{4.11}$$

where U is the number of instructions executed, p_{h_j} the frequency of hazard h_j over U, and k the number of hazard types.
Based on Formula (4.11), the CPI for the FG threaded processor can be written as

$$\begin{aligned} CPI &= (U + W)/U \\ &= 1 + \sum_{j=1}^{k} (p_{h_j} \times \lfloor \frac{h_i}{n} \rfloor) \end{aligned} \tag{4.12}$$

Formula (4.12) shows that the high the hazard rate and the longer the hazard latency, the larger the CPI. But CPI can be reduced if more threads are available.

CPI in the Coarse-Grained Multi-Threaded Design

For the CG multi-threaded processors, the thread switching is often based on the occurrence of pipeline stall and flush. The pipeline stall and flush are incurred by some types instructions, such as instructions for memory access and

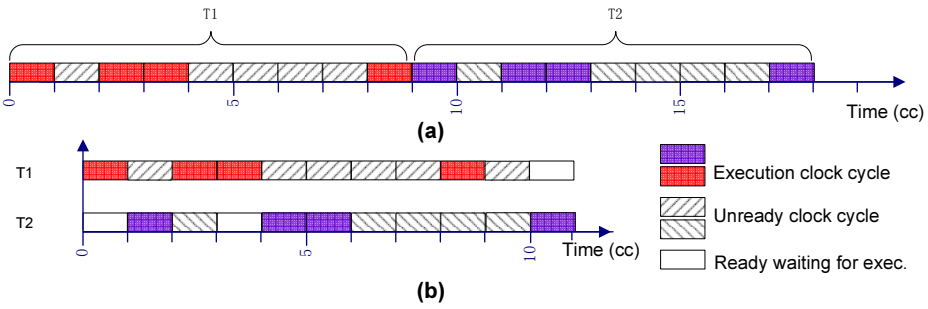


Figure 4.5: Execution in Coarse-Grained Design

for branching. We call such instructions bf hazard risk (HR) instruction. A HR instruction may and may not cause pipeline stall, depending on the real execution situation. For example, a memory load instruction may not cause pipeline stall if the memory data is cached and the cache access takes only one clock cycle.

The thread switching can be controlled statically or dynamically. With the static approach, the pipeline execution will switch to other thread whenever a HR instruction is encountered; while by the dynamic approach, the thread switching happens only when a pipeline hazard actually appears. The static approach is simple and easy to implement. Our model is based on the static approach.

The static thread switching can be implemented without extra performance overhead, as has been discussed in [7][11]. Therefore, in our model, we assume it takes zero clock cycles to switch from one thread to another. We also assume that if a **HR** instruction in a thread initiates thread switching, this thread is put in **unready state** and should wait at least ι_{h_j} clock cycles before it changes back to ready stage, where h_j is the hazard associated with the HR instruction and ι_{h_j} the hazard latency.

In CG processor, the pipeline is idle only when all threads are in unready state. Take the two-thread execution shown in Figure 4.5, as an example. Both threads each have 4 execution clock cycles (denoted by the colored blocks) and 5 clock cycles (shaded blocks) in the unready state. If executed in sequence, the two threads will have 18 unready clock cycles, as shown in Figure 4.5(a). Figure 4.5(b) shows the coarse-grained threaded execution. For each thread, there are three kinds of states: 1)execution (shown in colored blocks), 2) unready (in shaded blocks), and 3) ready waiting for execution (blank blocks). The pipeline is in the unready stage only when two threads are unready, which is in clock cycles 7, 8, 10.

We use C_{ur_0} to represent the number of unready clock cycles of a single thread and $C_{ur_{nt}}$ the unready clock cycles of n threads. The unready rate, R_{ur}, is defined as the ratio of the number of the unready clock cycles over the whole execution time.

For a single thread execution (only one thread is solely executed in the pipeline), the number of unready clock cycles is

$$C_{ur_0} = U \times \sum_{j=1}^{k} (f_{HR_j} \times \iota_{h_j}), \tag{4.13}$$

where U is the number of executed instructions in one thread, k is the number of different HR instruction types, f_{HR_j} is the frequency of the j-type HR instructions over the executed clock cycles number (U), and ι_{h_j} is the hazard latency of the related instruction.

For simplicity, here we assume all threads concurrently execute a same application with a similar execution behavior.

The unready rate of n threads is

$$R_{ur_{nt}} = \frac{C_{ur_{nt}}}{C_{ur_{nt}} + n \times U}. \tag{4.14}$$

From Formula (4.14), the unready clock cycles of the n threads is

$$C_{ur_{nt}} = \frac{n \times U \times R_{ur_{nt}}}{1 - R_{ur_{nt}}}. \tag{4.15}$$

The unready rate of single thread is

$$\begin{aligned} R_{ur_0} &= \frac{U \times \sum_{j=1}^{k}(f_{HR_j} \times \iota_{h_j})}{n \times U + C_{ur_{nt}}} \\ &= \frac{U \times \sum_{j=1}^{k}(f_{HR_j} \times \iota_{h_j})}{n \times U + \frac{n \times U \times R_{ur_{nt}}}{1 - R_{ur_{nt}}}} \\ &= \frac{\sum_{j=1}^{k}(f_{HR_j} \times \iota_{h_j})}{n + \frac{n \times R_{ur_{nt}}}{1 - R_{ur_{nt}}}}. \end{aligned} \tag{4.16}$$

The unready rate of the n threads can also, statistically, be

$$R_{ur_{nt}} = (R_{ur_0})^n. \tag{4.17}$$

With Formula (4.16) and Formula (4.17), we have

$$\frac{n^n \times R_{ur_{nt}}}{(1 - R_{ur_{nt}})^n} = (\sum_{j=1}^{k}(f_{HR_j} \times \iota_{h_j}))^n. \tag{4.18}$$

We can then use Equation (4.18) to estimate the unready rate of n threads[2].

With the idle rate, we can in turn estimate the CPI for a multi-threaded pipeline design based on Formula (4.10), as given below:

$$CPI = 1 + R_{ur_{nt}}. \tag{4.19}$$

5 Experimental Results

To verify the estimation models, we built hardware processors of different multi-threads, based on the PISA [14] ISA (Instruction Set Architecture). A six-stage pipeline was implemented for each of the processors: Instruction Fetch stage, Instruction Decode stage, Execution stage, two memory access stages; and the last Write Back stage. ASIPMeister [15] was used to generate the VHDL model for the processor, and six applications from the Mibench testbench suite [16] were selected in the experiment. The experimental setup and design flow are shown in Figure 5.1.

[2]It is worth to point out that there may not be an algebraic solution of the unready rate for a given n value. In that case, numeric approach with some math tool can be used.

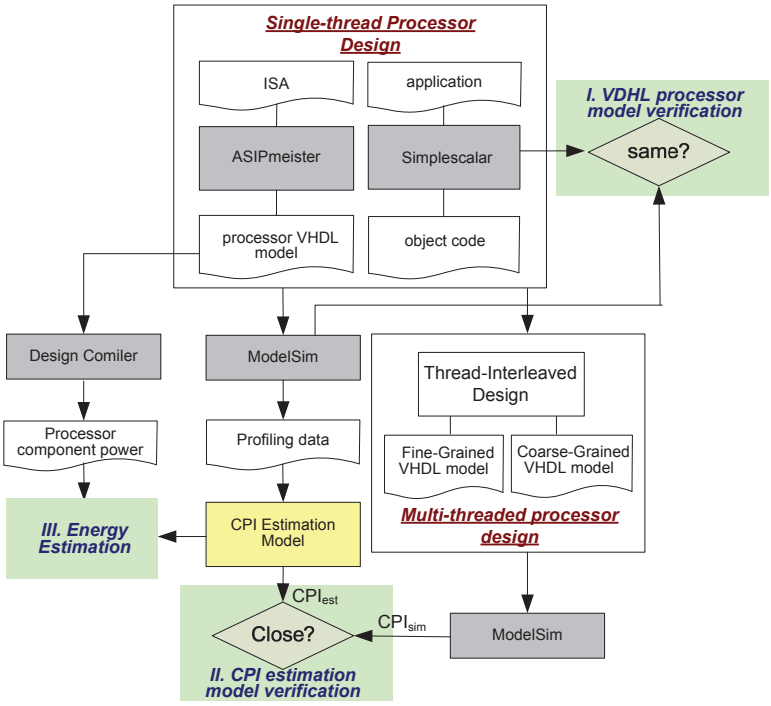


Figure 5.1: Experimental Setup and Design Flow

For an application written in C, it is compiled with the Simplescalar GCC [14]. The execution of the application on the processor model is simulated with Modelsim [17], and its results are compared with the Simplescalar simulation results for functional verification of the processor design. The execution trace from Modelsim is used in profiling the HR instruction frequencies and the hazard frequencies, based on which, the CPI and the power consumption are estimated. The Design Compiler [18] estimates the power consumption of the processor based on the 65nm technology. The experiments results are given in the next section.

5.1 Power Estimation Verification

With the baseline processor, we built the hardware models for the 2- and 3-thread multi-threaded processors, each of two different designs: the FG based and the CG based. The baseline register file is replicated and dedicated to each thread. In the FG based design, the forwarding unit is reduced according to the thread number. Table 5.1 shows the relative power consumption for the multi-threaded designs as compared to the baseline design. The estimated values given in Row 2 are calculated with Formula (4.9) and Formula (4.7). Row 3 shows the simulation values from the Design Compiler [18]. The percentage differences between the estimation and simulation values are given in the last row. As can be seen from the table, the differences are small. On average, there is a 3.26% difference for the power estimation when compared to the result from Design

Compiler.

	FG		CG		AVG
	2thread	3thread	2thread	3thread	
EstVal	2.195	2.680	2.384	2.932	
SimVal	2.304	2.750	2.441	3.022	
Diff.	4.95%	2.62%	2.39%	3.06%	3.26%

Table 5.1: Relative Power Consumption from Estimation and Simulation

5.2 CPI Verification for CG Designs

Unlike the CPI Formula 4.12 for the FG design, approximations have made in the derivations of the CPI for CG designs. To check how the approximations affect the estimation, we also run CPI comparisons between estimation and simulation.

With the PISA ISA and the processor implementation, there are pipeline hazards that related to two types of instruction dependencies: data dependency on memory access, and control flow dependency on branch condition. The hazard frequency is made of the memory access rate and branching rate.

The profiling data for each of the applications in our experiment on the baseline pipeline are presented in Table 5.2, where Row 2 gives the total number of instructions executed. The number of executed branch instructions and memory accesses are given in Rows 3-4, respectively. The branching and memory access rates over the total number of instructions are given in the last two rows.

	Sha	Qsort	Bitcount	Dijkstra	Strsearch	Aes	BSort
Exec. instr. number	19912	24336	14812	21710	14436	36966	14244
Branch number	3718	5568	4061	3571	2579	75157	2631
Mem-access number	2339	3545	91	3112	411	5995	2535
Branch rate	0.187	0.229	0.274	0.164	0.179	0.203	0.185
Mem-access rate	0.117	0.146	0.006	0.143	0.028	0.162	0.178

Table 5.2: Profiling Data

The CPI values are shown in Table 5.3. Here, the hazard length for memory access is 4cc and for the conditional branching is 2cc. Both the branch and memory access rates from profiling (shown in Table 5.2) are used in the CPI estimation and the calculated results from Formula (4.18 and 4.10) are presented in the second data section (rows 6 to 7) in the table. The simulation results are in the first section (rows 3 to 4), and the differences between the estimated value and simulation result (in percentage) are given in the last data section; The last column in this section shows the average value. As can be seen from the table, on average, there are, respectively, 2.94% and 2.52% differences for the 2-thread and 3-thread designs.

To see the impact of memory latency on the estimation model, we change the memory delay in the designs. Table 5.4 and Table 5.5 show the CPI with the memory delay of 8cc and 14cc, respectively. From Table 5.4, we see the average discrepancies of 3.62% and 3.95%, respectively in the 2-thread and 3-thread CG designs when the memory latency is 8cc, and the discrepancies slightly increase to 3.88% and 6.49% as the memory delay is raised to 14cc (shown in Table 5.5).

In summary, both the power estimation model and the CPI estimation model offer results very close to the simulation and therefore can be used to estimate

	Sha	Qsort	Bitcount	Dijkstra	Strsearch	Aes	BSort	AVG
simulation value								
2-thread	1.150	1.194	1.010	1.138	1.038	1.179	1.095	
3-thread	1.064	1.082	1.004	1.053	1.015	1.056	1.059	
estimated result								
2-thread	1.124	1.173	1.066	1.151	1.051	1.182	1.187	
3-thread	1.018	1.031	1.006	1.023	1.004	1.034	1.035	
estimation discrepancy								
2-thread	2.20%	1.70%	5.57%	1.18%	1.21%	0.32%	8.38%	2.94%
3-thread	4.33%	4.69%	0.24%	2.89%	1.06%	2.11%	2.29%	2.52%

Table 5.3: CPI in Coarse-Grained Designs (with 4cc memory latency)

	Sha	Qsort	Bitcount	Dijkstra	Strsearch	Aes	BSort	AVG
simulation value								
2-thread	1.364	1.474	1.022	1.380	1.090	1.489	1.445	
3-thread	1.161	1.207	1.009	1.103	1.039	1.184	1.121	
estimated result								
2-thread	1.279	1.380	1.071	1.355	1.077	1.417	1.446	
3-thread	1.065	1.109	1.007	1.092	1.007	1.124	1.140	
estimation discrepancy								
2-thread	6.20%	6.40%	4.84%	1.82%	1.21%	4.83%	0.07%	3.62%
3-thread	8.33%	8.18%	0.24%	1.06%	3.08%	5.05%	1.68%	3.95%

Table 5.4: CPI in Coarse-Grained Designs (with 8cc memory latency)

	Sha	Qsort	Bitcount	Dij	Strsearch	Aes	BSort	AVG
simulation value								
2-thread	1.663	1.866	1.037	1.720	1.169	1.848	1.964	
3-thread	1.328	1.435	1.019	1.301	1.083	1.407	1.250	
estimated result								
2-thread	1.557	1.733	1.080	1.713	1.125	1.820	1.889	
3-thread	1.190	1.296	1.008	1.270	1.015	1.342	1.387	
estimation discrepancy								
2-thread	6.37%	7.13%	4.15%	0.41%	3.76%	1.52%	3.82%	3.88%
3-thread	10.37%	9.72%	1.08%	2.40%	6.28%	4.59%	10.98%	6.49%

Table 5.5: CPI in Coarse-Grained Designs (with 14cc memory latency)

the energy consumption of a design, which is given in the following section.

5.3 Energy Efficiency Analysis and Multi-threaded Processor Evaluation

To evaluate a multi-threaded processor, we estimate its energy based on the average power consumption and CPI of a given set of applications.

For our given 6-stage baseline processor, here in the experiment we simply examined processors with the thread number range from two to six (more designs with even higher thread numbers can be easily evaluated in the same way based on the estimation models proposed). Based on Formula (4.7) and Formula (4.9), the relative power as compared to the baseline design of different thread number for the FG and CG designs are estimated and given in Table 5.6.

Table 5.7 shows the average CPI of 7 applications of different designs. Here, the branch delay always takes 2cc and the memory delay varies from 2cc to 10cc, as given in Rows 2-3. Row 5 shows the average CPIs with the baseline processor execution, while Rows 7-11 show the CPIs of the CG design. The CPIs of the FG design are given in Rows 13-17.

With the above relative power consumption (Table 5.6) and CPI, we can therefore estimate the energy costs of different processor designs, which are

Relative Power (times)		
	Coarse-grained	Fine-grained
2-thread	1.349	1.242
3-thread	1.659	1.516
4-thread	1.969	1.802

Table 5.6: Relative Power of Multi-Threaded Processor Over the Baseline Processor

Pipeline Characteristics									
Memory Delay	2cc	3cc	4cc	5cc	6cc	7cc	8cc	9cc	10cc
Branch Delay	2cc	2cc	2cc	2cc	2cc	2cc	2cc	2cc	2cc
CPI estimation									
1-thread	1.505	1.618	1.731	1.844	1.957	2.070	2.184	2.297	2.410
Coarse-grained CPI									
2-thread	1.073	1.102	1.136	1.172	1.211	1.251	1.294	1.337	1.382
3-thread	1.008	1.014	1.022	1.032	1.045	1.061	1.079	1.099	1.120
4-thread	1.001	1.002	1.003	1.004	1.007	1.011	1.016	1.022	1.030
5-thread	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.002	1.003
6-thread	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
Fine-grained CPI									
2-thread	1.253	1.253	1.366	1.366	1.479	1.479	1.592	1.592	1.705
3-thread	1.000	1.113	1.113	1.113	1.226	1.226	1.226	1.339	1.339
4-thread	1.000	1.000	1.113	1.113	1.113	1.113	1.226	1.226	1.226
5-thread	1.000	1.000	1.000	1.113	1.113	1.113	1.113	1.113	1.226
6-thread	1.000	1.000	1.000	1.000	1.113	1.113	1.113	1.113	1.113

Table 5.7: CPI Estimation

Pipeline Characteristic									
MemDly	2cc	3cc	4cc	5cc	6cc	7cc	8cc	9cc	10cc
BrchDly	2cc	2cc	2cc	2cc	2cc	2cc	2cc	2cc	2cc
Energy estimation									
1 thread	1.505	1.618	1.731	1.844	1.957	2.070	2.184	2.297	2.410
Coarse-grained									
2 thread	**1.447**	**1.486**	**1.532**	**1.581**	**1.633**	**1.688**	**1.745**	**1.803**	1.863
3 thread	1.671	1.681	1.695	1.713	1.734	1.760	1.790	1.822	**1.857**
4 thread	1.970	1.972	1.974	1.978	1.983	1.990	2.000	2.013	2.028
5-thread	2.279	2.279	2.279	2.279	2.279	2.279	2.279	2.283	2.285
6-thread	2.589	2.589	2.589	2.589	2.589	2.589	2.589	2.589	2.589
Fine-grained									
2 thread	1.556	1.556	1.696	1.696	**1.837**	**1.837**	1.977	**1.977**	2.117
3 thread	1.516	1.688	**1.688**	**1.688**	1.859	1.859	**1.859**	2.031	**2.031**
4 thread	1.802	1.802	2.005	2.005	2.005	2.005	2.209	2.209	2.209
5-thread	2.091	2.091	2.091	2.328	2.328	2.328	2.328	2.328	2.564
6-thread	2.383	2.383	2.383	2.383	2.653	2.653	2.653	2.653	2.653

Table 5.8: Energy Estimation

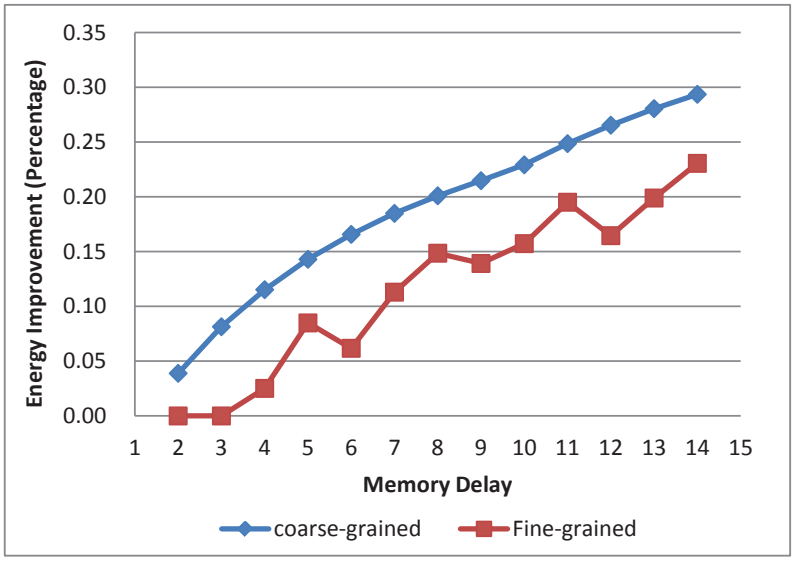


Figure 5.2: Energy Improvement

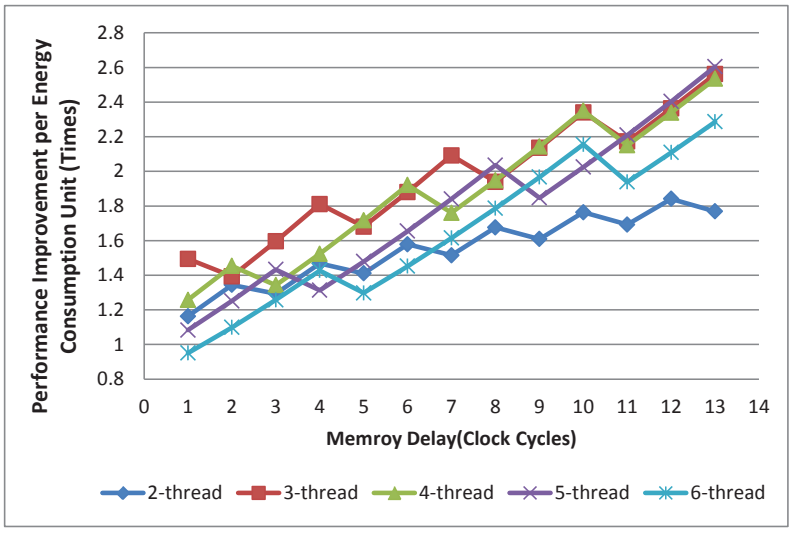


Figure 5.3: Performance Improvement (FG)

shown in Table 5.8. In Table 5.8, Row 5 gives the relative energy of a single-thread processor. Rows 7-11 and Rows 13-17 show, respectively, the relative energy consumptions of the CG and FG designs, where the minimum energy consumption among different designs for a given memory access delay is highlighted in bold.

Comparing the energy efficiency between the multi-threaded processors (Rows 7-17) and the baseline processor (Row 5), we can see that the multi-threaded processor designs always offer a better solution. In addition, the CG design can achieve better energy efficiency than the FG design. The average energy

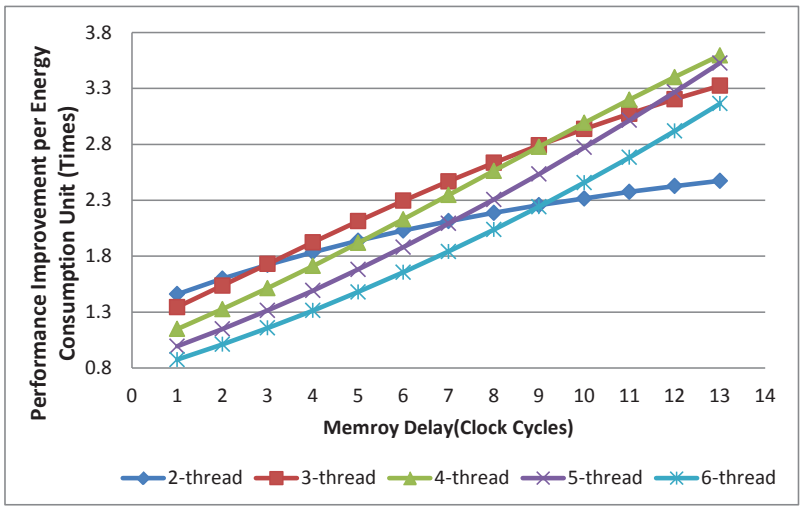


Figure 5.4: Performance Improvement (CG)

improvement(%) of both CG and FG are given in Figure 5.2, which shows the minimum energy consumption for different memory access delays.

The performance improvement per energy unit consumption (PIPE) of different processors with varied memory access delays for different number threads designs are shown in Figures 5.3 (FG based designs) and 5.4 (CG based designs). As can be seen from Figure 5.3, the 3-threaded processor generally gives a best energy efficiency in the FG design. But for the CG based designs, the optimal solution varies as the memory access delay changes. The best design solution changes from the 2-threaded processor for a small memory access delay, to the 3-threaded processor, and goes further to the 4-threaded processor when the memory access delay is increased.

6 Conclusions

Estimation models for performance and energy consumption of multi-threaded processors have been proposed in this paper. The models offer the similar results as produced by the commercial design simulation tool (ModelSim), where the RTL hardware description model for each processor should be built before simulation, which is much time consuming and tedious. The models proposed in this paper enable fast design space exploration for an optimal multi-threaded processor based on a given baseline pipeline design.

Based on our experiments on the multi-threaded processor designs with different thread number (from 1 thread to 4 threads), we found that the CG based designs are better than the FG based designs in terms of energy efficiency. And for the CG designs, the optimal solution changes with the memory access delay. With our target baseline processor (PISA ISA and 6-stage pipeline), the 2-threaded processor is best for the small memory access delay ($\leq 4cc$); but the 3-threaded design appears optimal when the memory delay is in the range of 4cc to 10cc, and when the delay exceeds 10cc, the 4-threaded processor becomes

superior.

Bibliography

[1] T. Ungerer, B. Robic and J. Silc. A survey of processors with explicit multithreading. In *ACM Computing Surveys*, volume 35, issue 1, page 29-63, March, 2003.

[2] H.F. Jordan. Performance measurements on HEP-a pipelined MIMD computer. In *10th annual international symposium on Computer architecture*, volume 11, issue 3, pages 207-212, June, 1983.

[3] R.H. Halstead, Jr and T. Fujita. MASA: A multithreaded processor architecture for parallel symbolic computing. In *Computer Architecture, 1988. Conference Proceedings. 15th Annual International Symposium*, pages 443-451, June, 1988.

[4] M.R. Thistle and B.J. Smith. A processor architecture for Horizon. In *Supercomputing '88 Proceedings of the 1988 ACM/IEEE conference on Supercomputing*, pages 35-41, 1988.

[5] R. Alverson and D. Callahan. The Tera computer system. In *ICS '90 Proceedings of the 4th international conference on Supercomputing*, 1990.

[6] M. Fillo, S.W. Keckler, and W.J. Dally. The m-machine multicomputer. In *Proceedings of the 28th Annual International Symposium*, pages 146-156, 1995.

[7] W. Grunewald and T. Ungerer. A multithreaded processor designed for distributed shared memory systems. In *Advances in Parallel and Distributed Computing, 1997. Proceedings*, page 206-213, 1997.

[8] K. Kurihara, D. Chaiken, and A. Agarwal. Latency tolerance through multithreading in large-scale multiprocessors. In *Proceedings International Symposium on Shared Memory Multiprocessing*, 1990.

[9] T. Killeen and M. Celenk. Stream-interleaved pipelined RISC processor design for SIMD and MIMD system development. In *1993 (25th) Southeastern Symposium on System Theory*, pages 452–456, 1993.

[10] A. Agarwal, J. Kubiatowicz, and D. Kranz. Sparcle: An evolutionary processor design for large-scale multiprocessors. *Micro, IEEE*, volume13, issue3, page48-61, December 1993.

[11] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded sparc processor. *Micro, IEEE*, volume25, issue2, page21-29, 2005.

[12] J.S. Seng, D.M. Tullsen, and G.Z.N. Cai. Power-sensitive multithreaded architecture. *Proceedings 2000 International Conference on Computer Design*, pages 199–206, 2000.

[13] A. Gontmakher, A. Mendelson and A. Schuster. Using fine grain multi-threading for energy efficient computing. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming-PPoPP '07*, pages 259-269, 2007.

[14] D. Burger and T.M. Austin. The SimpleScalar tool set, version 2.0. *ACM Sigarch Computer Architecture News*, volum25, issue3, page 13-25, 1997.

[15] ASIP-meister. http://www.asip-solutions.com/.

[16] MR. Guthaus, JS. Ringenberg, D. Ernst, TM. Austin, T. Mudge and RB. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *4th Annual IEEE Workshop on Workload Characterization*, 2001

[17] Mentor Graphics Modelsim. http://www.synopsys.com.

[18] Synopsys Synopsys Design Compiler. http://www.synopsys.com.