State-Retentive Power Gating of Register Files in Multi-core Processors featuring Multithreaded In-Order Cores

Soumyaroop Roy, Nagarajan Ranganathan, and Srinivas Katkoori
Department of Computer Science and Engineering
University of South Florida
Tampa, FL 33620
{sroy, ranganat, katkoori}@cse.usf.edu

Abstract*—In this work, we investigate state-retentive power gating of register files for leakage reduction in multi-core processors supporting multithreading. In an in-order core, when a thread gets blocked due to a memory stall, the corresponding register file can be placed in a low leakage state through power gating for leakage reduction. When the memory stall gets resolved, the register file is activated for being accessed again. Since the contents of the register file are not lost and restored on wakeup, this is referred to as state-retentive power gating of register files. While state-retentive power gating in single cores has been studied in the literature, it is being investigated for multi-core architectures for the first time in this work. We propose specific techniques to implement state-retentive power gating for three different multi-core processor configurations based on the multithreading model: (i) coarse-grained multithreading, (ii) fine-grained multithreading, and (iii) simultaneous multithreading. The proposed techniques can be implemented as design extensions within the control units of the in-order cores. Each technique uses two different modes of leakage states:** *low leakage savings and low wake-up latency* **and** *high leakage savings and high wake-up latency. The overhead due to wake-up latency is completely avoided in two techniques while it is hidden for most part in the third approach, either by overlapping the wake-up process with the thread context switching latency or by executing instructions from other threads ready for execution. The proposed techniques were evaluated through simulations with multiprogrammed workloads comprised of SPEC 2000 integer benchmarks. Experimental results show that in an 8-core processor executing 64 threads, the average leakage savings were 42% in coarse-grained multithreading, while they were between 7% and 8% for fine-grained and simultaneous multithreading.**

***Index Terms*—CGMT, FGMT, SMT, Niagara, M5, in-order.**

I. Introduction

Advances in integrated circuit (IC) technology have helped the semiconductor industry to keep pace with Moore's law for over five decades. Due to technology scaling, the minimum feature size has continued to shrink while the chip density as well as the transistor performance have continued to improve. Thus, it has been possible to build increasingly complex processor architectures with larger on-chip caches operating at higher clock frequencies. In recent years, however, increased power dissipation of chips has emerged as a fundamental barrier that has severely restricted the upward scaling of clock frequencies for further performance improvement [1].

Apart from the benefits offered by technology scaling, advances in architectural design techniques have further improved the performance of microprocessors. Superscalar CPU architectures with multiple functional units were developed so that several instructions could be executed simultaneously within a single clock cycle. Deeper pipelines and dynamic scheduling to allow out-of-order execution of instruction streams within a single thread are employed to exploit instruction-level parallelism in the program. However, several complex hardware units such as branch predictors, issue logic, reorder buffers, etc. are needed to implement out-of-order execution, which in turn requires higher power and die area budgets. It has been reported that, with the same process technology, a new microprocessor design with performance improvement of 1.5x to 1.7x results in 2x to 3x increase in the die area and 2x to 2.5x increase in the power consumption [2]. Thus, power efficiency has become the epicenter of all design efforts from an architectural standpoint as well.

One of the major reasons for the increase in power dissipation of circuits is the drastic increase in the leakage currents in the deep submicron and nanometer technologies. It has been reported that threshold voltage and gate oxide scaling cause about 3-5X increase in the subthreshold and gate leakage currents per generation [3]. Thus, combating leakage power dissipation has become critical in nanoscale circuits, besides dynamic and short circuit power. In this work, we explore reducing leakage power in multi-core processor architectures in the context of various multi-threading configurations.

While the CPU performance has been measured in terms of the execution throughput of the single thread, lately, an alternate metric, referred to as *throughput performance*, has been gaining more prominence. Throughput performance is defined as the number of threads that can complete execution within per unit time by utilizing multiple CPU cores to perform more computations in parallel. A survey of commercially available multi-core processors can be found in [4]. As power dissipation continues to be an increasingly difficult challenge, there has been a shift in the paradigm in terms of CPU design. Instead of building a large and complex out-of-order processor, the designers are building multiple simple in-order processors within the same chip area. Each of those simple cores could further support the simultaneous execution of multiple

Digital Object Indentifier 10.1109/TC.2010.249

threads resident within the core. Such multi-core systems are commercially available applied in high-end servers, gaming platforms and embedded processors. Niagara [5] and Niagara2 [6] are multi-core general purpose microprocessors from Sun Microsystems used in high end servers that feature up to eight in-order cores. While each core in Niagara is capable of executing four threads, each core in Niagara 2 is capable of executing eight threads simultaneously. Intel's Larrabee architecture [2] for visual computing uses in-order CPU cores that support an extended version of the x86 instruction set. Each core supports execution of four hardware threads. The number of CPU cores is implementation-dependent. MIPS 1004K coherent processing system [7] is comprised of 1-4 multi-threaded cores, where each core is capable of executing two hardware threads simultaneously.

In this work, we propose a class of runtime control techniques for fine-grained state-retentive power gating in integer register files to save leakage energy in multi-core processors featuring in-order cores that support hardware multithreading. In state retentive power gating, the registers retain their states through the power gating period. In an in-order core, when a thread gets blocked due to a memory stall, the corresponding register file can be placed in a low leakage state through power gating for leakage reduction. When the memory stall gets resolved, the register file is activated for being accessed again. While state-retentive power gating in single cores has been studied in the literature, it is being investigated for multi-core architectures for the first time in this work.

The remainder of this paper is organized as follows. We review various hardware multithreading approaches and the power gating technique in Section II. The related works in the area of power gating applied to reduce leakage in processors are discussed. Section III describes the motivation for this work and a schematic overview of the proposed techniques. Sections IV, V, and VI describe the control details of register file power gating for different multithreading approaches. The experimental setup and results are presented in Section VIII. Some conclusions are offered in Section X.

II. Background and Related Work

In this section, we present an overview of various hardware multithreading approaches followed by the power gating technique. Previous works relevant to the problem being addressed in this paper are also presented in this section.

A. Hardware Multithreading

Hardware multithreading is an approach which enables a processor to support the simultaneous execution of multiple threads. A processor that supports hardware multithreading is called a *multithreaded processor*. Multithreading approaches are categorized according to how they are implemented in hardware [8] and are briefly described here:

1) Coarse-Grained Multithreading (CGMT): In this approach (Figure 1a), a thread uses all CPU resources until a long latency event, like a cache miss, a long latency operation, etc., occurs. Such an event causes a context switch and another ready thread is switched in, which runs till it encounters a long latency event. This implementation has a context switch latency associated with it. This is because, depending on the pipeline stage where the long latency event is detected (e.g., I-cache miss happens in IF-stage but D-cache miss happens in MEM-stage in a MIPS pipeline [9]), the instructions in the preceding stages are squashed, while the instructions in the succeeding stages are allowed to finish before the next thread can be run. Each thread context has a private copy of the register file, instruction fetch buffers, if any, and control logic state, while the rest of the CPU resources are shared. This approach is also known as *blocked multithreading* technique.

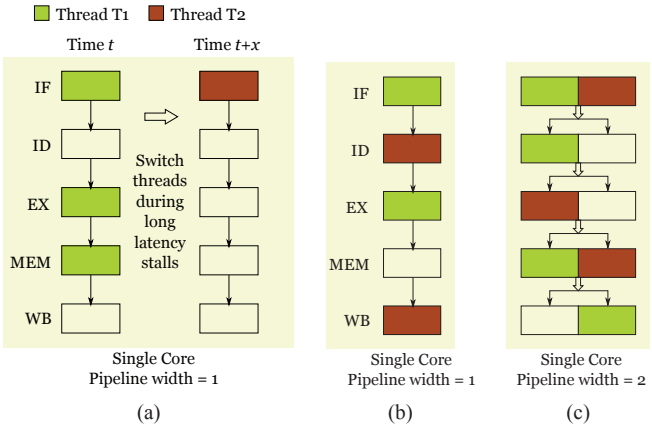


Fig. 1: Multithreading approaches: (a) coarse-grained multithreading (CGMT); (b) fine-grained multithreading (FGMT); (c) simultaneous multithreading (SMT); A 5-stage MIPS pipeline model is shown.

2) Fine-grained multithreading (FGMT): In this category (Figure 1b), thread context switching happens at the boundary of one of more clock cycles for ready threads (i.e. threads that are not blocked due to long latency events). Each thread context has a private copy of the register file and control logic state, while the rest of the CPU resources are shared. Instruction fetch-buffers may or may not be shared. FGMT is also known as *interleaved multithreading*.

3) Simultaneous multithreading (SMT): In SMT (Figure 1c), instructions from two or more threads are scheduled simultaneously on different functional units during the same cycle. SMT typically works on superscalar processors that have hardware to support simultaneous execution for two or more instructions in a single cycle. Each thread context has a private copy of the register file, instruction fetch buffers, interstage buffers, and control logic state. The rest of the resources are shared.

B. Power Gating

Power gating is a multithreshold CMOS (MTCMOS) circuit design technique that is widely used to reduce leakage power during standby periods of circuit blocks [10]. In this technique, a high threshold voltage (V_T) transistor, known as a *sleep transistor*, is inserted between the actual ground and the circuit ground, called the *virtual ground*, as shown in Figure 2. When the circuit block is idle, the sleep transistor is placed in the cut-off mode, thereby cutting off the standby leakage path (due to *stack effect*) between the supply and the ground. The circuit is referred to be in the *sleep* or *inactive* state. During this state, the virtual ground charges up to a steady state

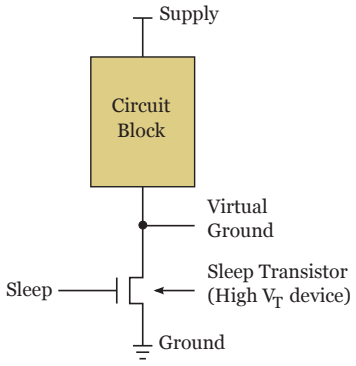


Fig. 2: Power gating with a footer sleep transistor.

value that is determined by the resistive divider formed by the other transistors in the stack. To bring the circuit back to the *active* state, the virtual ground is restored to its nominal value by placing the sleep transistor in the saturation mode. Since this requires discharging the virtual ground node to actual ground, there is a *wake-up latency* associated with it. Moreover, since the deactivation and activation of the circuit block involves discharging and charging the output capacitances of the internal circuit nodes, it restricts how often the circuit block can be transitioned between the two states to achieve overall energy savings. The period of time that the circuit block should be kept in sleep state before bringing it back to the active state so that the leakage energy savings equals the dynamic power overhead incurred circuit activation is known as the *breakeven period* [11].

The leakage reduction in circuits depend on the effectiveness of power gating which in turn is impacted by long wakeup latency and when the period of idleness is larger than the breakeven period. To improve the effectiveness of power gating, Singh et al. [12] proposed a better method called *intermediate strength power gating* in which multiple sleep states with different leakage levels and wakeup latencies are used to trade off between leakage reduction and wakeup latency. In our work, we use the intermediate strength power gating technique for state retentive power gating of the register files in the multi-core processors.

C. Related Work

Due to its runtime controllability and the magnitude of leakage savings it achieves, power gating has been widely used in reducing leakage in CPU components in both research and commercial products. It has been applied to reduce leakage in caches [13], [14] by partitioning them and putting one or more partitions to sleep when they are idle. Its effectiveness in reducing leakage in arithmetic functional units in CPU cores has been explored in numerous works [11], [15]–[21]. More recently, power gating has also been used as a primary power management technique in modern commercial processors [22], [23].

Although there is significant work reported in the literature on leakage power reduction in functional units, very few works in the literature have tried to address leakage reduction in register files. A multi-banked register file design to improve access speed and reduce total power is presented in [24], while low-leakage register files with dynamic controls have been proposed in [25], [26]. A state-retentive register file designed for the ARM processor was fabricated using 65-nm [27] technology just to study the leakage aspects of a register file in general. While this is an important work in the context of state retentive register file design for leakage reduction, the impact of its application in the context of multi-core processors has not been explored. Thus, our work described in this paper is the first such attempt, to the best of our knowledge, at investgating fine-grained leakage reduction in the context of multi-core processor architectures.

III. Proposed Approach

The motivation for our work arises from the simple observation that, in an in-order CPU, when an instruction from a particular thread encounters a pipeline stall, no further instructions from that thread (i.e., following instructions in program order) may be executed till that stall gets resolved. Thus, the thread gets blocked and the hardware units that are private to that thread could be placed in a low-leakage state. When the pipeline stall is resolved, the thread gets ready to run and those hardware units need to be brought back to the active state from the low-leakage state so that they are functional again.

As discussed earlier, the datapath components that are replicated to support hardware multithreading are the register files and buffer structures such as, instruction fetch buffers, load-store buffers, and, in some cases, pipeline registers. Among these, the register files are the largest in area and at the same time are the leakiest. For example, the SPARC architecture uses windowed integer register files with eight windows [5]. In the Niagara processor, each thread requires 128 registers for the eight windows (with 16 registers per window) and another 32 global registers, which makes a total of 160 registers per thread. Since, each SPARC core supports four hardware threads, there are a total of 640 registers in each SPARC core. Each register is 64 bits in size and there are additional bits for implementing error correcting codes (ECC). This makes each integer register file in the Niagara processor a 5 KB storage structure. If this is compared with the L1 data cache, which is private to each core in the Niagara processor and is 8 KB in size, the register file has more than 60% of the storage size of the L1 data cache. Thus, placing parts of the register files in a low-leakage state during pipeline stalls appears to be a very attractive option for saving over all leakage energy in a processor core.

Another observation is that when multiple CPU cores are required to be accommodated to build a multi-core processor, the caches that are private to each core are shrunk in size to fit the chip within a given area limit. This results in an increase in the cache miss rates experienced by each core. For instance, as mentioned earlier, each core in the Niagara processor features an 8 KB private L1 data cache which results in average miss rates of around 10% [5]. However, having four threads to run on a single core hides the latencies in stalls due to access misses from the L1 and L2 caches very effectively. Thus, as the number of cores and the number of threads per

core are increased, the fraction of time for which the integer register files for each thread stays idle due to memory stalls also increases.

The chip power breakdown for the Niagara [28] (Max 63 W @1.2 GHz, 1.2V in 90 nm CMOS) and Niagara2 [6] (Max 84 W @ 1.4 GHz, 1.1 V in 65 nm CMOS) processors show that the SPARC cores consume 27% and 31% of the total chip power, respectively. The total leakage power is about 26% and 22% of the total chip power, respectively. The rest of the power is consumed primarily by the shared L2 cache and I/O. However, the further breakdowns of the power consumption of each core into its components are not published. Therefore, we take the following approach to estimate the leakage power contributed by the register file with respect to the total power consumption of the SPARC core. The megacell specification [29] for OpenSPARC T1 (open source version of Niagara) indicates that the main components in each SPARC core are the L1 data and instruction caches, the integer register file sets (more than 60% in functional size of the L1 data cache), and the integer ALU. The unified L2 cache and the floating point unit are shared by all the eight cores. This indicates that the functional size of the register file set in each SPARC core is about 23% of the combined size of the L1 caches and the register file set. We further reasonably assume that the primary contributors of the leakage power in a core when it is consuming its peak power rating are the L1 caches and the register file set. It is estimated that the leakage power is about 40% of the core dynamic power for multi-core processors in sub 65-nm technologies [30]. According to the above estimates, the component of leakage power consumed by the register file set in each core is about 10% of the total core power. The above calculation assumes that the number of hardware thread contexts supported by each core is four (for Niagara). If the number of TCs is increased beyond four (to up to eight), this component is likely to be even larger.

To estimate the breakdown of the power consumption of the register file in terms of dynamic and leakage components, we take the following approach. We calculated that the per thread register file activity factor is about 12% writes and 25% reads averaged over all the number of register file reads and writes over all the workloads. For the above activity factor, using the power characterization tables published for 45-nm Nangate D-flip-flops in the typical corner, we estimated that the leakage power is about 71% of the total power consumed by the register file. In the slow corner, the leakage power is about 69%, while in the fast corner, the leakage power is about 82% of the total register file power.

It is important that any approach to reduce leakage based on the stalls incurred by hardware threads meet two important requirements:

1) The leakage reduction technique that is chosen to put the register file to a low-leakage state should be state retentive so that, when it is brought back to the active state, its contents are preserved.
2) Every dynamic leakage reduction technique has a performance overhead when transitioning between the active and the low-leakage states. Thus, it is important to ensure that the overhead does not negatively impact the overall performance of the processor.

In this work, we apply state-retentive power gating to save leakage in integer register files during memory stalls in multithreaded processor cores. Figure 3 illustrates the schematic view of this approach for a core which supports execution of four hardware threads. The fundamental idea is that when a memory stall (cache miss) is detected, the running thread either gets blocked or gets switched out and, therefore, its register file is placed in a low-leakage state. Eventually, when the stall gets resolved (following a cache line fill) the register file is put back in the active state.

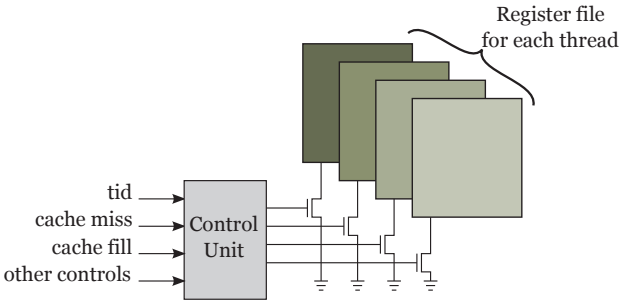


Fig. 3: Schematic view of the proposed approach.

An intermediate strength power gating technique presented in [12] is applied to characterize 32 entry 64-bit integer register file for leakage savings. The technique is also fine-grained in that the wake-up latencies are between three and five clock cycles (for a clock frequency of 1 GHz). We ensure that wake-up latency associated with this technique is effectively hidden by virtue of more than one thread sharing the CPU pipeline. Further, the register files have two distinct low-leakage states - one with lower leakage savings and lower wake-up latency; and the other with higher leakage savings and higher wake-up latency. Depending on the duration of the stall and the time between when the stall gets resolved and the register file is accessed again, it is placed is one of those low-leakage states. This is elaborated in Figure 4. The register file is designed to have two low-leakage states, *sleep1* and *sleep2*. When an L1 miss is incurred by an instruction from a particular thread, that thread's register file is placed in *sleep1* state. If the L1 fill request further experiences an L2 miss, then the register file is placed in the higher leakage savings state, *sleep2* state. When the L2 miss completes, the register file is brought back to the *sleep1* state. The wake-up latency, t_{w2}, is overlapped with the L1 fill latency and, thus, gets hidden. If, however, an L2 miss is not experienced, it continues to stay in *sleep1* state. When the L1 miss completes, the register file is brought back to the active state. The wake-up latency, t_{w1}, is hidden very effectively due to multiple threads running on the core.

The main contributions of this work are highlighted as follows:

- We propose specific techniques to implement state-retentive power gating for three different multi-core processor configurations based on the multithreading model: (i) coarse-grained multithreading (CGMT), (ii) fine-grained multithreading (FGMT), and (iii) simultaneous multithreading (SMT).

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

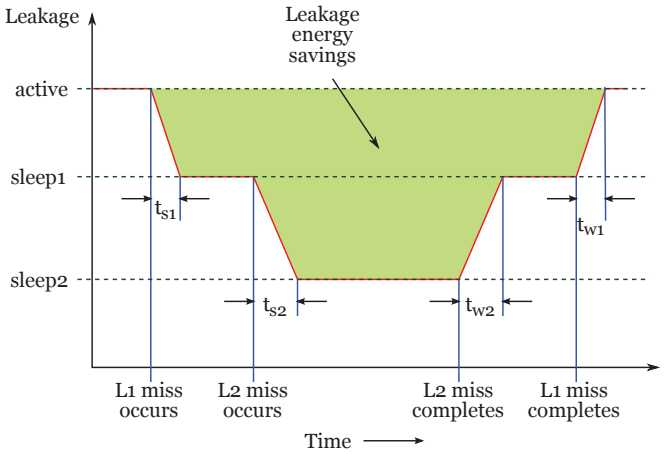


Fig. 4: Intermediate strength power gating.

- The proposed techniques can be implemented as design extensions to the control units of the in-order processor core, with incurring negligible control overhead.
- The overhead due to wake-up latency is completely avoided in two techniques while it is hidden for most part in the third approach, either by overlapping the wake-up process with the thread context switching latency or by executing instructions from other threads ready for execution.

IV. Register File Power Gating in CGMT Approach

In the CGMT approach, whenever a thread is switched, there is a multiple cycle penalty incurred due to the context switching process. The penalty is due to either squashing (or rolling back) of instructions from the pipeline or draining of the pipeline following an event that triggers the context switch. For instance, when a thread encounters a data load miss, all the instructions in the pipeline following the load instruction are squashed before a ready thread could be switched in. Conversely, in the case of an instruction fetch miss, all the leading instructions in the pipeline are allowed to finish before the next ready thread is switched in. In both the cases, bubbles are inserted into the pipeline that negatively affects the pipeline performance. A direct approach to avoid this switch penalty is to have copies of the pipeline registers at each stage, which results in increased area and complexity of the CPU core. However, for short pipelines, the context switch penalty is only a few cycles and the additional hardware does not justify the small improvement in performance (for e.g., IBM Northstar/Pulsar [9]).

In this section, we describe the timing details involved in putting an integer register file in and out of low-leakage state following a memory stall. The wake-up latency of the register file is completely overlapped with the thread switch latency discussed above. We consider a CGMT model in which thread context switching happens only during instruction fetch misses (referred to as *fetch misses* in the remainder of the text) and data load misses (referred to as *load misses* in the remainder of the text). Also, the CPU pipeline is modeled around a MIPS pipeline with instruction fetch (IF), instruction decode (ID), execute (EX), memory (MEM), and writeback (WB) stages. However, the register file is read during the first cycle in the EX-stage and then dispatched to the arithmetic functional units.

A. Power Gating Control During Fetch Miss

Figure 5 illustrates the scenario and explains the timing details of putting a register file to sleep following a fetch miss. The figure shows the state of the pipeline during clock cycle c, when thread T_1 is running while thread T_2 is in the ready state. The scenario described in this figure assumes that:

1) T_2 was switched out earlier following a fetch miss and was eventually put back in the ready state after its fetch miss completed.
2) T_2's register file is currently in a low-leakage state and needs 3 cycles to wake up before it can be accessed.
3) All the stages of the pipeline are busy executing T_1's instructions, I_{k-4} to I_k.

While fetching I_k, an instruction fetch miss is encountered following which T_1 starts to drain in cycle $(c+1)$. This is done so that instructions I_{k-4} to I_{k-1} finish before a thread context switch happens. During this draining period, T_1's register file needs to be active so that reads and writes may be performed to it. Assuming that no other instruction in the pipeline gets stalled, the last instruction, I_{k-1}, finishes in cycle $(c+3)$. During this cycle, T_2's register file is signaled to wake up. In cycle $(c+4)$, thread T_2 gets switched in and it starts to fetch instruction I_j, while T_1's register file is put to sleep. By the time I_j reaches the EX-stage in cycle $(c+6)$ and accesses T_2's register file, it is already in active state. The wake-up latency is overlapped with the context switching latency and, therefore, the pipeline does not incur any stalls due to the unavailability of the register file.

Eventually, as shown in Figure 6, T_1's fetch miss completes at cycle $(c+m)$ and T_1 switches in to the ready state. Then, we can consider waking up T_1's register file. Two

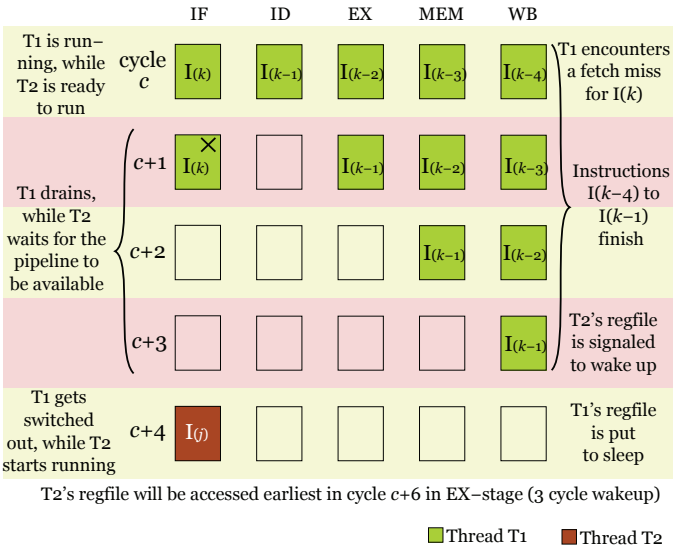


Fig. 5: Timing details for putting register files to sleep following an instruction fetch miss.

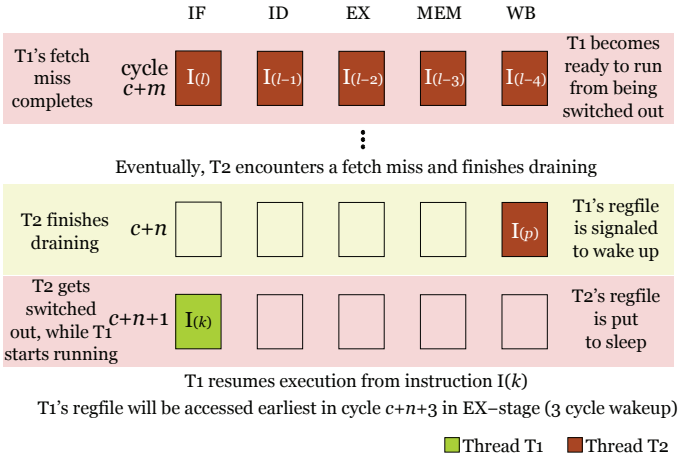


Fig. 6: Wake-up details of T_1's register file if the pipeline is busy when its fetch miss completes.

possible scenarios occur: (1) T_2 is currently running; (2) T_2 is switched out and the pipeline is currently idle. In scenario 1, there is no need to wake up T_1's register file because T_1 will get to run only after T_2 gets switched out following a memory stall (assume a fetch stall again). This situation is shown in Figure 6, when at cycle $(c+n)$, T_2 finishes draining following a fetch stall and gets switched out. T_1's register file is signaled to wake-up during this cycle. During the next cycle, T_1 resumes execution from I_k, which would need to access the register file earliest during the EX-stage. This allows the 3-cycle wake-up period to for T_1's register file. In scenario 2 (Figure 7), however, T_1 gets to run right after the fetch miss completes because T_2 is switched out and the pipeline is idle. Therefore, T_1's register file is signaled to wake up at cycle $(c+m)$. Since T_1 resumes execution (from instruction I_k) at cycle $(c+m+1)$, T_1's register file does not get accessed till cycle $(c+m+3)$ when it reaches the EX-stage. In 1, T_1's register file stays in a low-leakage state for $(n-5)$ cycles, while in 2 it stays in a low-leakage state for $(m-5)$ cycles.

B. Power Gating Control During Load Misses

During data store misses, the thread need not stall as long as the result of the store instruction can be placed in a store buffer (unless the store is part of a specialized atomic instruction). However, during a data load miss, the thread gets stalled and

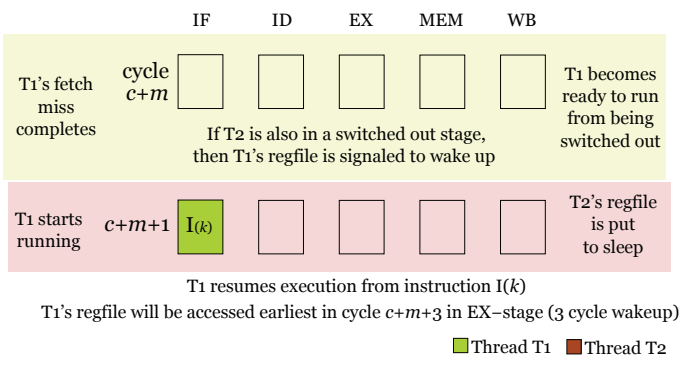


Fig. 7: Wake-up details of T_1's register file if the pipeline is idle after its fetch miss completes.

it starts the transition process towards being switched out. At the same time, its register file is placed in a low-leakage state. In case, a newly switched in thread always starts from the IF-stage (as in Niagara), then the wake-up latency of the register file can be hidden with the number of cycles that the instruction takes to traverse from the IF-stage to the EX-stage. However, if the load instruction that encounters the load miss is placed in a load buffer in the MEM-stage so that it may resume execution as soon as the load miss gets processed, then efforts are need to hide the wake-up latency. Load instructions write into the register file during the W-stage, and, therefore, the register file needs to be in active state before it can be written into.

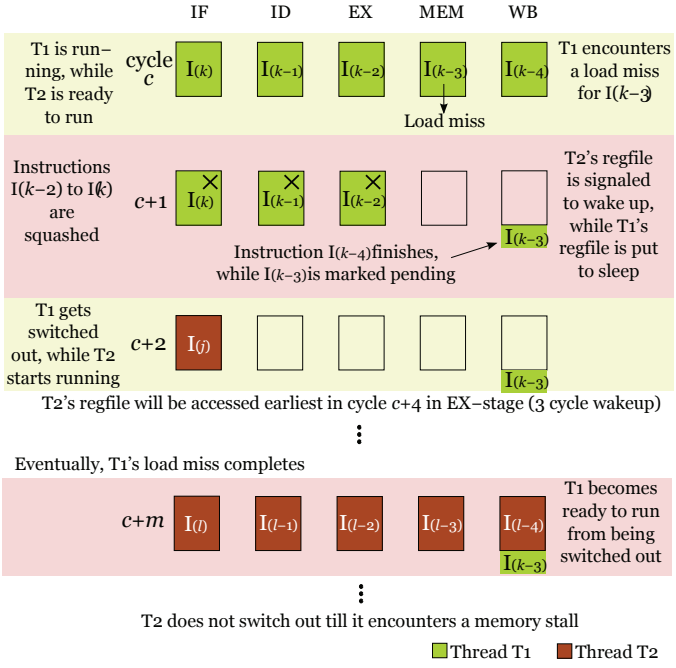


Fig. 8: Timing details for putting a register file to sleep following a data load miss.

Figure 8 shows the register file sleep strategy following a load miss. As in Figure 5, this figure also shows the state of the pipeline during clock cycle c, when thread T_1 is running and thread T_2 is in the ready state. The scenario described in this figure assumes that:

1) T_2 was switched out earlier following a *fetch miss* and was eventually put back in the ready state after its fetch miss completed.
2) T_2's register file is currently in a low-leakage state and needs 3 cycles to wake up before it can be accessed.
3) All the stages of the pipeline are busy executing T_1's instructions, I_{k-4} to I_k.

Following a load miss encountered by thread T_1 while executing instruction I_{k-3} in the MEM-stage, instructions I_{k-2} to I_k are squashed in cycle $(c+1)$, while I_{k-3} is placed in the load buffer. Since T_2 will be switched in to be executed during the next cycle, its register file is signaled to wake up, while T_1's register file is put to sleep. In cycle $(c+2)$, T_2 resumes execution by fetching instruction I_j, which does not access the register file till cycle $(c+4)$, thereby giving it the adequate number of cycles to become active.

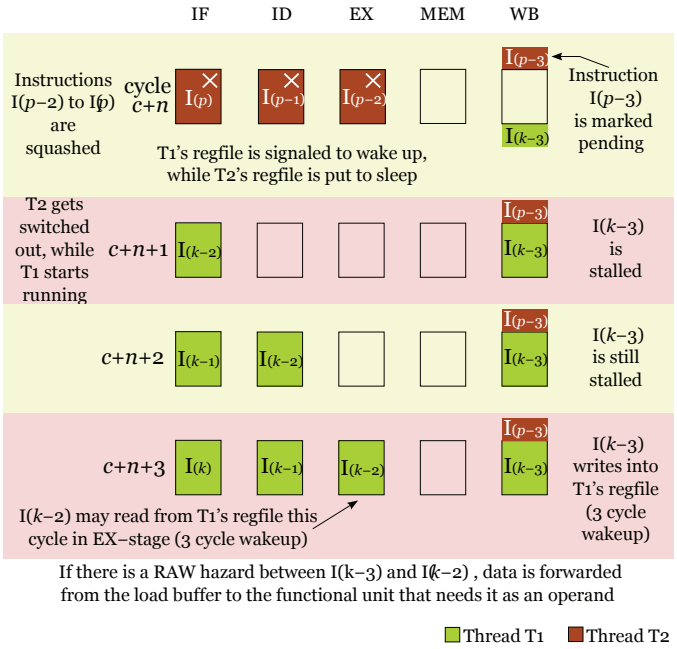


Fig. 9: Timing details for waking up T_1's register file from sleep after its load miss completes and it gets ready to run.

Eventually, when T_1's load miss completes during a later cycle, say $(c+m)$, T_1 transitions from switched out state to the ready state. Again, the decision to wake up its register file depends on whether T_2 is running (condition shown in Figure 9) or is switched out. In the former case, the register file is signaled to wake up when T_2 eventually encounters a stall (a *load miss* this time) and gets switched out (cycle $(c+n)$ in Figure 9). In the following cycle, cycle $(c+n+1)$ as shown in the figure, T_1 resumes execution from I_{k-2} in IF-stage and I_{k-3} in W-stage. Since a load instruction needs to write into the register file in the W-stage, it is stalled for three cycles to allow the 3-cycle wake-up latency needed by the register file. This timing also coincides with the earliest cycle that T_1's register file need to be accessed by I_{k-2} (when it reaches the EX-stage). If, however, there is a read-after-write (RAW) data dependency between I_{k-3} and I_{k-2}, then the result of the load operand is forwarded to the functional unit that needs it as a operand to execute instruction I_{k-2}.

V. Register File Power Gating in FGMT Approach

In contrast to the CGMT approach, FGMT and SMT approaches do not typically suffer from multiple cycle thread switch penalties. This is because, in these approaches, each pipeline stage processes one or more instructions from multiple threads. If an instruction from a thread encounters a stall, no further instructions from that thread are fetched to be dispatched to the pipeline. Instead, instructions from one or more of the ready threads are fetched and processed. The policy to select a thread to fetch from may vary across designs. For instance, Niagara uses round-robin (RR) policy to select one thread among a list of ready threads, while Niagara2 implements a least recently fetched (LRF) policy to do the same. As long as there are ready threads available to the CPU, no bubbles are inserted into the pipeline. This very idea is utilized to hide the wakeup latency of the register files when they are transitioning from the low-leakage state to the active state.

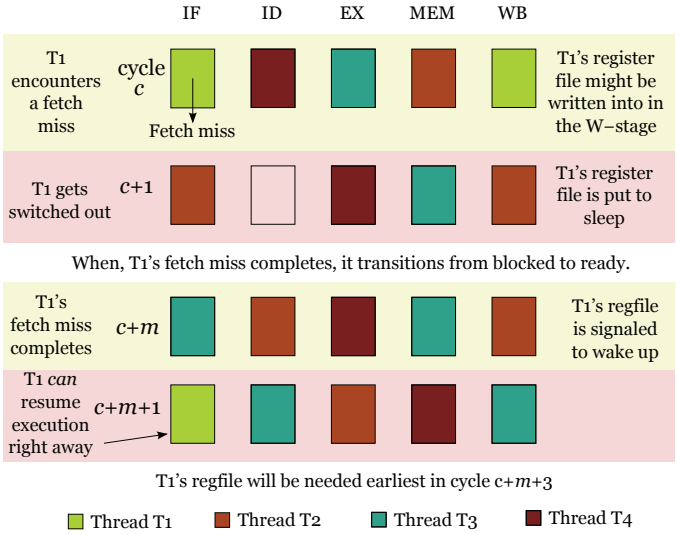


Fig. 10: Timing details for putting a thread's register file in and out of low-leakage state following a fetch miss in FGMT.

A. Power Gating Control During Fetch Miss

A CPU that is designed to support the FGMT approach, fetches an instruction from a new ready thread each cycle and dispatches it to the pipeline. Figure 10 illustrates the timing details of putting a thread's register file in and out of low-leakage state following an instruction fetch miss. It is assumed that the CPU has 4 hardware threads, T_{1-4}, and a round-robin fetch policy is implemented. The pipeline contents are shown for clock cycle c. Instruction from all four threads are currently being processed by the different stages in the pipeline when an instruction from T_1 encounters a fetch miss. Therefore, in the next cycle, $(c+1)$, one of the ready threads is selected (T_2 in the figure) and an instruction from that thread is fetched. The decision to assert the sleep control to T_1's register file depends on whether there are any instructions belonging to T_1 in the pipeline and require to access the register file. For instance, as shown in the figure, one of T_1's instructions is in the W-stage in cycle c. Therefore, it is imperative that the register file be active till that instruction finishes writing into the register file. In this case, T_1's register file is put to sleep at the end of cycle $(c+1)$.

Eventually, when T_1's fetch miss completes at cycle $(c+m)$, it becomes ready to run. T_1's register file is signaled to wake up right away. Assuming that it is indeed selected by the thread scheduler in the next cycle, i.e., cycle $(c+m+1)$, it will access T_1's register file earliest in cycle $(c+m+3)$, thereby providing for the 3-cycle wake-up latency.

B. Power Gating Control During Load Miss

In FGMT processors, when a load miss is detected for an instruction from a thread, all the instructions following the load instruction in the pipeline are squashed (or rolled back to the instruction buffer). However, since in each cycle

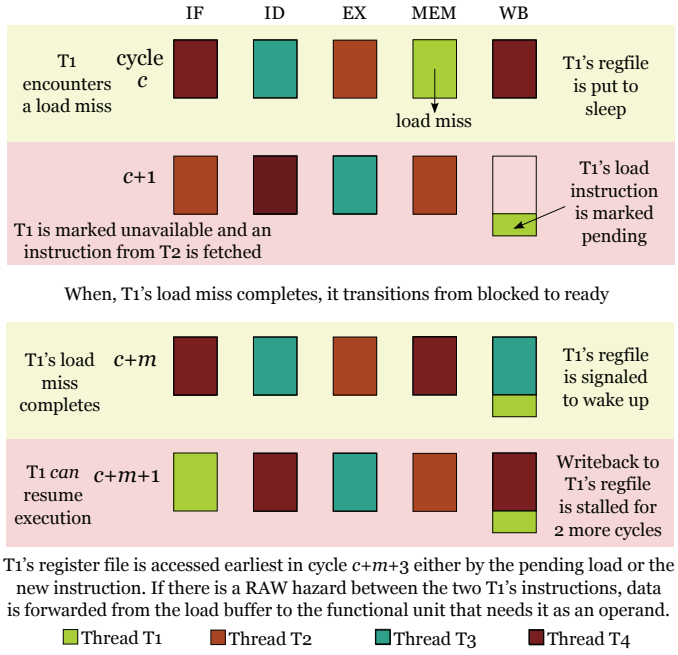


Fig. 11: Timing details for putting a thread's register file in and out of low-leakage state following a data miss in FGMT.

a different thread is dispatched to the pipeline in an FGMT approach, the number of instructions squashed is expected to be much smaller than that in the case of CGMT. The load instruction itself may also be squashed or it may be marked pending at the MEM-stage (in a load buffer). The choice of implementation here impacts the wake-up strategy applied to wake up the register file for a stalled thread when its load miss completes. In the former case (as in Niagara), the wake-up latency of the register file is overlapped with the number of cycles that the instruction takes to reach the EX-stage from the IF-stage (described earlier in Section IV-B). However, in the latter case (shown in Figure 11), the writeback to the register file is deferred for additional cycles (2 cycles in the figure) to account for the wake-latency. If there is a RAW hazard between the two T_1's instructions in the pipeline, then data is forwarded from the load buffer to the consuming instruction when it reaches the EX-stage.

VI. Register File Power Gating in SMT Approach

We model a simultaneous multithreading in-order core processor [2], [31], in which each pipeline stage is capable of processing multiple instructions from *distinct threads* during the same clock cycle. To support this capability, each pipeline stage is equipped with stage buffers for each thread context. Once an instruction is processed by the stage, it is placed in the stage buffer for its thread context for the next stage to process it in a subsequent clock cycle. However, if an instruction gets stalled at a stage due to a multicycle latency operation, like an integer multiplication or a memory stall, it is marked blocked or unavailable till the operation finishes. Along with that, all the instructions from that thread in all the preceding stages are also marked blocked. Each stage picks up instructions from only ready threads to process during a clock cycle.

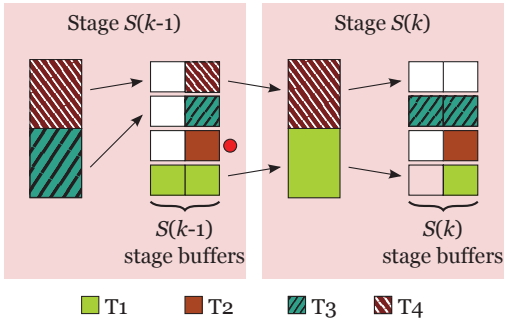


Fig. 12: Schematic view of a pipeline organization to support SMT in in-order cores.

Figure 12 shows the schematic view of this structure. It shows a snapshot of two back to back stages of the pipeline, S_{k-1} and S_k. S_{k-1} processes instructions from threads T_3 and T_4 and places them in their respective thread buffers in that stage. S_k, on the other hand, picks up instructions from threads T_1 and T_4 from S_{k-1}'s stage buffers, processes them, and places them in their respective thread buffers in this stage. Thread T_2 is shown to have been marked as unavailable (the circular shape in the figure) or stalled in S_{k-1} because it is currently performing a multicycle latency operation. Also, an instruction from T_3 cannot be processed by S_k since the buffer for thread T_3 is full in this stage.

In this SMT core, placing the register file in and out of the low-leakage state during both fetch misses and data misses is very similar. Once a miss is detected, the thread is marked blocked in *all* the stages in the pipeline so that the register file may be put to sleep immediately. Note that this is different from the regular case in which the thread is marked blocked only in the preceding stages. When the miss completes, the register file is signaled to wake up but the thread is marked ready only after a few additional cycles to account for the wake-up latency of the register file. As long as there are instructions from other ready threads in the pipeline, the additional blocked cycles do not result in any performance degradation.

VII. Discussion

A summary of the proposed techniques for CGMT, FGMT, and SMT cores is presented in Table I. In the CGMT approach, the wake-up latency of the register files is overlapped with the latency associated with the latency of thread context switching. It can also be observed that, since no more than one thread is active simultaneously, the register files for all the other threads, irrespective of whether they are stalled or ready, may be kept in low-leakage states. Thus, as the number of threads are increased in a CGMT approach, it is expected that the leakage savings in the register files also increase linearly. On the other hand, in the FGMT and SMT approaches, multiple threads are simultaneously active in the CPU at the same time. Therefore, the leakage savings achieved in these approaches are not expected to scale with the number of hardware thread contexts supported by the CPU. Instead, they are expected to be proportional to the fraction of the time that the threads spend waiting on memory stalls.

TABLE I: A Summary of the proposed techniques

Control Feature	CGMT	FGMT	SMT
Sleep after a fetch miss	Wait till the pipeline drains	Wait till there are no instructions from the target thread in the pipeline	Immediately; block all the instructions belonging to this thread in the entire pipeline
Wake up after the fetch miss completes	When the thread gets switched in (its state changes from ready to run)	As soon as the fetch miss completes	As soon as the fetch miss completes
Performance degradation due to wake-up overhead	Zero cycles; the thread resumes execution from the IF-stage	Zero cycles; the thread resumes execution from the IF-stage	Best case is zero; there are instructions available from other threads. Worst case is wake-up latency number of cycles; otherwise.
Sleep after a load miss	Immediately; any following instructions in the pipeline are squashed	Immediately; any following instructions from that thread in the pipeline are squashed	Immediately; block all the instructions belonging to this thread in the entire pipeline
Wakeup after the load miss completes	When the thread gets switched in (its state changes from ready to run)	As soon as the load miss completes	As soon as the load miss completes
Performance degradation due to wake-up overhead	Zero cycles; either the load instruction or the instruction following the load resumes execution from the IF-stage.	Zero cycles; either the load instruction or the instruction following the load resumes execution from the IF-stage.	Best case is zero; there are instructions available from other threads. Worst case is wake-up latency number of cycles; otherwise.

Further, in the CGMT and FGMT cores, the latency of putting a register file from a low-leakage state to the active state can be overlapped completely, thereby not having to incur any performance overhead. However, for the SMT cores, performance degradation can happen when there are not enough ready threads in the core and keeping a thread blocked for the additional cycles inserts bubbles into the pipeline. However, the likelihood of this scenario can be reduced by increasing the number of threads that the SMT core is able to support.

A. Control Overhead

The techniques proposed for controlling the sleep and wake-up of the register files impose a small overhead on the control logic in terms of the logical complexity involved. This is because for the following reasons:

- No additional thread states are required by the control logic to support the proposed techniques. In a CGMT core, a thread is one of *running*, *ready*, *switched out*, and *blocked*. In FGMT and SMT cores, each thread is in one of *running*, *ready*, and *blocked* states. In CGMT, the wake-up of a register file is initiated as soon as the corresponding thread transitions from *ready* to *running*. In FGMT, the wake-up is initiated when the thread transitions from *blocked* to *ready* state. Only in the SMT approach, a thread is not allowed to transition from *blocked* to *ready* state till its register file is active. This would typically be an additional 1-2 cycle because, after the cache fill is serviced, it usually takes 1-2 cycles for the instruction or the data to be forwarded to the fetch buffer or the load buffer, respectively [7], [32].
- The pipeline drain and squash mechanisms are already implemented in the control logic to support multithreading.
- Data-forwarding, which is used to supply the result of a load instruction to a following instruction that has a RAW dependency on that instruction, is also a very common technique that is supported in almost any microprocessor core.

VIII. Experimental Setup and Results

In this section, we describe the experimental setup used in this work to evaluate the effectiveness of the proposed techniques in multi-core processors.

A. Integer Register File Characterization

For the purpose of estimating leakage characteristics and the latency of a register file with intermediate-strength power gating, we consider a 32-entry 64-bit flip-flop based NOR-decoder register file (without any error correction code bits) with two read ports and one write port in 45nm FreePDK technology [33]. The layout design of a D-flip-flop from the Nangate 45 nm open cell library [34] was extended to include the two read ports and one write port. Instead of laying out the entire register file and then extracting a spice netlist with global parasitics, spice netlists with local parasitics for a 1-bit flip-flop based cell, 1-bit NOR-decoder slice, and the sleep transistor cells were extracted using the RC extraction tool, Calibre by Mentor Graphics. The extracted spice modules were structurally instantiated in spice to emulate a cell-based extraction supported by Calibre. The read/write latency, steady-state leakage, wake-up latency, and breakeven period characterization of the register file was performed using spice simulations using HSPICE. To measure the write latency, 200 randomly chosen 64-bit vectors were used to write into written into 200 randomly chosen entires in the register file. For each of the 200 instances of the writes, 100 entries were read on each of the two read ports to measure the read latency. The average latency was then computed as the arithmetic mean of the latencies over all the simulation runs. The average access latency was computed to be 0.89 ns using devices in the typical corner. The supply voltage, V_{DD} was set to 1.1 V and operating temprerature was set to 25° C. The steady-state leakage was computed by setting the virtual V_{SS} to about 170 mV for leakage state *sleep1* and about 250 mV for leakage state *sleep2* for each of the 200 functional states of the register file desribed above. We obtained steady-state leakage reductions of 36% and 52%, respectively. The

wake-up latency and the breakeven period were computed by performing a transient analysis after asserting the gate of the sleep transistors again for each of the 200 functional states of the register file described above. Their wake-up latencies, for a clock frequency of 1 GHz, were set at 3 cycles and 5 cycles, respectively. The breakeven periods were shorter than the wake-up latencies. These leakage reduction and the wake-up latencies, tabulated in Table II, are used in the experimentation phase with the architectural simulator, which is described in Section VIII-B.

TABLE II: Register File Leakage States

Leakage State	Normalized Leakage	Wakeup Latency (1 GHz Clock)
active	1	-
sleep1	0.64	3 cycles
sleep2	0.48	5 cycles

B. Processor Configurations and Workload Details

We used the M5 simulator [35] for modeling the multi-core processors featuring multithreaded CPU cores. The M5 simulator supports four different CPU models to provide simulation platforms for functional and detailed simulations. Among them, *O3CPU* models a detailed out-or-order processor core and the *InOrderCPU* models a detailed in-order processor core. The in-order code has some default support for both CGMT and SMT models. It was further extended to provide comprehensive support to model the multithreading approaches described in this paper. Since the M5 simulator, particularly the in-order code, currently has the comprehensive support for the DEC Alpha ISA in syscall emulation mode, we run all our detailed simulations for the Alpha ISA. However, we model the multi-core multithreaded processors based on the Niagara 1 core, which is a SPARC processor, without the floating point hardware.

TABLE III: SPEC 2000 Integer Benchmarks

Name	Dynamic Instruction Count (Millions)
vpr	17.6
gap	44.8
vortex	88.3
twolf	91.9
eon	94.0
crafty	94.4
gcc	96.8
mcf	188.6
perlbmk	200.6
parser	267.8
gzip	601.9

- The dynamic instruction counts are for the small reduced (smred) input sets [36].
- Benchmark `bzip2` is not shown because it does not have an smred input set.

Table III enumerates the integer benchmarks from the SPEC 2000 benchmark suite and their dynamic instruction counts for the small reduced (smred) input sets [36]. The binaries are tru64 binaries (COFF version 3.11-10) built with optimization levels O2 and O3. We took the following approach for creating the multiprogrammed workloads [37]. A multiprogrammed workload for a n-core processor, such that each core is m-way multithreaded, comprises of $n \cdot m$ SPEC 2000 integer programs. In our experimentation, $n \in 2, 4, 8$, while $2 \leq m \leq 8$ (Table IV). Since 11 of the 12 SPEC2000 integer benchmarks (excluding `bzip2`) are chosen, there are $\binom{11}{m}$ ways to construct a multiprogrammed workload for one processor such that all the binaries are distinct. All distinct binaries were chosen to create each workload to avoid the same dynamic runtime characteristics for two or more threads running on the same core. Since the simulation runtime is proportional to the total number of instructions simulated, the number of simulations was determined based on the workload size, in terms of the number of binaries. For workloads of size 40 or greater (i.e., for 8-core processor and 5-way to 8-way support for multithreading), 4 simulation runs were performed. For workloads of size 16 or more up to 32, 8 simulation runs were performed. For the rest of the workload sizes, 12 simulation runs were performed. The harmonic means were computed for the IPC and cache miss rates over all the simulation runs, while arithmetic means were computed for the leakage energy dissipated and saved (with and without power gating). Each simulation was run till the first thread finished execution.

We configure a number of multi-core processors comprising of in-order CPU cores by varying the number of cores, the number of hardware contexts that each core supports, and a number of L1 and L2 cache parameters. The processor parameters are tabulated in Table IV. The number of cores is either two, four or eight. The number of hardware threads that each core supports is scaled from two to eight in case of CGMT, from three to eight in case of FGMT, and from four to eight in case of SMT. We cap the number of threads per core at eight threads because the cost growth for supporting additional hardware threads is linear up to around eight threads but is superlinear after that [38].

The in-order cores have simple specifications. In case of CGMT and FGMT, the cores can process at most one instruction each cycle in each of its pipeline stages, while, in the case of SMT, the cores can process two. Therefore, we provide two integer ALUs to each core in case of SMT but only one integer ALU to the cores in case of CGMT and FGMT. The count of integer multipliers, however, is the same (one) for all the cores. The execution latencies of the integer ALU and integer multiplier are 1 cycle and 3 cycles, respectively. Furthermore, we model a fully pipelined integer multiplier so that integer multiply instructions that are not data dependent on each other may be issued every clock cycle. The clock frequency is

TABLE IV: Multi-core Processor Parameters

Parameter	Multithreading Approach		
	CGMT	FGMT	SMT
Clock Speed	1 GHz		
Number of Cores	2, 4, and 8		
Number of Contexts	2-8	3-8	4-8
Pipeline Bandwidth (in insts/cycle)	1	1	2
Functional Units	1 int ALU 1 int Mult	1 int ALU 1 int Mult	2 int ALU 1 int Mult
Load/Store/Fetch Buffers	1 per thread		
Fetch Select Policy	Round-robin		

uniform (1 GHz) across all the processor configurations. Each core has one load buffer, one store buffer, and one fetch buffer per thread. The policy to select a thread to fetch instructions from is set to round-robin.

TABLE V: Memory Access Latencies

Memory Unit	Access Latency
L1 D-cache	1 cycle
L1 I-cache	1 cycle
L2 cache (shared)	10 cycles
Physical Memory	30 cycles

TABLE VI: L1 D-cache and I-cache Parameters

Size	2 cores	4 cores	8 cores
	64 KB	32 KB	16 KB
Set Associativity	2 TCs	3-4 TCs	5-8 TCs
	2	4	8
MSHRs	*as many as the number of TCs*		

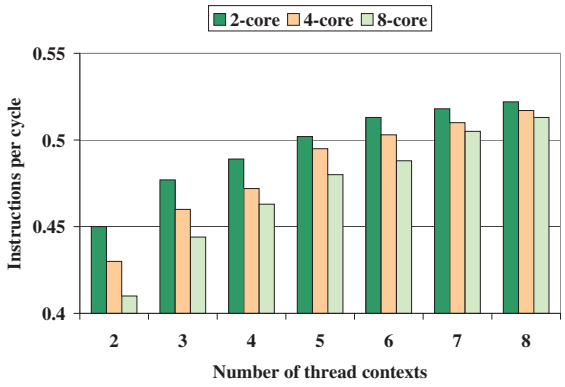
TCs - Thread Contexts

The cache parameters are tabulated in Tables V, VI, and VIII-B. We set the cache parameters based on the specifications of the Niagara series of processors. The hit latencies for the private L1 caches (both I-cache and D-cache) and the shared L2 cache are set to 1 cycle and 10 cycles, respectively. The physical memory access latency is set to 30 cycles. The cache line size for each cache is set to 64 bytes. As the number of cores are increased, the private L1 caches are reduced in size to have larger shared L2 caches. Therefore, while the L1 cache size is scaled down from 64 KB to 16 KB as the number of cores is increased from 2 to 8, the size of the L2 cache is scaled upward between 2 MB and 8 MB based on the number of cores and the number of thread contexts (Tables VI and VIII-B).

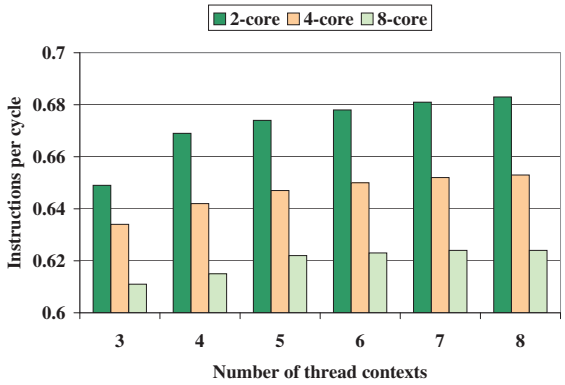
TABLE VII: L2 Cache Size (in MB)/Set Associativity/MSHR Count

Number of Threads	Number of Cores		
	2	4	8
2	2/4/4	3/6/6	4/8/8
3-4	3/6/6	4/8/8	6/12/12
5-8	4/8/8	6/12/12	8/16/16

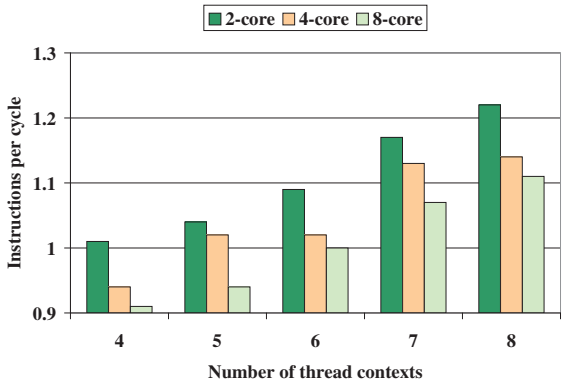
When the number of cores and the number of thread contexts increase, the set associativity for both L1 and L2 caches are increased to reduce the number of conflict misses (Tables VI and VIII-B). The number of Miss Status Handling Registers (MSHRs) in the L1 caches is also increased with the number of thread contexts to allow at most one outstanding L1 cache miss per thread (Table VI). The MSHR count for the L2 cache is dependent on both the number of cores and the number of thread contexts (Table VIII-B). During the simulations, the caches are warmed up for the first 100,000 cycles.



(a) Average IPC per core for CGMT approach



(b) Average IPC per core for FGMT approach



(c) Average IPC per core for SMT approach

Fig. 13: Average instructions per cycle (IPC) count per core for the different multithreading approached

C. Results

The instructions per cycle (IPC) counts for the workloads on the different multi-core processor configurations are plotted in Figures 13a, 13b, and 13c. The IPC for CGMT cores ranges between 0.41 and 0.52, while for FGMT cores the IPC is between 0.61 and 0.68. This marked difference is primarily due to the fact that the FGMT approach is very effective in hiding stalls due to both long latency events (for e.g., cache misses) and short latency events (branch resolution, data dependency resolution, etc.). However, CGMT switches threads to hide stalls only due to long latency events. Moreover, the thread switch penalty in CGMT cores may be more than one cycle, whereas, in FGMT cores, this penalty is exactly one cycle as long as there are ready threads available. The IPC counts for the SMT cores are in the range of 0.91 and 1.22 because the pipeline width for the SMT cores is double of that of the CGMT and FGMT cores. For the same number of threads, the

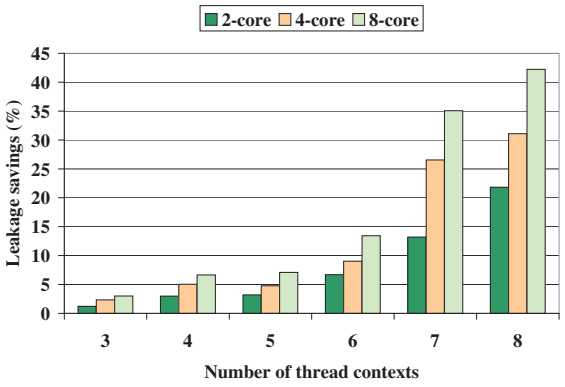


Fig. 14: Average IRF leakage energy savings for CGMT cores

IPC reduces as the number of cores are increased because the L1 cache sizes become smaller. It can also be observed that while the IPC counts for the CGMT and FGMT cores tend to saturate as the number of thread contexts is increased to eight, the IPC for SMT cores increase more linearly indicating that it could support more threads before its performance levels out. It should be noted that the IPC drops when the number of cores are increased because the L1 cache size reduces (Table VI). However, to make sure that the drop in performance is not too dramatic, the size of the shared L2-cache is increased as the number of cores is increased. It should also be noted that the IPC values plotted in Figures 13a, 13b, and 13c are per core. So, for instance, the average IPC for the 8-core processor with each core executing 8 threads is $0.62 \times 8 = 4.96$, whereas the same for a 2-core processor is $0.68 \times 2 = 1.36$.

The leakage savings in the integer register files in CGMT cores is shown in Figure 14. As expected, the savings in the CGMT processors scale very well with the number of thread contexts. For 2 thread contexts, the savings are in the range of 0.9% to 2.9%, while, for 8 thread contexts, the savings are between 22% and 42%. This is because, in a CGMT approach, only a single thread context is active at a time in the entire pipeline till it experiences a long latency stall. Therefore, the register files for the rest of the thread contexts, irrespective of whether they are ready or stalled, may be put to sleep.

In contrast to this, in the FGMT and SMT approaches, instructions belonging to different thread contexts are processed by different pipeline stages. Therefore, the savings do not scale with the number of thread contexts but instead are proportional only to the fraction of the time that is spent by the threads waiting on memory stalls. For FGMT cores, the leakage savings range from 0.8% to 2.02% for 3 thread contexts and 3.09% - 7.8% for eight thread contexts (Figure 15a). The total latencies of L1 D-cache read misses, L1 I-cache read misses, and L2 read misses (normalized over the total number of CPU cycles) averaged over the number of threads are plotted in Figures 15b, 15c, and 15d. For 4 TCs, as the number of cores is varied between 2, 4, and 8, the L2 cache size is set as 3 MB, 4 MB, and 6 MB, respectively. However, for 5 TCs, the sizes are 4 MB, 6 MB, and 8 MB, respectively. Therefore, the capacity miss rate in case of 5 TCs is lower than that for 4 TCs. Further, the set-associativity and the MSHR count for the L2 cache are also increased when the number of hardware thread contexts increases from 4 to 5. Due to this, the conflict misses reduce and the total outstanding cache misses that the processor can support increase as the number of TCs increase from 4 to 5. This the reason why the leakage savings for 4 TCs is more than those for 5 TCs. However, as the number of TCs varies from 5 to 8, the leakage savings increase monotonically.

For SMT cores, the leakage savings range from 1.02% to 2.23% for 4 thread contexts and 2.97% - 7.27% for eight thread contexts (Figure 16). The degradation in performance due to the proposed technique in SMT cores was calculated by counting the number of cycles when an instruction could not be processed by a pipeline stage because the register file was not awake. For SMT cores, the degradation was 0.023% in case of a 8-core processor with each core executing 8 threads.

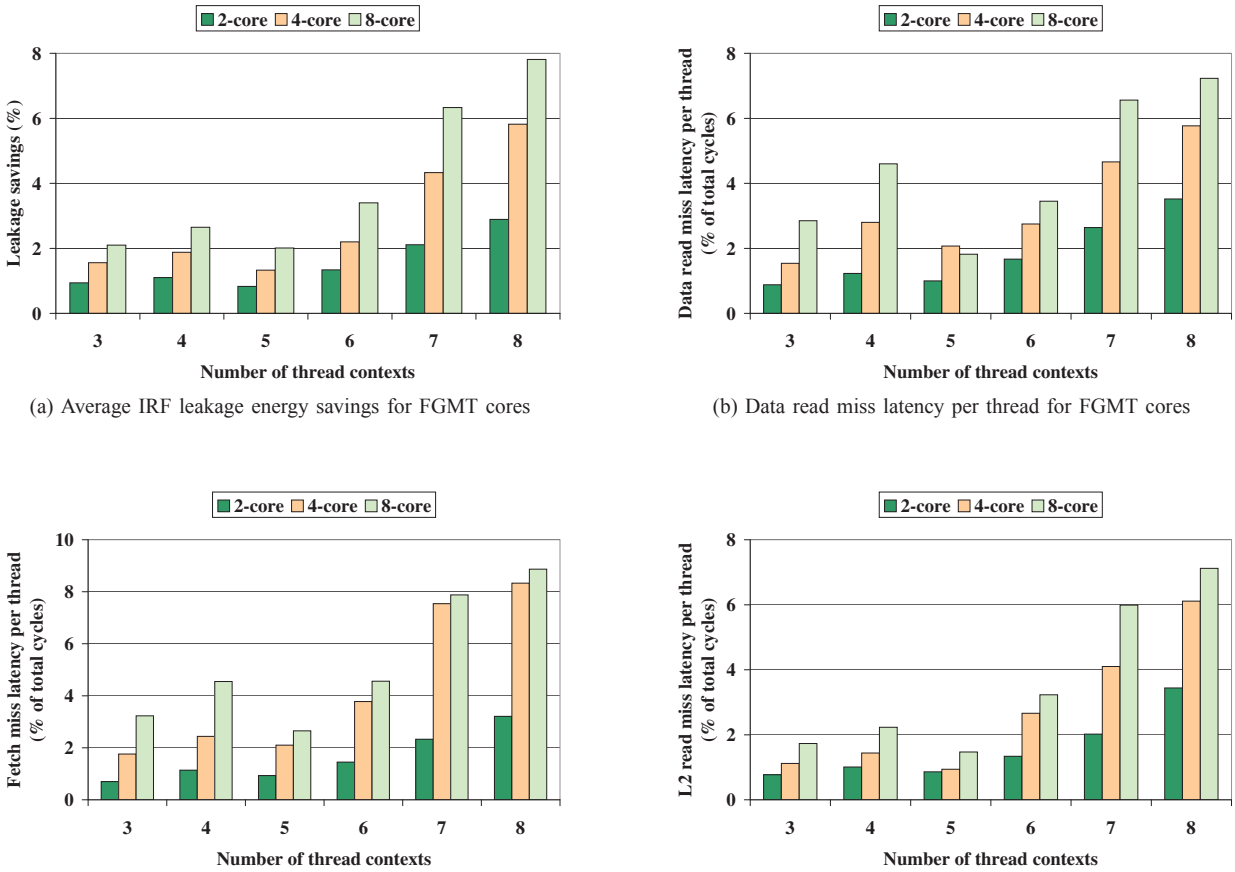
IX. Acknowledgments

We would like to thank the entire M5 development team, especially Korey Sewell, and all the contributors and users of M5 who provided valuable help with respect to using M5. We are also thankful to the anonymous reviewers of the original version of the manuscript whose valuable comments helped improve the quality of this paper. We also acknowledge the services provided by Research Computing, University of South Florida.

X. Conclusions and Future Work

In this work, we synchronize the sleep of a register file private to a thread with the unavailability of that thread and the wake-up with the readiness of that thread to run. This is because the integer register file is accessed very frequently by integer applications. In the future, we would like to extend this work to in-order cores with floating point register files. Floating point applications use both integer and floating point register files and, therefore, the patterns of accesses to these units may provide more opportunities of power gating the register files. distributed between the two register files. For instance, when a thread gets ready to execute after its stall gets resolved, it may need to access only one of the register files before the other. This means that the latter register file has more time to wake-up than the former. Further, in floating point applications, there are regions that are primarily integer computation-intensive. When the application is executing in such regions, the floating-point register file and even floating point hardware units may be put to sleep for much longer periods. Therefore, the fundamental approach that is at the core of the techniques proposed in this paper will result in conservative savings if directly applied in the case of floating point register files.

In the future, we will also extend this work for parallel applications. Parallel programs use atomic instructions for synchronization purposes that gives ride to additional concerns. Special considerations have to be made for atomic instructions because they induce cache coherence issues. In this work, we use multiprogrammed workloads that have mutually exclusive address spaces. Due to this reason, the techniques proposed in this work do not put register files to



(a) Average IRF leakage energy savings for FGMT cores

(b) Data read miss latency per thread for FGMT cores

(c) Instruction fetch miss latency per thread for FGMT cores

(d) L2 read miss latency per thread for FGMT cores

Fig. 15: IRF leakage savings and cache miss latencies for the processors featuring FGMT CPUs

sleep when a cache write miss happens (following a memory store instruction). However, in case of atomic instructions, there will be opportunities to power gate register files during cache write misses as well. Also, using a partitioned register file for each thread could have further advantages. Instead of waking up the entire register file at the same time, only the partitions that needs to be accessed can be woken up more urgently compared to the rest of the partitions.

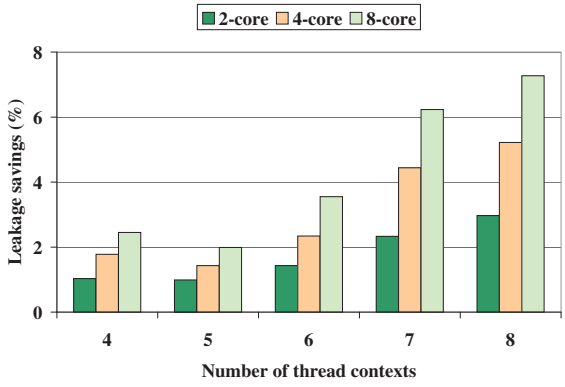


Fig. 16: Average IRF leakage energy savings for SMT cores

REFERENCES

[1] S. Borkar. Design Challenges of Technology Scaling. *Proc. IEEE MICRO*, 19:23–29, 1999.

[2] L. Seiler et al. Larrabee: A Many-Core X86 Architecture for Visual Computing. *ACM Trans. Graph.*, 27(3):1–15, 2008.

[3] R.K. Krishnamurthy et al. High-Performance and Low-Voltage Challenges for Sub-45nm Microprocessor Circuits. *Proc. Intl. Conf. on ASIC*, pages 283–286, 2005.

[4] A. Sodan et al. Parallelism via Multithreaded and Multicore CPUs. *IEEE Computer*, 43(3):24–32, 2010.

[5] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-Way Multithreaded Sparc Processor. *Proc. IEEE MICRO*, 25(2):21–29, 2005.

[6] P. Kongetira, K. Aingaran, and K. Olukotun. Implementation of an 8-core, 64-thread, Power-Efficient SPARC Server on a Chip. *IEEE JSSC*, 43(1):6–20, 2008.

[7] MIPS. MIPS32 1004KTM CPU Family Software Users Manual. *http://www.mips.com*, 2009.

[8] T. Ungerer, B. Robič, and J Šilc. A Survey of Processors with Explicit Multithreading. *ACM Computing Surveys*, 35(1):29–63, 2003.

[9] J. P. Shen and M. H. Lipasti. *Modern Processor Design: Fundamentals of Superscalar Processors, 1st ed.* McGraw-Hill Science/Engineering/Math, 2004.

[10] K. Roy. Leakage Power Reduction in Low-Voltage CMOS Design. *Proc. IEEE ICECS*, pages 167–173, 1998.

[11] Z. Hu et al. Microarchitectural Techniques for Power Gating of Execution Units. *Proc. ISLPED*, pages 32–37, 2004.

[12] H. Singh et al. Enhanced Leakage Reduction Techniques using Intermediate Strength Power Gating. *IEEE Trans. VLSI Sys.*, 15(11):1215–1224, 2007.

[13] S. Kaxiras, Z. Hu, and M. Martonosi. Cache Decay: Exploiting Generational Behavior to Reduce Cache Leakage Power. *Proc. ACM/IEEE ISCA*, pages 240–251, 2001.

[14] K. Flautner et al. Drowsy Caches: Simple Techniques for Reducing Leakage Power. *Proc. ACM/IEEE ISCA*, pages 148–157, 2002.
[15] S. Rele et al. Optimizing Static Power Dissipation by Functional Units Superscalar processors. *Proc. Intl. Conf. on Compiler Construction*, pages 261–274, 2002.
[16] W. Zhang et al. Compiler Suppport for Reducing Leakage Energy Consumption. *Proc. DATE*, pages 1146–1147, 2003.
[17] Y. You, C. Lee, and J.K. Lee. Compiler Analysis and Supports for Leakage Power Reduction on Microprocessors. *ACM TODAES*, pages 147–164, 2006.
[18] N. Seki et al. A Fine-Grain Dynamic Sleep Control Scheme in MIPS R3000. In *Proc. IEEE ICCD*, pages 612–617, 2008.
[19] S. Roy, N. Ranganathan, and S. Katkoori. A Framework for Power Gating Functional Units in Embedded Microprocessors. *IEEE Trans. VLSI Sys.*, 17:1640–1649, 2009.
[20] H. Homayoun, K.F. Li, and S. Rafatirad. Functional Units Power Gating in SMT Processors. In *Proc. IEEE PACRIM*, pages 125 –128, 2005.
[21] A. Youssef et al. On the Power Management of Simultaneous Multi-threading Processors. *IEEE Trans. VLSI Sys.*, PP(99):1 –1, 2009.
[22] S. Rusu et al. Power Reduction Techniques for an 8-core Xeon Processor. In *Proc. IEEE ESSCIRC*, pages 340 –343, 2009.
[23] R. Kumar and G. Hinton. A Family of 45nm IA Processors. In *Proc. IEEE ISSCC*, pages 58 –59, 2009.
[24] T. Saito et al. Design of Superscalar Processor with Multi-Bank Register File. In *IEEE ISCAS*, pages 3507 – 3510, 2005.
[25] A. Agarwal, R. Kaushik, and R.K. Krishnamurthy. A Leakage-Tolerant Low-Leakage Register File with Conditional Sleep Transistor. In *Proc. IEEE Intl. SOC Conf.*, pages 241 – 244, 2004.
[26] J. Lingling et al. Reduce Register Files Leakage Through Discharging Cells. In *Proc. IEEE ICCD*, pages 114 –119, 2006.
[27] H. O. Kim et al. Supply Switching with Ground Collapse for Low-Leakage Register Files in 65-nm CMOS. *IEEE Trans. VLSI Sys.*, 18(3):505 –509, 2010.
[28] A. S. Leon et al. A power-efficient high-throughput 32-thread sparc processor. *IEEE JSSC*, 42(1):7–16, 2007.
[29] Sun Microsystems. OpenSPARC T1 Processor Megacell Specification. *http://www.sun.com*, 2006.
[30] S. Li et al. Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proc. IEEE MICRO*, pages 469–480, 12-16 2009.
[31] H. Q. Le et al. Ibm power6 microarchitecture. *IBM J. Res. Dev.*, 51(6):639–662, 2007.
[32] Sun Microsystems. OpenSPARC T1 Microarchitecture Specification. *http://www.sun.com*, 2006.
[33] J.E. Stine et. al. FreePDK v2.0: Transitioning VLSI Education Towards Nanometer Variation-Aware Designs. In *Proc. IEEE Intl. Microelectronic Sys. Education*, pages 100 –103, 2009.
[34] Nangate. Nangate 45nm Open Cell Library. *www.nangate.com/openlibrary*, 2008.
[35] N. L. Binkert et al. The M5 Simulator: Modeling Networked Systems. *IEEE MICRO*, 26(4):52–60, 2006.
[36] A. J. KleinOsowski and D. J. Lilja. MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research. *IEEE Comput. Archit. Lett.*, 1(1):7, 2002.
[37] M. Jahre and L. Natvig. Performance effects of a cache miss handling architecture in a multi-core processor. *Norwegian Informatikkonferanse (Nik) "http://www.nik.no"*, 2007.
[38] J. Burns and J. L. Gaudiot. SMT Layout Overhead and Scalability. *IEEE Trans. Parallel Distrib. Sys.*, 13(2):142–155, 2002.

Soumyaroop Roy (S'07) received B.E. in Electronics and Communication Engineering in 2001 from Birla Institute of Technology, Ranchi, India and M.S. in Computer Engineering and Ph.D. in Computer Science and Engineering in 2006 and 2010, respectively, from University of South Florida, Tampa, FL. He is a Senior Design Engineer in the Architecture Performance Modeling team at AMD, Austin, TX. His research interests are in architecture and compiler methodologies for low-power design of microprocessors, architecture level performance and power modeling, and low-power VLSI design. From 2001 to 2004, he was a Software Engineer with the NCVHDL group at Cadence Design Systems, Noida, India.

Nagarajan "Ranga" Ranganathan (S81-M88-SM92-F02) received the BE (Honors) degree in electrical and electronics engineering from the National Institute of Technology, Tiruchirapalli, University of Madras, India, 1983, and the PhD degree in computer science from the University of Central Florida, Orlando, in 1988. He is a distinguished university professor of computer science and engineering at the University of South Florida, Tampa. His research interests include VLSI circuit and system design, VLSI design automation, multimetric optimization in hardware and software systems, biomedical information processing, computer architecture, and parallel computing. He has developed many special purpose VLSI circuits and systems for computer vision, image and video processing, pattern recognition, data compression, and signal processing applications. He has coauthored more than 250 papers in refereed journals and conferences, four book chapters, and co-owns six US patents and three pending. He has edited three books titled VLSI Algorithms and Architectures: Fundamentals and VLSI Algorithms and Architectures: Advanced Concepts, IEEE CS Press, 1993, VLSI for Pattern Recognition and Artificial Intelligence, World Scientific Publishers, 1995, and coauthored a book titled Low Power High Level Synthesis for Nanoscale CMOS Circuits, Springer, June 2008. He was elected as a fellow of the IEEE in 2002 for his contributions to algorithms and architectures for VLSI systems. He is a member of the IEEE Computer Society, the IEEE Circuits and Systems Society, and the VLSI Society of India. He has served on the editorial boards for several IEEE and ACM journals. He served on the steering committee of the IEEE Transactions on VLSI Systems during 1999-2003 and 2007-10 and as the editor-in-chief for two consecutive terms during 2003-2007. He received the Distinguished University Professor honorific title and the university gold medallion honor in 2007. He was a corecipient of the Best Paper Awards at the International Conference on VLSI Design in 1995, 2004, and 2006, as well as the IEEE Circuits and Systems Society Transactions on VLSI Systems Best Paper Award in 2009.

Srinivas Katkoori received his doctoral degree in computer engineering from the University of Cincinnati, Cincinnati, Ohio, in 1998. In Fall of 1997, he joined the Department of Computer Science and Engineering at the University of South Florida as an Assistant Professor. In 2004, he was tenured and promoted as an Associate Professor. His research interests are in the general area of VLSI CAD Algorithms and Design Methodologies. Specific research areas include: High level synthesis, Low power synthesis, FPGA Synthesis, and Radiation Tolerant CAD for FPGAs. Dr. Katkoori is a recipient of 2001 National Science Foundation (NSF) CAREER award. Besides NSF, his research sponsors include Honeywell, NASA JPL, and Florida I4 High Tech Corridor Initiative. He is the recipient of the inaugural 2002-03 University of South Florida Outstanding Faculty Research Achievement Award. He is the recipient of 2005 Outstanding Engineering Educator Award from the IEEE Florida Council (Region 3). He serves on technical committees of several VLSI conferences and as a peer reviewer for several VLSI journals. Since 2006, he is serving as an Associate Editor of IEEE Trans. on VLSI Systems. To date, he has published over 50 peer-reviewed journal and conference papers. He holds one US Patent (6,963,217). Dr. Katkoori is a senior member of ACM and IEEE.