Variable Latency Caches for Nanoscale Processor

Serkan Ozdemir Arindam Mallik Ja Chun Ku
Gokhan Memik Yehea Ismail
Department of Electrical Engineering and Computer Science
Northwestern University
Evanston, IL 60208
{soz463, arindam, jck273, memik, ismail} @ eecs.northwestern.edu

ABSTRACT

Variability is one of the important issues in nanoscale processors. Due to increasing importance of interconnect structures in submicron technologies, the physical location and phenomena such as coupling have an increasing impact on the latency of operations. Therefore, traditional view of rigid access latencies to components will result in suboptimal architectures. In this paper, we devise a cache architecture with variable access latency. Particularly, we a) develop a non-uniform access level 1 data-cache, b) study the impact of coupling and physical location on level 1 data cache access latencies, and c) develop and study an architecture where the variable latency cache can be accessed while the rest of the pipeline remains synchronous. To find the access latency with different input address transitions and environmental conditions, we first build a SPICE model at a 45nm technology for a cache similar to that of the level 1 data cache of the Intel Prescott architecture. Motivated by the large difference between the worst and best case latencies and the shape of the distribution curve, we change the cache architecture to allow variable latency accesses. Since the latency of the cache is not known at the time of instruction scheduling, we also modify the functional units with the addition of special queues that will temporarily store the dependent instructions and allow the data to be forwarded from the cache to the functional units correctly. Simulations based on SPEC2000 benchmarks show that our variable access latency cache structure can reduce the execution time by as much as 19.4% and 10.7% on average compared to a conventional cache architecture.

1. INTRODUCTION

Sub-wavelength lithography used for aggressive technology scaling is causing increased variability in process technology parameters. Irrespective of its source or manifestation, variability poses a major challenge in designing high performance processors or complex systems [4]. Techniques that deal with reducing variability have been proposed at lower levels of abstraction, i.e., at the circuit level [18]. These techniques mostly aim at minimizing the variance and tend to have large overheads. Previous research has largely focused on Process Variations that happen due to imperfections in the fabrication technology, but ignored the effects of coupling capacitance and physical location on latency. In this paper, we propose an architecture-level technique to minimize the impact of operation latency variation aiming to fill in this gap. In the heart of our technique lies a pseudo-asynchronous cache architecture that can provide data with varying latencies. Particularly, we analyze the variability in memory access latencies due to coupling effects in the cache lines and data location and propose a self-timed, pseudo-asynchronous cache architecture that can utilize such variations to gain performance.

Each generation of technology scaling reduces the minimum transistor dimensions by roughly 30%, approximately doubling the transistor density [24]. It has been discovered that the importance of wire delay is increasing with decreasing dimensions. Additionally, as technologies scale, designers tend to pack more and more modules onto a chip, which results in a rapid growth of wire lengths [11]. The increase in wire delay with each technology generation has an impact on large memory-oriented elements. Memory (SRAM, DRAM) is organized as an array of bit cells. Very long wires connect all the bit cells together, resulting in heavy capacitive loads. Therefore, much of the access time for memory is spent in charging the word and bit lines that run horizontally and vertically across the array. This impact can be reduced through dividing the array into sub-arrays, but at some point, we need to select the bit cell and propagate the value from the bit cell to the edge of the array, which can still be a costly operation. As a result, elements such as caches and register files cannot be accessed in one cycle. An alternative commonly used is to pipeline the cache or register file accesses and allow an access to take multiple cycles to complete. However, this approach also has limitations. It is predicted that every stage inserted into the critical path of the pipeline reduces the performance by approximately 5% [35]. As a result, level 1 cache sizes are limited. On the other hand, the increasing gap between the performance of the processor cores and the access times to memory, commonly referred to as the memory wall problem, is a primary bottleneck for increasing performance of computing systems. Designers are compelled to develop new techniques to close this gap. Therefore, there is strong motivation to implement larger level 1 caches. By using the pseudo-asynchronous architecture proposed in this paper, the penalty of employing larger level 1 caches will be reduced. In addition, the impact of the wire delays on the latency of cache operations will also be minimized.

Growing wire delays have a negative effect on designing large on-chip caches. For a cache, access latencies can vary significantly. This observation can be attributed to two reasons. First, physical location of the data affects the cache access

SC07 November 10-16, 2007, Reno, Nevada, USA

latency. Data residing in the part of a cache close to the port could be accessed much faster than data that reside physically farther from the port [15]. Second, coupling capacitance is a dominant factor in interconnect latencies. For a set of long interconnect wires in parallel, where coupling capacitance dominates the total load capacitance, the wire delay can vary by several times as a function of the switching activities of neighboring wires [9]. The bulk of the access time will involve routing to and from the banks, not the bank accesses. Hence, coupling becomes an important factor in determining the access latency. Since the latencies are going to vary up to several times, a single, discrete hit latency becomes inadequate. We must note that a number of circuit-level optimizations have been proposed that minimize the latency variation of interconnect wires [10, 21, 34]. However, these techniques are usually costly. In addition, they eliminate the advantages of coupling as well. For example, two neighboring wires that switch in the same direction are shown to have lower latency and lower energy consumption [34]. Circuit-level optimizations eliminate such advantages. Therefore, we argue that architectural techniques that minimize the negative impact of variation while taking advantage of faster execution times is a better alternative than simply trying to eliminate it.

A recent work proposed an adaptive, non-uniform cache architecture (NUCA) [15] to manage large on-chip caches. By exploiting the variation in access time across subarrays, NUCA allows fast access to close subarrays while retaining slow access to far subarrays. Intel Itanium 2 9000 processor on the other hand utilizes a self-timed asynchronous L3 cache [32]. The NUCA and the self-timed L3 cache deals with lower cache levels. Therefore, the latency of the accesses can be adjusted without any impact on the rest of the processor. Our work, on the other hand, deals with level 1 caches. Therefore, our decisions impact the remainder of the datapath.

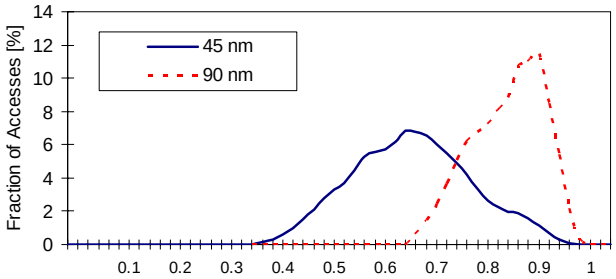

Figure 1. Variation of load access latencies for different

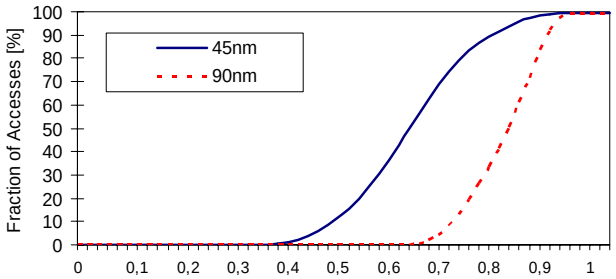

Figure 2. Cumulative distribution of access latencies for different manufacturing technologies.

Since we are dealing with level 1 data caches, we have first analyzed the access latency variation for a representative cache architecture. Figure 1 shows the SPICE simulation results for a 32 KB, 4-way associative cache with different input address transitions, hence accesses to different locations within the cache, and different coupling factors in the cache lines. Figure 2 shows the same results using a cumulative distributive function. We have plotted the distribution of cache access latencies in both 45 nm and 90 nm technologies. For the 90 nm technology, majority of the cache accesses take at least 70% of the maximum latency. However, for the 45 nm technology, a very small percentage (roughly 2.3%) of all cache accesses take longer than 90% of the maximum cache latency and almost 80% of the accesses are resolved in less than 75% of the maximum cache latency. Note that, these fractions are based on all possible switching combinations without any consideration on representative switching patterns. In Section 4, we present the latency distribution of accesses for a variety of applications and show that the fraction of the accesses that require full latency is even lower, 7.4% on average. These results show that there is a scope for improvement at the level 1 cache by utilizing this variation. Our work aims at this objective. First, we show that smaller manufacturing technologies would cause variability in the load access latencies. Based on such a cache model, we propose a variable load latency architecture that exploits this phenomenon. Particularly, our contributions in this work are:

- Based on our SPICE simulations, we develop a model that considers physical phenomena (e.g., coupling capacitance, temperature) and physical location to estimate the access latency of various cache accesses, and
- We implement a cache architecture with variable access latency and show that this can be achieved with small overhead while increasing the overall processor performance by 10.7% and the data access latency by 18.7% on average.

The rest of the paper is organized as follows. In Section 2, we describe our cache design and the corresponding latency model. In Section 3, we give an overview of the processor architecture. The experimental environment is explained in Section 4 and Section 5 presents the results. In Section 6, we give an overview of related work. Section 7 concludes the paper with a summary.

2. CACHE ARCHITECTURE AND CIRCUIT MODELING

This section describes the key components of our cache architecture and circuit models to understand their latency behavior.

2.1 Need for a Pipelined Cache Architecture

Technology scaling has enabled us to put great number of transistors in a single chip. However, this comes with the added overhead of longer wire delays [2]. Furthermore, in smaller technologies, coupling capacitance dominates the total interconnect capacitance and affects the cache latency causing large variations as evidenced by our experiments presented in Section 1. Uniform access latency fails to represent this variability. In addition, popular performance boosting techniques like wave-pipelining [33] will not be efficient in such technologies. To assure correct execution in wave-pipelining, it must be guaranteed that signals from two consecutive accesses will not cross each other. As we have shown in the previous

section, to preserve this guarantee, the cycle time of the cache should be set extremely high, which will cause significant performance degradation. Therefore, there is a strong need to implement pipelined cache architectures. Hence, we implement our adaptive cache architecture based on a pipelined design.

Pipelining reduces the cycle time of a block, which may otherwise be governed by the critical delay of the block. The key hurdle in pipelining the cache into more stages is the bitline delay, which cannot be pipelined because the signals on the bitlines are weak, and not digital; latching can be done only after the sense amplifiers convert the bitline signals from analog to digital. A bitline is loaded by both the multiple memory cells' capacitances and the bitline's wire capacitance and resistance. Consequently, the bitline delay depends on the bank size.

The cache access delay can be divided into five parts: Address bus Delay ($Delay_{AD}$), Decoding Delay ($Delay_{DD}$), Wordline Delay ($Delay_{WD}$), Bitline to Sense Amplifier Delay ($Delay_{BSD}$), and Mux to Data Out Delay ($Delay_{MDD}$) as shown in Figure 3. The locations of pipeline registers are determined by the relative delays of these stages, which depend on cache parameters. Another important characteristic that determines these latencies is subbanking, which is a primary hit time reduction technique in caches. In subbanking, the memory is partitioned into M smaller banks. An extra address word called bank address selects one of the M banks to be read or written. This technique reduces the wordline and bitline (capacitance) and in turn reduces the $Delay_{WD}$ and $Delay_{BSD}$. The array can be split with either vertical cut lines (creating more, but shorter wordlines), or with horizontal cut lines (creating shorter bitlines). However, reduced hit time by increasing these parameters comes with extra area, energy, and/or delay overhead. Increasing horizontal cut increases the number of sense amplifiers, while increasing vertical cuts translates into more wordline drivers and bigger decoder due to increase in the number of wordlines. Most importantly, a multiplexer is required to select the data from the appropriate bank. Horizontal cut increases the size of the multiplexer, which in turn increases the critical hit time delay and energy consumption. As a result, partitioning the cache into multiple banks decreases $Delay_{BSD}$, but increases $Delay_{MDD}$. Beyond a certain point, further partitioning increases the multiplexer delay, and it dominates the decrease in bitline delay. In an optimally banked cache, $Delay_{MDD}$ delay catches up with $Delay_{WD}$ + $Delay_{BSD}$ delay. Therefore, placing a latch in between divides the wordline to data out delay into two approximately comparable portions. This technique has the advantage of having more banks (low bitline delay) than the conventional design. The increase in multiplexer delay due to aggressive banking is hidden in a pipeline stage. The clock cycle of the cache is governed by the wordline to sense amplifier delay ($Delay_{WD}$+$Delay_{BSD}$) and can be made smaller by aggressive banking. The technique can be used to aggressively bank the cache to get the least possible $Delay_{WD}$+$Delay_{BSD}$, which is limited by $Delay_{WD}$. In addition to these two stages, our cache architecture has two more stages for the address bus to the decoder ($Delay_{AD}$) and the decoder itself ($Delay_{DD}$). With the increasing memory size and aggressive banking, $Delay_{AD}$ also increases, and it is a significant fraction of the total delay in our sub-banked cache architecture. Hence, similar to the data cache in the Intel Pentium Prescott [3], our cache design has 4 pipeline stages.

We put latches to store the result of each of the cache pipeline stages after every clock cycle. Sense amplifiers are duplicated for each of the horizontal cuts. A latch is placed at both input and output paths of every sense amplifier to pipeline both read and write delays. Note that, only a single set of latches dissipates energy since switching occurs only on the datalines of the accessed bank. Thus we have incurred area overhead for the latches, but power overhead is negligible. As shown by Agarwal et al. [1], a pipelined implementation can improve the bandwidth of the cache by 60% compared to the conventional cache. The area overhead for their implementation is about 30% for a 64 KB, 4-way associative cache.

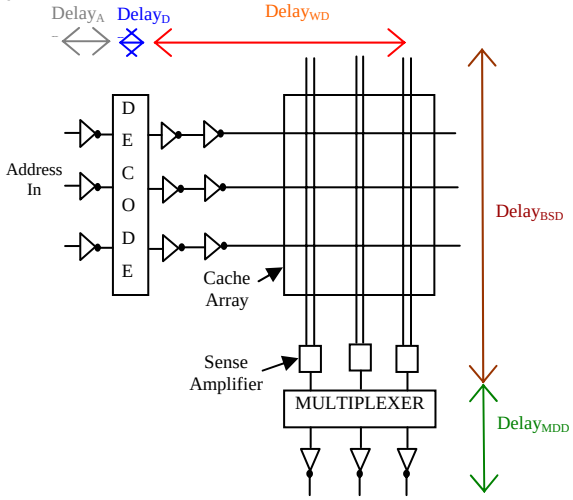

Figure 3. Different components of a cache access delay.

The vertical cuts increase the number of wordlines. This requires a bigger row decoder and more latches. However, at most only two lines switch in a single read/write operation: one that was decoded in the last cycle and one that is being selected in the current cycle. Hence, the power overhead is negligible.

2.2 Circuit Design Issues for Variable Latency Cache

2.2.1 Asynchronous Pipelined Cache

Our variable latency cache consists of four pipelined stages as described in Section 2.1. The first stage is the address bus to the decoder. The second stage is the decoder, and the third stage consists of wordlines, bitlines, and sense amplifiers. Finally, the last stage has the multiplexers (column decoders) and the output driver. The clock frequency is determined such that a data is guaranteed to pass through one stage in a clock period in the worst case. These four stages form the basis of our self-timed cache architecture. As we will explain in the following, accesses will traverse through these stages one-by-one similar to a synchronous pipelined cache. However, we implement handshaking mechanisms between the stages that will allow an access to complete two stages in a single cycle if there is no access in the next stage. In fact, based on the architecture, it is possible that three consecutive stages are merged into two cycles. Overall, as long as the proceeding stages are empty, an access will try to merge those stages and finish the operation as fast as possible. If, for example, the cache is empty at the time an access is initiated, it is possible that the four stages will be merged and

the access will complete in 1, 2, 3, or 4 cycles. However, if there are already intermediate load operations (i.e., accesses that have not completed, yet), the cache will merge the pipeline stages as long as the signals are not crossing each other. The control of the relative timing of the signals are implemented with the help of the cache pipeline registers and the handshaking signals that use *Done* signals generated at each cache stage.

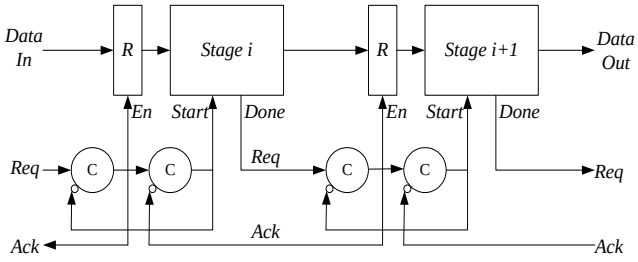

Figure 4. Asynchronous design of pipelined datapath.

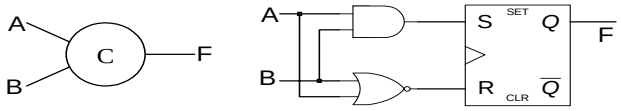

Figure 5. Logic of the Muller C-element used for handshaking.

Figure 4 illustrates the flow of the asynchronous pipelined datapath for our cache. The Muller C-elements whose logic is shown in Figure 5 are used in the handshaking process. We used the four-phase signaling as the handshaking protocol between different stages in which the *Req(uest)* signal goes high followed by the *Ack(nowledge)* signal, then the *Req* signal goes low again followed by the *Ack* signal in a cycle. If the *Req* signal from the previous stage goes high while the *Start* signal of the current stage is low (which indicates the precharge phase), the first C-element is triggered to raise the *En(able)* signal, effectively latching the input data from the previous stage into the register of the current stage. At the same time, the *Ack* signal is sent to the previous stage from the current stage, and the *Start* signal in the previous stage goes low, starting the precharge phase for that stage. As the *Start* signal goes low for precharge, the *Req* signal from the previous stage also goes low. If the *Ack* signal from the next stage is low, the second C-element is triggered as well, which starts the logical operation in the current stage by raising the *Start* signal. The rise in the *Start* signal also makes both the *En* signal of the current stage and the *Ack* signal to the previous stage go low. When the logical operation of the current stage is completed, the *Done* signal is raised, and this becomes the *Req* signal for the next stage repeating the cycle.

2.2.2 Done Signal Generation

In order for the asynchronous design to work without overlapping different data, each pipelined stage requires the *Done* signal generating circuitry except for the first stage since it is guaranteed that a data is able to pass through the first stage within a clock period. The remaining three stages need to be able to generate the *Done* signal whenever the logical operation is completed in the corresponding stage.

The *Done* signal generating circuit in the second stage is placed after the decoder. We employed a dynamic NAND decoder instead of a static one since it has the precharging component built in, which can be used for detecting signal transitions to generate the *Done* signal. Furthermore, since it has a NAND structure, only one signal is going to change during the evaluation phase, thereby making the *Done* signal generation easier and faster. Figure 6 shows the implementation of the circuit with an example with four wordlines. When the *Start* signal is raised in a dynamic NAND decoder, only the selected wordline goes low while all the other lines remain at V_{dd}. When all the wordlines are inverted, and fed into NMOS transistors in a NOR gate implemented in pseudo-NMOS style as shown in Figure 6, only the transition in the selected line will turn one of the NMOS transistors on. This effectively discharges the current from the output node of the NOR gate, and thus the *Done* signal can be generated after inverting the output of the NOR gate.

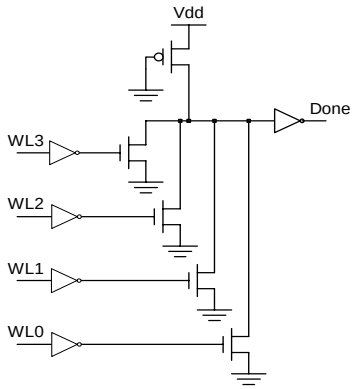

Figure 6. *Done* signal generating circuit after decoder.

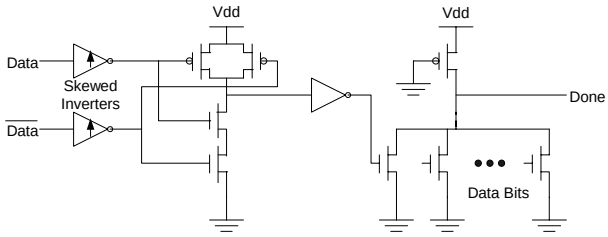

Figure 7. *Done* signal generating circuit after sense amplifiers

Figure 7 illustrates the implementation of the *Done* signal generating circuit in the third stage. This circuit is placed after the changes in the bitlines (and !bitlines) are amplified through the sense amplifiers. Thus, before entering this extra circuitry, the values of the data should be correctly amplified. In other words, the value of either Data or !Data in Figure 7 should reach V_{dd} if the data is being read. These input signals are first inverted using a set of skewed inverters as shown in the figure. Note that these are not typical balanced inverters, instead, the transistor sizes are skewed such that the output of the inverter becomes zero only when the input is very close to V_{dd}. Hence, the output of the skewed inverters will stay at V_{dd} unless either Data or !Data reaches close to V_{dd}. Therefore, there exists a tradeoff between delay and noise-immunity when the transistors are sized. The more the sizing of the pull-up and pull-down transistors is skewed, the closer the input signal has to be to V_{dd} in order to generate zero as the output, which is better in terms of robustness. However, at the same time, the driving force of the pull-down network becomes relatively weaker, thereby increasing the falling

delay of the output. Then, the output signals are fed into a 2-input NAND gate as inputs, which outputs V_{dd} only if at least one of the inputs is zero. Since the output of the NAND gate takes care of only one bit of the data output, we need to AND all the data bits in order to generate the final *Done* signal. To avoid a huge sizing of the gate, the outputs of the NAND gates are first inverted, then fed into a NOR gate implemented in pseudo-NMOS style as inputs, which is logically the same as an AND gate.

Based on SPICE simulation results, it was found that there is a little variation within the last stage. Since the delay of the last stage is almost constant, we used a critical-path replica to generate the *Done* signal for the last stage, where the *Start* signal is fed into both the real logic and the critical-path replica. The *Done* signal is generated when the critical-path replica is completed.

We implemented the extra circuits needed for the asynchronous cache design, and ran simulations using HSPICE. The simulation results showed that the worst-case delay overhead due to the extra circuitry will require the clock period to increase by 13.2% compared to a conventional (synchronous) case. However, note that within the cache, the asynchronous design proceeds at the average speed of the hardware, whereas the synchronous design proceeds at the worst case speed. Furthermore, based on the 256 x 256 bit memory bank that we laid out, the area overhead due to the extra circuitry for the asynchronous design was found to be 4.6% when compared to a synchronous pipelined cache architecture.

2.3 Deriving a Circuit Level Cache Model using SPICE

The proposed design technique is applicable to future generation processors where smaller manufacturing technologies will be used. To analyze our architecture realistically, we propose a new delay model for the proposed cache architecture.

CACTI [28] has been a popular tool for estimating cache latency. However, CACTI has some drawbacks for deep submicron technologies. First, the models used in CACTI are based on an old technology (0.8um). Although the properties can be scaled to a certain extent, for current technologies that reached sub-100nm scale, the results are less reliable. Furthermore, some important effects that arise in deep submicron regime such as coupling capacitance crosstalk, *IR* drop, and temperature are not considered by CACTI. Thus, a new delay model is required to correctly estimate the cache latencies in nanoscale technologies. Most importantly, CACTI does not provide input dependent (e.g., the location of the block, address, data value, switching activity) latency models. In the following, we describe our model that considers such phenomena.

We built a new SPICE model for a 32 KB cache that is based on the structure used in CACTI using 45nm PTM technology models [5]. Note that although we take this model as our base architecture, our approach can be applied to many cache architectures. In addition, since we are considering the latency of the data array, we believe that our results are representative of a large number of implementations. In our model, each memory bank consists of 256 x 256 bits. We employed a dynamic NAND decoder instead of a static one for the reasons explained in Section 2.2. There are three places in the cache where coupling capacitances were added in our new cache model; address bus, parallel wires in decoder, and bitlines (between bitline and !bitline). These lines are modeled as distributed RC ladders, and coupling capacitances between adjacent lines are added in each section in the ladder. The parasitic values of the interconnect wires are based on the interconnect models from PTM [5]. The gate sizes are then optimized using HSPICE simulations to minimize the overall cache latency.

With aggressive technology scaling, both the wire width and the spacing between wires are decreased, thereby making coupling capacitance between adjacent wires a dominating component of the total interconnect capacitance. The effective value of coupling capacitance depends on the signal transitions. In order to analytically capture the effective coupling capacitance, capacitive decoupling is usually used where the nominal coupling capacitance value is multiplied by a discrete constant called Miller Coupling Factor (MCF). Interconnect total capacitance, Ct is then expressed as

$$C_t = C_g + \sum_{\substack{allcoupled \\ linesj}} MCF_j C_c \qquad (1)$$

where C_g and C_c are ground and coupling capacitance, respectively. For simple models, the value of MCF is usually taken to be 2 for two adjacent lines switching simultaneously in the same direction, and 0 for switching simultaneously in the opposite direction. However, the value of MCF actually also depends on the slew rates of the input signals and relative delay between them. It has been shown that MCF can have its theoretical maximum value of 3.85 and theoretical minimum value of -1.85 [9]. Therefore, as the coupling capacitance becomes a dominant portion of the total interconnect capacitance, the variation in the *RC* delay that depends on the input state and transition also increases significantly.

Besides the inclusion of coupling capacitance, we also varied the temperature and the supply voltage during our SPICE simulations in order to observe the thermal and *IR* drop effects on the cache latency. Temperature adversely affects both gate and wire delay. As temperature rises, both mobility and saturation velocity decreases which usually outweighs the effect of decreasing threshold voltage, and it results in a slight increase in the gate delay. Note however, that the temperature dependence of gate delay is also affected by the value of the supply voltage [22]. Metal resistance also increases linearly with temperature, and this increases the wire delay. Furthermore, as the *IR* drop increases, the decrease in the supply voltage results in a decrease in the drain current of the transistors, increasing the gate delay. Therefore, temperature and *IR* drop both have an impact on the clock cycle. In our cache model, we assume the clock cycle time to be 36% of the maximum cache latency measured in room temperature and without any variation in the supply voltage to allow 20% variation in access latency due to each of these effects. According to the SPICE simulations, this selection guarantees that our circuit model will resolve a cache access in at most 4 cycles with worst case temperature and supply voltage variations. Note that, this variation changes the maximum frequency that the cache can work and has an impact on the latency distribution with respect to the worst-case delay. In other words, the latency curve in Figure 1 (in Section 1) is further skewed with the voltage and

temperature variation and the fraction of the accesses that can be completed in small number of cycles increases.

In spite of our efforts, we were not able to define a closed formed formula to represent the latency of cache accesses for each possible input combination mainly due to the nonlinear circuit components. Therefore, the results of the SPICE simulations indicating the exact latency of operations are stored in a four dimensional table. This table determines the exact latency of the cache operations based on (a) last accessed address in the cache, (b) current accessed address, (c) physical location, and (d) the coupling capacitance dependent on the switching combinations. During the architectural simulations, this table is used to determine the cache access latency. Then, using the clock cycle time found as described previously, we can determine the number of cycles a particular cache access takes.

2.4 Discussion

Throughout this paper, we describe our schemes based on the Intel Prescott core, which has level 1 data cache with one read and one write port. Since the write port is separate, there are individual decoders for store operations. Hence, as long as the addresses of a store and load operation are not identical, the store operations can be performed at full latency without impacting the load operations.

Another interesting property of a self-timed component is that the idea can be applied to most of the structures in the processor datapath, such as the issue queue and the register file. We particularly targeted the level 1 data cache due to its high latency variation, its regular structure, and the relatively high number of pipeline stages occupied by it in the datapath.

3. PROCESSOR ARCHITECTURE FOR VARIABLE LATENCY CACHES

An important issue with the self-timed, variable latency caches is the capability of forwarding the values that are read by the load instruction to the dependent instructions. Without a variable latency load operation, the time that the loaded data is available is known exactly (assuming a load hit). However, with the variable latency cache, the data will be available in a window of cycles. Therefore, we need to augment the corresponding structures in the datapath, namely the scheduler and the functional units, such that when the data is available from the cache, it is sent to the correct destination as fast as possible allowing the dependent operation to proceed immediately.

An overly conservative approach is to schedule the dependent instructions assuming the load will take maximum possible hit latency, but this delays the dependent instructions for loads that actually take shorter than this maximum latency, thereby undoing all the possible gains from variable latency cache. Here, we propose a new forwarding and scheduling mechanism that can actually exploit variable latency accesses. After a load instruction is scheduled, the dependent instructions start executing assuming the load access will take the shortest possible amount of time, which is single cycle in our architecture. To avoid these instructions from reaching the execute stage before the data from cache is available, we also add queues at the inputs of each functional unit. These queues are called load-bypass queues. Each queue has $MAX_{hit\_latency}$-$MIN_{hit\_latency}$ number of entries that will allow the instructions to wait until the data is available. These entries will be used if load does not complete in $MIN_{hit\_latency}$ cycles. For the base case, the load-bypass queues have 3 entries in our study. It is also possible to add these queues into the dispatch units, however, to eliminate any effect on the rest of the processor, we choose to implement them just before the functional units. Once the cache access is complete, the destination register number and the data read from the cache is forwarded and broadcasted to the load-bypass queues; where each entry compares the stored register number (which is the input register for the dependent instruction) with the forwarded value. If the two values are identical, the data (i.e., the output of the load operation) is latched into the queue. Then, in the next cycle, the operation will start execution if the functional unit is empty, i.e., not used by another operation that proceeds it in the queue. Figure 8 shows the hardware for this approach. Note that we omitted the multiplexers at the inputs of the functional unit that selects from different forwarded values for simplicity.

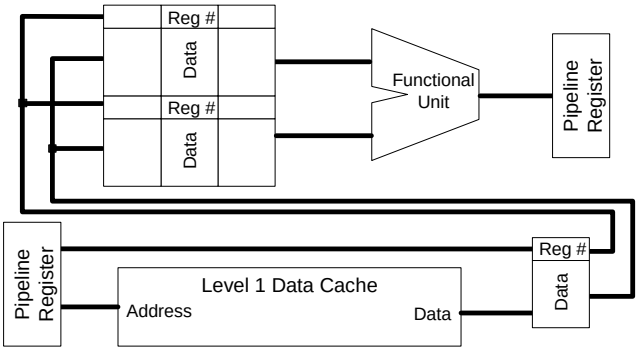

Figure 8. Forwarding the Cache Return Value to the Execute Stage through Load-Bypass Queue.

If the input operands of an instruction are ready, i.e., no forwarding is needed from the cache; it can simply skip the queue and start the execution. However, if one of the input operands will be provided by the cache, then the instruction will enter the queue from the tail. Once an instruction enters the queue, it will move forward one by one every cycle until it reaches the head of the queue or receives the data from the cache. Once it receives the data, it will move to the functional unit and start execution as described above. The start of the dependent instruction will also initiate the dispatching of its dependent operations. Assuming that the functional unit latency is longer than the register file read latency, the pipeline stages will be overlapped (i.e., the dependent instruction will reach the functional unit exactly at the time its input data is available). Therefore, only instructions that are dependent on a load will have to enter the load-bypass queue and the rest of the instructions can directly go to the functional units or receive their inputs from one of the existing forwarding sources. To illustrate this relative timing, consider three instructions: L_1, D_1, D_2, where D_1 is dependent on L_1 and D_2 is dependent on D_1. The cycle after the L_1 finishes, D1 will start its execution. At the same cycle, D_2 will be sent to the RF stage. As soon as D_1 finishes, D_2 will be at the execute stage and will start its execution. If the relative timing in the architecture is different, e.g., the register file stage takes longer than the execution stage, the architecture needs to be augmented with another queue. Particularly, we need to add queues for dependents of dependents. In other words, instead of having a single queue that synchronizes

the cache output with the dependents of the load instructions, we will need to add a second queue (named ALU-bypass queue) of same length that will synchronize the output of the functional units with the instructions that depend on the output of the functional units. Note that this queue should be connected to the outputs of all the functional units. In such a case, each instruction will be marked by the scheduler which queue it should go to. It is also possible for an instruction to be dependent on a load and an ALU operation. In that case, the instruction will have an entry in both queues. Considering the same example, if the timing of the architecture does not allow explicit synchronization, D_2 will be scheduled two cycles after D_1. If by the time it reaches the execute stage, D_1 is not completed, then it will enter the ALU-bypass queue and wait until D_1 is completed. Once D_1 completes, its result will be broadcasted to the ALU-bypass queue (only to the ALU-bypass queue, the broadcasts on the load-bypass queue will be initiated by the cache only). Then, D_2 will store the results and start execution in the next cycle assuming that the functional unit is empty.

If the instruction reaches the head of the queue, and still has not received its input, it means that the load access missed in the cache. Hence, the data will never be forwarded to the queues. Therefore, the dependent instruction needs to be flushed and re-executed based on the replay mechanism that is employed in the processor. Note that, the complexity of replay and the miss penalty of the load operations are not affected by our variable latency architecture.

We also want to point out that the load-bypass queue depicted in Figure 8 resembles the issue queue of an out-of-order processor. Particularly, the operation of our architecture resembles a distributed issue queue used similar to that of Alpha 21264 [14]. However, we instantiate a single small queue for each function unit. Since this “mini issue queue” is designed for the specific cache structure of the datapath, it is much more efficient than a regular issue queue. In addition, the instructions leave this queue regardless of whether they finish their execution or not.

In our experiments we are not modeling process variation, which is crucial for an on-chip cache implementation due to the large area occupied by this component. We must note that, our self-timed cache architecture will handle the effects of latency variation caused by process variation. If for some reason one of the stages fail to complete in a single processor cycle, the *Done* signal will not be generated for that stage, effectively stalling the following load instructions in the cache thereby allowing adequate time for that particular load to complete. To accommodate this increase, we have to assume a maximum cache latency of 5 cycles. Thereby, any increase in access latencies due to process variation will be effectively handled. In this case, such variations will cause a performance degradation, however, the processor will continue working correctly. If there is no process variation, the addition of this extra cycle will not cause any performance degradation because of our variable access latency architecture. The only cost associated with this addition will be the increase in the size of load-bypass queues before the functional units.

4. SIMULATION ENVIRONMENT

The SimpleScalar 3.0 [29] simulator is used to measure the effects of the proposed techniques. The necessary modifications to the simulator have been implemented to model the variable access load latencies, functional unit queues, selective replay, the busses between caches, and port contention on caches. We have also made changes to SimpleScalar to simulate a realistically sized issue queue and to model the events in the issue queue in detail (instructions are released from the issue queue only after we know there is no misscheduling). We simulate 13 floating-point and 11 integer benchmarks from the SPEC2000 benchmarking suite [30]. We simulate 100 Million instructions after fast-forwarding application-specific number of instructions as proposed by Sherwood et al. [26]. Important characteristics of the applications are explained in Table 1. This table presents the number of simulation cycles for the applications, number of DL1 accesses and its miss rate as well as the distribution of the DL1 latencies. Note that the level 1 data cache latency distribution is derived for each application running on the processor core; hence the percentages for different number of cycles are not necessarily equal to the distribution of cache latencies described Figure 1. To explain the difference, consider a loop that consists of a single load L_x that can retrieve the data in x-cycles; iterating throughout the program. The load accesses the same location repeatedly, effectively having 100% x-cycle accesses rather than the curve given in the motivational results. In addition, if the load accesses consecutive addresses, the probability of a bit-switch in the address bus reduces, which in return decreases the coupling capacitance and hence the latency of the load operation. Therefore, the principle of locality helps in reducing the average access latency of our cache architecture.

The base processor is a 4-way processor with an issue queue of 128 entries and a ROB of 256 entries. The simulated processor has separate level 1 instruction and data caches. Both level 1 caches are 32 KB, 4-way associative with 64-byte block size and 4 cycle latency. Unified level 2 cache is 512 KB, 8-way associative cache with 128-byte line size and 18 cycle latency. The level 3 cache is 8 MB 8-way associative cache with 128-byte line size and 50 cycle latency. The memory access delay is set to 160 cycles. All caches are lockup-free. In all the simulations, we assume 7 cycles between the schedule and execute stages and model a bimodal branch predictor with 4 KB table.

5. EXPERIMENTAL RESULTS

This section presents the simulation results for the variable latency cache architecture. Figure 9 shows the reduction in execution time relative to base case for the proposed cache architecture. Note that the base case architecture has a fixed 4-cycle latency data cache access. The performance is improved by up to 19.4% (for *galgel* application) and 10.7% on average. From Table 1 and Figure 9, one can observe that the main factor affecting the performance improvement is the level 1 data cache miss rate. A high miss rate translates into more accesses to the lower levels. As a result, the average data access latency increases, which in turn, diminishes the advantages obtained by employing our idea in the level 1 data cache (*swim*, *applu* and *art*). Similarly any scheme that reduces the average latency spent on the memory hierarchy improves the effectiveness of our scheme. This correlation can also be observed in Figure 10, which shows the reduction in average data access latencies. As expected, there is a very strong correlation between the reduction in the data access latencies and the performance improvement: the applications that have lower latency reduction tend to have less

Table 1. SPEC 2000 application characteristics: Execution cycles (cycles), number of level 1 data cache accesses (DL1 acc), level 1 data cache miss rate (DL1 miss), Cache model latency distribution (1,2,3 and 4 cycles).

Application	Cycles [M]	DL1 acc [M]	DL1 miss [%]	DL1 Latency Distribution			
				1 cycle [%]	2 cycles [%]	3 cycles [%]	4 cycles [%]
168.wupwise	97.2	56.9	0.9	0.8	35.7	58.2	5.3
171.swim	115.7	41.5	10.0	0.3	26.1	63.0	10.6
172.mgrid	106.8	39.0	4.0	1.0	30.8	59.5	8.7
173.applu	165.9	25.1	7.2	1.7	35.1	56.2	7.0
177.mesa	85.9	37.7	0.3	1.6	29.2	64.0	5.1
178.galgel	100.1	34.8	0.1	0.1	25.8	65.5	8.6
179.art	206.1	30.1	32.9	1.7	36.5	54.4	7.4
183.equake	220.4	33.8	6.0	2.5	35.1	56.8	5.7
187.facerec	116.0	34.5	1.5	1.2	28.5	60.7	9.6
188.ammp	123.5	24.6	4.8	3.1	34.9	55.7	6.3
189.lucas	108.2	37.2	5.3	0.2	35.9	58.5	5.5
200.sixtrack	97.4	39.7	1.3	1.3	33.4	58.4	6.9
301.apsi	107.1	29.8	6.4	2.1	29.3	60.8	7.8
FP avg.	126.9	35.7	6.2	1.3	32.0	59.4	7.3
164.gzip	121.7	36.2	6.9	0.7	46.4	49.0	3.9
175.vpr	154.2	48.4	4.0	0.6	20.6	62.7	16.1
176.gcc	169.5	35.8	1.4	1.7	27.6	60.5	10.3
186.crafty	137.5	33.0	0.9	0.8	29.0	62.6	7.6
197.parser	169.1	39.2	5.4	1.6	30.6	59.5	8.3
252.eon	117.4	33.3	0.1	2.2	38.1	55.4	4.3
253.perlbmk	135.9	39.6	2.2	2.5	39.6	52.9	5.1
254.gap	140.7	35.0	0.2	2.0	27.4	64.5	6.1
255.vortex	83.5	47.6	0.3	1.2	19.2	65.3	14.3
256.bzip2	93.2	31.3	0.2	0.7	45.4	49.1	4.7
300.twolf	166.9	43.9	4.6	1.8	43.0	51.8	3.4
INT avg.	135.4	38.5	2.4	1.4	33.4	57.6	7.6
Arith. Mean	130.8	37.0	4.6	1.4	32.6	58.5	7.4

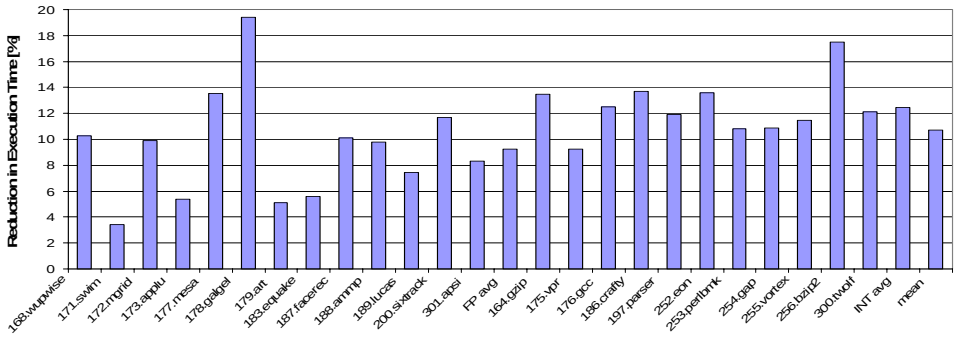

Figure 9. Reduction in Execution Time.

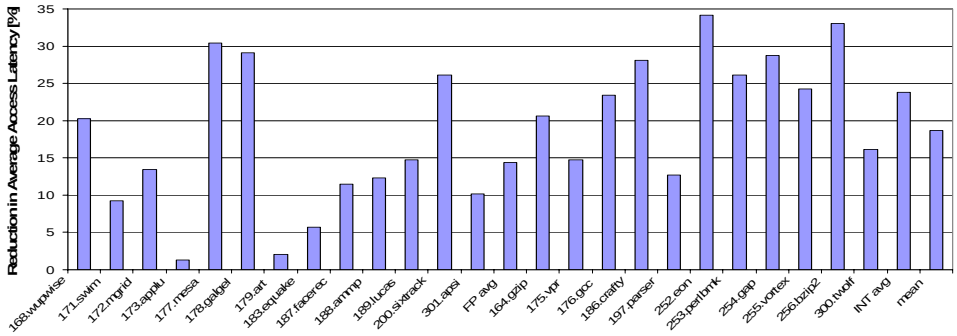

Figure 10. Reduction in Average Access Latency.

performance improvement compared to other applications. It is also important to note that by having a variable latency level 1 data cache we can reduce the average cycles spent in the memory access per load instruction by up to 34.2% and 18.7% on the average.

5.1 Design Alternatives

In this section, we explore different design alternatives to our scheme. More specifically, we measure the execution time for two caches, where one is smaller and the other one is larger than the base cache we have implemented. Starting from the 32KB, 4-way associative cache, which has a 4 cycle access latency, using CACTI we found the largest caches that can be implemented for having 3 and 5 cycles of maximum access latency. We calculated that a 16KB, direct mapped cache can complete accesses in 3 cycles, hence it was the configuration we adopted for the small cache. Similarly we chose a 128 KB, 4-way set associative cache as our large cache configuration, which has a maximum access latency that corresponds to 5 cycles.

The results for these two alternatives are presented in Figure 11 along with our proposed variable access latency cache (32 KB). In addition, we show the performance of a larger variable access latency cache (128 KB). An interesting observation is that regular pipelining for the large cache performs worse than the base case, because of the extra cycle in the cache access despite reduced cache miss rates due to larger cache size. Similarly, the small cache with regular pipelining has very small gain over the base case even though the load execution path is 1-cycle shorter. This clearly shows that ignoring the variability in access latency either hurts our performance or in the very best results in lost opportunity. On the other hand, both of the variable latency schemes result in considerable reduction in execution time. Even the large cache with longer access latency reduces the execution time by 8.0% on average.

Also, note that the behavior of *gzip* application is a good example showing how cache miss ratio affects the performance. Using a 128KB large cache, the dl1 miss rate is reduced from 6.9% to 0.5% reflected as 24.4% improvement in execution time for the 128 KB variable access latency cache. We must note that the SPEC applications generally have low miss rates. Particularly, with the base architecture, the average miss rate is 4.6%. With a more data-intensive application set, the advantages of a larger cache will likely be more profound.

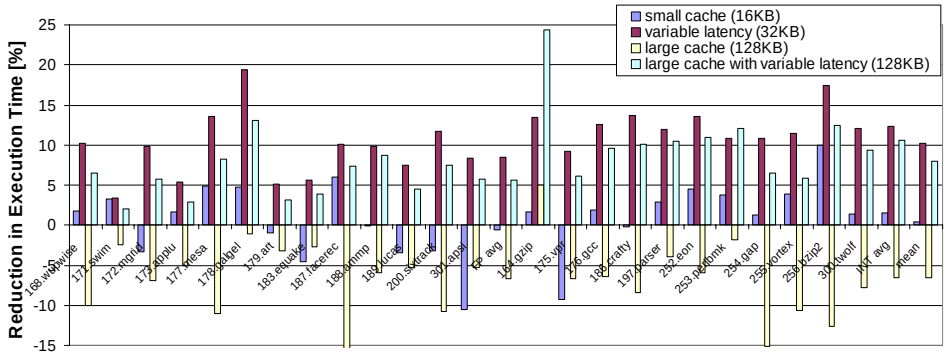

Figure 11. Reduction in execution time for different alternatives.

6. RELATED WORK

Caches supporting non-uniform accesses are not new [23]. Kim et al. proposed NUCA [15] architecture to manage large on-chip caches. By exploiting the variation in access time across subarrays, NUCA allows fast access to close sub-arrays while retaining slow access to far subarrays. NURAPID [6] improves over NUCA by using sequential tag-data access, which causes low power consumption. It is also more fault-tolerant and area-efficient. Variable latency functional units have also been studied by Mueller in the context of parallel machines [20]. A more recent example is the 24-Mbyte L3 cache in the Intel Itanium 2 9000 processor which has a self-timed asynchronous design minimize process variation effects [32]. However, these studies are proposed for lower level caches or for remote machines. In contrast, we utilize the variability in the level 1 cache in our proposal.

To support variation in the load access latency, we use a pipelined cache. Previous research has proposed pipelined cache architectures for high bandwidth applications [1]. However, the pipeline used in that study has been static in nature. Our proposal of varying the number of pipeline stages is novel for the cache structure. Note that, collapsible pipelining has been proposed to reduce clock power in pipelines [12, 16, 27]. In addition, Constructive Timing Violation [8, 25, 31] has been proposed for variable latency pipelines. However, none of these studies have considered the architectural implications of such schemes.

Various researchers have realized the impact of load scheduling and instructions dependent on them on the processor efficiency [13]. Mowry and Luk [19] examine software profiling to predict hit/miss outcome of accesses to dispatch these instructions earlier. Alpha 21264 utilizes a hardware-based hit-miss predictor [14]. Memik et al. [17] proposed a novel precise scheduling technique that determines the accurate access latencies of load operations. However, these studies do not consider variable latency operations.

7. CONCLUSIONS

In this paper, we have proposed a novel self-timed variable access latency cache architecture that optimizes the performance of a processor utilizing the variability in the load access latencies. First, we showed that there will be a large variation in load access latencies in future manufacturing technologies due to effects such

as coupling capacitance. We have provided SPICE simulations to support our claim. Based on this observation, we proposed a pipelined cache architecture that can support variable access latency. Our proposed instruction scheduling and data forwarding scheme improves the processor performance by as much as 19.4% and 10.7% on the average.

8. REFERENCES

[1] A. Agarwal, K. Roy, and T.N. Vijaykumar. Exploring High Bandwidth Pipelined Cache Architecture for Scaled Technology. in Design, Automation and Test in Europe Conf. and Exhibition. 2003. pp. 10778.

[2] V. Agarwal et al. Clock rate vs. IPC: The end of the road for conventional microprocessors. in Intl. Symp. on Computer Architecture. Jun. 2000. pp. 248-259.

[3] D. Besedin, Platform Benchmarking with RightMark Memory Analyzer. 2005,http://www.digit-life.com/articles2/rmma/rmma-p4.html.

[4] S. Borkar. Microarchitecture and Design Challenges for Gigascale Integration. in Intl. Symp. on Microarchitecture. 2004.

[5] Y. Cao et al. New paradigm of predictive MOSFET and interconnect modeling for early circuit design. In Custom Integrated Circuits Conf.. 2000. pp. 201-204. http://www.eas.asu.edu/~ptm.

[6] Z. Chishti, M. Powell, and T.N. Vijaykumar. Distance associativity for high-performance energy-efficient non-uniform cache architectures. in Intl. Symp. on Microarchitecture. Dec. 2003. pp. 55-66.

[7] D. Ernst et al. Razor: Circuit-Level Correction of Timing Errors for Low-Power Operation. in IEEE Micro, November/December, 2004. 24(6): pp. 10-20.

[8] S. Fujii and T. Sato. Non-uniform Set-Associative Caches for Power-Aware Embedded Processors. in Embedded And Ubiquitous Computing. 2004. pp. 217-226.

[9] M. Ghoneima and Y. Ismail. Accurate Decoupling of Capacitively Coupled Buses. in Intl. Symp. on Circuits and System. May 2005.

[10] H. Hidaka et al. Twisted Bitline Architectures for Multi-Megabit DRAM's. in IEEE Journal of Solid-State Circuits. Feb. 1989.

[11] R. Ho, K.W. Mai, and M.A. Horowitz, The Future of Wires. in Proceedings of the IEEE., Apr. 2001.

[12] H.M. Jacobson. Improved Clock-Gating through Transparent Pipelining. in Intl. Symp. on Low Power Electronics and Design. 2004. pp. 26-31.

[13] D.R. Kerns and S.J. Eggers. Balanced Scheduling: Instruction Scheduling When Memory Latency is Uncertain. in ACM SIGPLAN Conf. on Programming Language Design and Implementation. 1993.

[14] R. Kessler, The Alpha 21264 Microprocessor. in IEEE Micro, Mar/Apr 1999. 19(2).

[15] C. Kim, D. Burger, and S.W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. in Intl. Conf. on Architectural Support for Programming Languages and Operating Systems. Oct. 2002. pp. 211–222.

[16] J. Koppanalil et al. A Case for Dynamic Pipeline Scaling. in CASES. 2002.

[17] G. Memik, G. Reinman, and W.H. Mangione-Smith, Precise Instruction Scheduling. in Journal of Instruction-Level Parallelism, Jan. 2005.

[18] S. Mitra et al. Robust System Design from Unreliable Components. in Intl. Symp. on Computer Architecture. 2005.

[19] T.C. Mowry and C.K. Luk. Predicting Data Cache Misses in Non-Numeric Applications Through Correlation Profiling. in Intl. Symp. on Microarchitecture. Dec. 1997. pp. 314-320.

[20] S.M. Müller. On the Scheduling of Variable Latency Functional Units. in Proceedings of the 11th ACM Symp. on Parallel Algorithms and Architectures. 1999.

[21] K. Nose and T. Sakurai. Two Schemes to Reduce Interconnect Delays in Bi-Directional and Uni-Directional Buses. in Symp. on VLSI Circuits. 2001. pp. 193-194.

[22] C. Park, Reversal of Temperature Dependence of Integrated Circuits Operating at Very Low Voltages. in IEEE Electron Devices Meeting. 1995. pp. 71-74.

[23] H. Pilo et al. An 833MHz 1.5w 18Mb CMOS SRAM with 1.67Gb/s/pin. in Intl. Solid-State Circuits Conf.. Feb. 2000. pp. 266-267.

[24] S. Rusu et al. Trends and Challenges in VLSI Technology Scaling towards 100nm. in Asia and South Pacific Design Automation Conf.. 2002. pp. 16-17.

[25] T. Sato and I. Arita, Combining Variable Latency Pipeline with Instruction Reuse for Execution Latency Reduction. in Systems and Computers in Japan, 2003. 34(12).

[26] T. Sherwood, E. Perelman, and B. Calder. Basic Block Distribution Analysis to Find Periodic Behavior and Simulation Points in Applications. in Intl. Conf. on Parallel Architectures and Compilation Techniques. Sep. 2001.

[27] H. Shimada, Ando, H., Shimada, T. Pipeline Stage Unification: A Low-Energy Consumption Technique for Future Mobile Processors. in Intl. Symp. on Low Power Electronics and Design. Aug. 2003. pp. 326–329.

[28] P. Shivakumar and N. Jouppi, CACTI 3.0: an integrated cache timing, power and area model, in WRL Research Report. 2003.

[29] SimpleScalar, The SimpleScalar Tool Set, Simplescalar Home Page. 2001,http://www.simplescalar.com.

[30] Standard Performance Evaluation Council, Spec CPU2000: Performance Evaluation in the New Millennium, Version 1.1. Dec. 2000.

[31] A. Tanino, T. Sato, and I. Arita. An Evaluation of Constructive Timing Violation via CSLA Design. in Cool Chips 2003.

[32] O. Unsal et al. Impact of Parameter Variations on Circuits and Microarchitecture. in IEEE Micro, Nov/Dec 2006. 26(6).

[33] D.C. Wong, G.D. Micheli, and M.J. Flynn, Designing high-performance digital circuits using wave pipelining:algorithms and practical experiences. in IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems. Jan 1993. 12(1): pp. 25-46.

[34] J.S. Yim and C.-M. Kyung. Reducing cross-coupling among interconnect wires in deep-submicron datapath design. in Design Automation Conf.. 1999. pp. 485-490.

[35] V. Zyuban et al. Integrated Analysis of Power and Performance for Pipelined Microprocessors. in IEEE Trans. on Computers, August, 2004. 53(8): pp. 1004-1016.