

Reordering Memory Bus Transactions for Reduced Power Consumption

Bruce R. Childers, Tarun Nakra
Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260
{childers, nakra}@cs.pitt.edu

Abstract

Low energy consumption is becoming the primary design consideration for battery-operated and portable embedded systems, such as personal digital assistants, digital still and movie cameras, digital music playback, medical devices, etc. For a typical processor-based system, the energy consumption of the processor-memory component is split roughly 40-60% between the processor and memory. In this paper, we study the impact of reordering memory bus traffic on reducing bus switching activity and power consumption. To conduct this study, we developed a software tool, called MPOWER, that lets an embedded system designer collect a trace of memory bus accesses and determine the switching activity of the trace, given design parameters such as data and address bus width, bus multiplexing, cache size, block size, etc. Using MPOWER, we measured the effectiveness of reordering memory accesses on switching activity. We found that for small caches, which are typical of embedded processors, the number of signal transitions in an ideal case can be reduced by an average of 53%. This paper also describes a practical hardware scheme for reordering the elements within a cache line to reduce switching activity. We found that cache line reordering reduces switching activity by 15–31%.

1. Introduction

In the past decade, the demand for small, battery operated embedded systems, such as cellular phones, digital motion and still cameras, personal digital assistants (PDAs), medical devices, etc., has seen explosive growth. For example, in 1997, 163 million cellular phones were shipped worldwide with market growth expected to reach 356 million units in 2000 [21].

Not only has the sheer demand for battery-operated devices grown, but the expectations of those devices has also increased dramatically. As the capabilities of handheld portable systems continue to increase, there is an ever greater need for more computing power with extended battery life. Indeed, in the cellular phone and laptop computer markets, processing capabilities and battery life are two of the most important features that manufacturers use to differentiate their products.

Although aggressive processors such as Hewlett-Packard's PA-8500 [12] and Compaq's Alpha 21264 [13] have the processing capabilities demanded by high-performance embedded applications, they are not suitable for small portable devices due to high power consumption and heat dissipation (for example, the Alpha 21264 dissipates 72 watts of power [11]). To solve this problem, new processors are being developed which have high performance, while offering low power consumption [8,22].

This paper studies the impact of reordering memory bus transactions on the switching activity and power consumption of the memory system. We developed a tool set, called MPOWER, for evaluating the impact of memory bus design on power consumption. MPOWER lets an embedded system designer vary the underlying bus protocol and structure to determine its effect on the power/energy budget. In this paper, we use MPOWER to answer the question of whether reordering memory bus transactions is worthwhile for embedded processors, and in what situations to apply it. We also briefly outline in this paper a hardware scheme for reordering a cache line to reduce switching activity based on the encouraging results from our initial experiments.

The rest of this paper is organized as follows. Section 2 presents background material on power consumption, and Section 3 describes the goal of bus transaction reordering. Section 4 presents a detailed performance analysis of the effect of different memory bus structures and cache sizes on the ability of bus reordering to reduce switching activity. Section 5 briefly sketches an application of reordering to the data elements of a cache line and how reordering within a cache line can reduce switching activity. Section 6 describes related work, and Section 7 concludes the paper.

2. Power Consumption

The amount of power consumed by a portable system is a useful metric for determining the maximum current that a battery

must supply. It is also useful for determining the packaging features needed to handle an embedded processor’s expected heat dissipation. Along with power consumption, a device’s energy should be considered because it measures power consumption over time and determines battery life: $energy = power \times time$

We need to be careful that architectural features that reduce power consumption do not increase execution latency so much that there is no net decrease in energy consumption. In CMOS circuits, the power consumption due to dynamic switching (or so called “bit flips”) dominates the power lost to static leakage [5]. The average power of a CMOS device, called P_{avg} , can be expressed in terms of its switching activity:

$$P_{avg} = \alpha_T \cdot f_{CLK} \cdot C_{load} \cdot V_{DD}^2$$

where α_T is the switching activity factor (i.e., the average number of signal transitions per clock cycle), f_{CLK} is the clock frequency, C_{load} is the capacitance load, and V_{DD} is the supply voltage. Assuming that f_{CLK} , C_{load} , and V_{DD} are held constant, switching activity determines the power consumption of a device.

In the case of off-chip resources, the capacitance load is very large relative to on-chip resources, and high C_{load} leads to high power consumption (i.e., the drivers for external pins are very large and consume much power when driving long wires with high capacitance). Thus, reducing the average number of signal transitions when driving external pins reduces overall energy consumption, assuming execution latency is not increased. For the memory bus, which accounts for 15-30% of the memory hierarchy’s energy consumption [10], reordering bus transactions to minimize switching activity may significantly reduce the energy consumption of the memory hierarchy. This paper focuses on reducing α_T for the external memory bus. Improving α_T corresponds to reducing the number of signal transitions that occur over a program’s execution lifetime, and for this paper, we use the reduction in switching activity as our metric to indicate reduced power consumption.

3. Memory Bus Reordering

Because off-chip memory accesses are very expensive power-wise, we are studying how much reordering bus transactions (to minimize signal transitions) reduces overall energy consumption.

One strategy for reducing the number of bit flips on the memory bus is to schedule bus transactions (we consider a bus transaction to be a read or write of a single word) in the order in which they would cause the minimal signal changes. Of course, the dependences between bus transactions must be taken into account. For example, a load that follows a store in program execution order can not be scheduled before the store (assuming they use the same address). Likewise, an instruction that causes a data cache line replacement must be scheduled before the transactions that replace the line.

To investigate the effect of reordering bus transactions on energy consumption, we incorporated a *transaction scheduler* in MPOWER. The scheduler maintains a list of recently seen

off-chip memory access events (e.g., a request to load a cache line), and schedules bus transactions from the list according to which transaction causes the least bit flips given the current state of the bus. The scheduler ensures that all dependences between instructions and memory reads and writes are satisfied by building the dependence graph on-the-fly and only scheduling transactions that are leaf nodes. The transaction scheduler uses *priority list scheduling*, where the priority is the number of bit transitions between successive bus transactions.

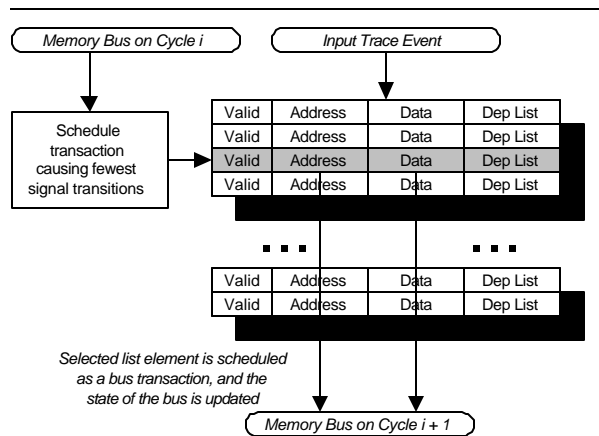


Figure 1: Transaction scheduler

Figure 1 illustrates the operation of the scheduler. The scheduler accepts an input event from a memory access trace, and places it in the scheduling list. If the new event causes the list to reach a threshold size, an event is removed from the list and scheduled as a bus transaction. Only those events that have no dependences with other events in the list may be scheduled as an output transaction. Of the ready events, the one that causes the least number of signal transitions is chosen.

The scheduler is locally greedy and may not find an optimal order for the whole trace. However, with large thresholds, the scheduler gives a good approximation that indicates the worth of reordering transactions. Furthermore, we do not use the scheduler as a hardware device to reorder memory bus transactions. Instead, we use the scheduler to answer the question of whether reordering is valuable, and if so, in what situations to apply it. We are currently studying hardware mechanisms and compiler support for transaction reordering. Section 5 presents preliminary results in this regard.

4. Experimental Results

To study the effect of memory bus reordering on switching activity, we used several applications from the MediaBench [16] and SPEC’95 benchmark suites. These applications are described in Table 1 (the ones marked with † are from SPEC’95). The benchmarks were chosen to be representative of common applications for communication devices and PDAs.

MPOWER uses a modified version of the SimpleScalar processor simulator from the University of Wisconsin [4]. SimpleScalar was instrumented to collect a trace of memory accesses between the L1 instruction and data caches and off-chip memory.

Benchmark	Description
<i>adpcm</i>	Audio decoder
<i>gsm</i>	Full-rate speech decoder
<i>g721</i>	Voice compression
<i>jpeg</i>	Image compression
<i>perl†</i>	Perl interpreter
<i>li†</i>	Lisp interpreter
<i>go†</i>	Game of GO

Table 1: Benchmark applications

Figure 2 shows the structure of a typical processor-memory interface [8]. Our simulated architecture has small on-chip separate instruction and data caches, with a narrow bus between the caches and the off-chip DRAM memory controller. In the experiments that follow, we evaluate different bus structures (multiplexing and bus width) and their impact on reordering memory transactions. We first examine the general issues associated with reordering, and based on our experiments, we briefly describe in the second portion of the paper a simple hardware scheme for arranging the elements of a cache line to minimize signal transitions.

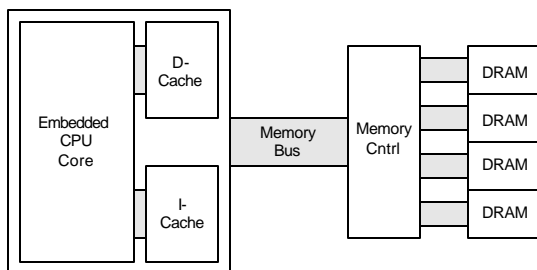


Figure 2: Processor-memory system structure

In the initial experiments, we assume that the external memory bus has no bidirectional signals; i.e., there are separate data input and output lines. For unidirectional buses, a signal is always driven and will only experience a change in state when a different bit value is written. We use a unidirectional bus for our initial experiments because it gives the most scheduling freedom and lets us best answer the question of how well reordering transactions reduces switching activity. The experiments in Section 5 use a bus that has bidirectional data lines with tristate drivers, as commonly found in embedded processors such as Motorola’s PowerPC 7400 [20]. In this case, the bus is not driven following a burst transfer (i.e., a cache line), and the scheduling scope is limited to a cache line.

Although there is likely to be a latency penalty for reordering bus transactions, in this paper, we make the assumption that the order of transactions does not affect latency. We are

currently implementing a form of transaction reordering and investigating its impact on memory latency. In the initial experiments, we assume that the memory system does not do burst transfers, and an individual word in a cache line is transferred in a single transaction. This assumption is relaxed in Section 5 where we consider a bus that supports burst transfers.

The experiments in the first part of this paper use a trace of the first 1.5 million off-chip memory accesses. In Section 5, the programs are run to completion.

4.1. Data Width and Multiplexing

The processor-memory bus for an embedded system often has a narrow external interface to reduce cost. For instance, the external memory interface of the Motorola M•CORE MMC2001 embedded processor has a non-multiplexed bus with a data width of 16 bits (two transfers are done to read/write a native word of 32 bits) and an address width of 19 bits [19]. In this section, we examine the effect of bus width and multiplexing on reordering bus transactions and its associated switching activity.

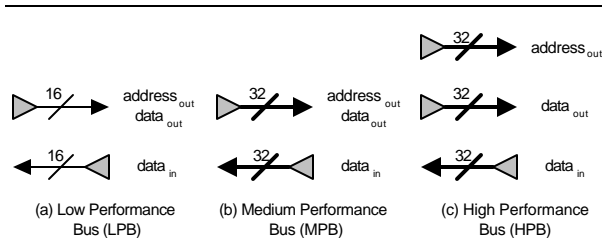


Figure 3: Bus structures

Using MPOWER, we evaluated the impact of the different bus structures that are shown in Figure 3. In the figure, the bus structures have unidirectional connections to external memory (i.e., data input and output). The low performance bus (LPB) has a bus width of 16 bits that is multiplexed for 32-bit addresses and data. The medium performance bus (MPB) has a bus width of 32 bits that is multiplexed for addresses and data. Finally, the high performance bus (HPB) has separate address and data buses that are 32 bits wide, giving a total bus width of 96 bits.

Figure 4a shows the assumptions made about bus timing. In the case of the LPB, the addresses are placed on the bus starting with the high order address ($a_{31:16}$), followed by the low order address ($a_{15:0}$). Data also appears in the same order for a read or write. The MPB places the full address on the bus prior to reading or writing data. Finally, the HPB places the full address and data on the bus simultaneously for writes. For HPB reads, the data read from memory is available on the next cycle from the data bus.

Figure 4b shows how bit flip cost is computed for each of the three bus structures. In the cost equations, pc is population count (i.e., the number of 1’s in a word) and \wedge is exclusive-or. The cost computation takes into account the unidirectional data buses and the multiplexing of the address and data bus for

Cycle	LPB		MPB		HPB	
	Read	Write	Read	Write	Read	Write
0	$a_{31:16}$	$a_{31:16}$	$a_{31:0}$	$a_{31:0}$	$a_{31:0}$	$a_{31:0}, d_{31:0}$
1	$a_{15:0}$	$a_{15:0}$	delay	$d_{31:0}$	$d_{31:0}$	
2	delay	$d_{31:16}$	$d_{31:0}$			
3	$d_{31:16}$	$d_{15:0}$				
4	$d_{15:0}$					

(a) Bus timing & transaction ordering

		Cost Computation
LPB	Write	$pc(da_{out} \wedge a_{31:16}) + pc(a_{31:16} \wedge a_{15:0}) + pc(a_{15:0} \wedge d_{31:16}) + pc(d_{31:16} \wedge d_{15:0})$
	Read	$pc(da_{out} \wedge a_{31:16}) + pc(a_{31:16} \wedge a_{15:0}) + pc(d_{in} \wedge d_{31:16}) + pc(d_{31:16} \wedge d_{15:0})$
MPB	Write	$pc(da_{out} \wedge a_{31:0}) + pc(a_{31:0} \wedge d_{31:0})$
	Read	$pc(da_{out} \wedge a_{31:0}) + pc(d_{in} \wedge d_{31:0})$
HPB	Write	$pc(a_{out} \wedge a_{31:0}) + pc(d_{out} \wedge d_{31:0})$
	Read	$pc(a_{out} \wedge a_{31:0}) + pc(d_{in} \wedge d_{31:0})$

(b) Bit flip cost calculation

Figure 4: Bus timing and bit flip cost calculation (pc is population count and \wedge is exclusive-or).

Benchmark	2K			4K			8K		
	LPB	MPB	HPB	LPB	MPB	HPB	LPB	MPB	HPB
adpcm	13.73%	26.36%	36.25%	14.51%	29.64%	43.62%	9.23%	23.41%	48.23%
g721	12.79%	54.62%	55.27%	12.76%	51.96%	52.94%	12.90%	55.00%	55.33%
go	13.73%	40.25%	49.57%	13.01%	39.14%	48.29%	12.33%	40.02%	48.51%
gsm	14.14%	49.08%	52.09%	13.84%	48.20%	50.98%	13.79%	49.68%	52.42%
jpeg	18.60%	45.78%	66.43%	18.92%	43.69%	67.45%	18.56%	42.89%	66.15%
li	16.63%	42.18%	49.17%	15.02%	44.98%	52.76%	13.92%	40.89%	52.22%
perl	17.16%	40.54%	45.62%	16.69%	39.86%	45.45%	14.90%	41.20%	45.89%
Average	15.25%	42.69%	50.63%	14.96%	42.50%	51.64%	13.66%	41.87%	52.68%

Figure 5: Percentage reduction in switching activity with bus transaction reordering.

writes. For the LPB, the bit flip cost of a write is the number of signal transitions that result between successive portions of the address (high and low halfwords) and the data (high and low halfwords). The cost of a read is computed similarly, except the data cost is computed with respect to the data input bus. Bit costs are computed in a similar way for the other bus structures, taking into account bus width and multiplexing.

Figure 5 shows the percentage reduction in switching activity with bus reordering, assuming perfect knowledge about the instruction and data streams (i.e., the scheduler knows the value of words read from memory), and a scheduling window size of 32. The percentages were calculated with respect to a baseline system with the same cache and bus structures, but without transaction reordering. The figure shows three different instruction and data cache sizes from 2K to 8K. Also shown are the LPB, MPB, and HPB structures. In the figure, the reduction in switching activity varies from 12.8–67% depending on cache size and bus structure.

The figure shows two trends. First, as cache size increases, the relative reduction in switching activity decreases for most of the cases. As expected, with larger caches, the miss rates for the instruction and data accesses is lower, and there are fewer off-chip accesses to main memory. Because the numbers in Figure 5 are relative percentages, they do not always show the decreasing trend in bit flip cost. Although the results are not reported here, the absolute number of bit flips always decreases with an increase in cache size for all benchmarks, even for the HPB (which shows an increasing trend in Figure 5).

The second trend shown in Figure 5 is that the high performance bus benefits the most from cache line reordering. The HPB has the largest relative decrease in switching activity because it gives the scheduler the most leeway in choosing transactions. For the LPB, the cost of a transaction is computed over the individual halfwords in the full native word. This has the effect that it causes a large number of bit flips between upper and lower halfwords, which may have very different bit patterns. For example, the instruction format for a branch typically has a small immediate constant encoded in the lower portion of an instruction with an opcode specified in the upper portion. Hence, there may be a large number of bit changes required between reading successive portions of a branch instruction. For the HPB, the upper opcode bits are not as likely to change between successive instructions. Furthermore, the switching activity is dependent on only adjacent words for the HPB, which the scheduler can arrange to minimize bit flips. For the LPB, the scheduler cannot arrange the halfwords of an instruction or data access to minimize signal transitions. This behavior suggests that the scheduling granularity should match bus width. If the scheduler could order the individual halfwords of an LPB access independently, we expect that there would be a greater decrease in switching activity for the LPB.

From Figure 5, we conclude that bus transaction reordering is most effective for embedded processors which have a split address and data bus with small on-chip caches, but it still

helps for other bus structures (LPB and MPB). For the remainder of this paper, we assume an architecture that has a HPB and 8K instruction and data caches.

4.2. Scheduling Window Size

Figure 6 shows the percentage reduction in signal transitions for different scheduling window sizes, assuming perfect knowledge about the memory access trace. The figure demonstrates the effect that scheduling window size has on reducing the number of signal transitions.

A window size of 64 does the best overall, ranging from 50–71% reduction, with an average of 57%. However, from the figure, the reduction quickly levels off, and a window size of 16 effectively captures most of the reduction in switching activity. For a window size of 16, the reduction varies from 41–57% with an average of 47%. The small table size is encouraging since it suggests that a small hardware buffer may capture most of the energy reduction possible through bus transaction reordering.

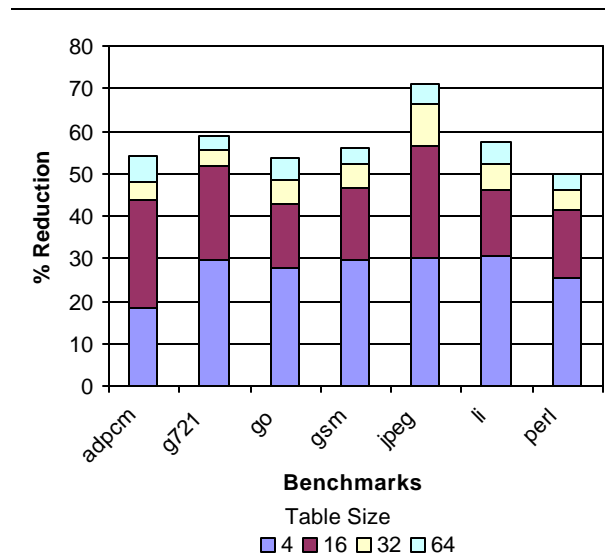


Figure 6: Percentage reduction in signal transitions for different window sizes.

4.3. Instruction vs. Data Reordering

Because the instruction and data streams have very different properties and behavior, we investigated which stream contributes the most toward reducing bit flips; i.e., for which stream is reordering the most effective, and where should we focus our efforts? Figure 7 shows the percentage of total switching activity that can be attributed to instructions and data. The bottom portion of each bar shows the percentage of total bit flips due to data loads or stores, and the top portion shows the percentage of bit flips due to instruction accesses.

In Figure 7, the relative contribution of each stream to switching activity is highly application dependent. For example, *adpcm* and *jpeg* get the most benefit from reordering the

data stream because these benchmarks have very low instruction cache miss rates (less than 0.05%). *Adpcm* and *jpeg* have a hot loop that runs directly out of the L1 I-cache, giving a low instruction cache miss rate. Because the on-chip I-cache nicely captures the kernel loop’s execution, it causes fewer off-chip accesses, and the data stream dominates switching activity (the L1 D-cache miss rate for *adpcm* and *jpeg* is 2.7% and 1.3%). *G721* and *gsm* have different behavior: they are dominated by instruction accesses, which means that reordering the instruction stream is most effective.

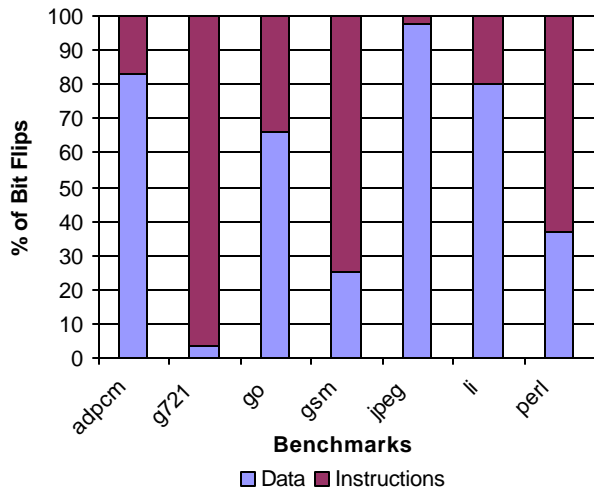


Figure 7: Percentage of the total number of signal transitions according to type (instruction or data).

Not only is reordering dependent on cache miss rates, it is also dependent on data and address density. For an application which has dense data (i.e., minimal bit changes between successive data items), the data stream will likely contribute less to the overall reduction in switching activity. Also, the encoding of the instruction set architecture can have a large impact on switching activity and energy consumption of the memory hierarchy [3,18]. An encoding that ensures the smallest number of bit changes for common code sequences is the most effective at reducing switching activity.

Assuming that we have an efficient hardware scheme to do reordering, then Figure 7 indicates that we should adjust the reordering scheme to match an application’s behavior. For example, *adpcm* does not need data reordering, so we could turn off reordering for the data stream to reduce the energy consumption of the reordering hardware. I.e., the energy saved by reordering bus transactions for the data stream may not be worth the energy spent computing the best order to access memory. Similarly, if we are building an application-specific processor, we may leave out instruction or data reordering depending on the application’s behavior to reduce chip area. Finally, for a system with a mixed workload, including both data and instruction reordering is important. In this case, as suggested above, we will likely want to turn on/off the reordering hardware on a per application basis using dynamic (e.g., counters for cache miss rates) or profile information.

4.4. Reordering with Real Knowledge

The experiments in the previous sections assume perfect knowledge about the instruction and data streams. Having perfect knowledge about the data stream means that the scheduler knows the value of data reads. Perfect knowledge of the instruction stream means that the scheduler knows the value of the loaded instructions as well as the execution paths within the program. The scheduler takes into consideration the value of a data or instruction read when deciding which transaction to schedule. In practice, the values of loaded items would not be available while reordering transactions. Also, since the scheduler is trace driven, knowledge of execution paths is implicit within the traces. However, the previous experiments give an upper bound on how much we can reduce switching activity of the memory bus. Section 5 presents the effects of a cache line reordering technique on switching activity by actually executing the programs.

In this section, we relax the assumption about perfect knowledge, and consider the case where the scheduler only knows execution paths and the value of stores and instructions. Figure 8 shows the effect of our relaxed assumptions as a decrease in switching activity over a baseline that does not reorder bus transactions. The first set of bars shows the decrease in switching activity when the scheduler only has knowledge about execution paths and stored values, and the second set of bars shows the decrease in switching activity when the scheduler has knowledge about execution paths, stored values, and instruction values. The latter case is shown to demonstrate the potential of instruction reordering when the compiler is able to rearrange the instructions.

In Figure 8, the reduction in switching activity for reordering with knowledge about execution paths and stored values varies from 15–57%, with an average of 29%. *G721* and *gsm* have small decreases in switching because the majority of their accesses are associated with instruction or data reads. Nevertheless, even in this case, it is beneficial to reorder only stores, especially since the hardware mechanisms for reordering stores are likely to be very simple (e.g., a store buffer).

The figure also shows the effect of reordering with knowledge about instruction values (with knowledge about stores and execution paths). For instruction reordering, the compiler could arrange instructions in an order that causes the minimal signal transitions. The reordering scope over which the compiler can arrange instructions is limited to sequential code sequences (applying trace-based optimization may help to increase the reordering scope) or to a single cache line. The compiler must convey to the hardware the dependences between instructions that would normally be implicit in sequential code. (Section 5 gives some preliminary results for such a scheme, where the compiler reorders instructions within a cache line.)

As the second set of bars in Figure 8 shows, adding knowledge about instruction values to the transaction scheduler greatly improves the ability of the scheduler to reduce switching activity. In this case, the improvement in switching ranges from 37–57%, with an average of 46%. In the figure, *g721* and

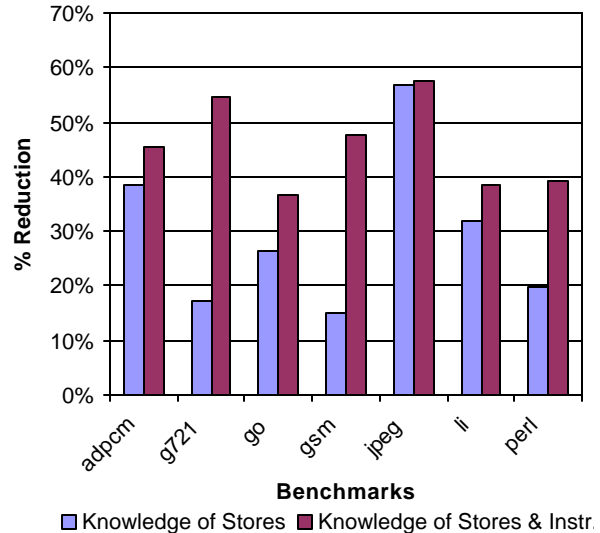


Figure 8: Percentage decrease in bit flips without knowledge about load values.

gsm have a large improvement due to knowledge about instruction values. As mentioned previously, for these benchmarks, the switching activity is most strongly associated with the instruction stream. Reordering instructions for these benchmarks greatly improves the switching activity because they are dominated by instruction accesses.

From Figure 8, we conclude that reordering with knowledge about instruction and stored values can reduce switching activity nearly as well as reordering with complete knowledge about both the instruction and data streams. The reduction shown in Figure 8 compares favorably to the scheme that has full knowledge (see Section 4.1), which reduces switching by an average of 53% (for the HPB and 8K caches).

5. Cache Line Reordering

The previous sections demonstrate that a significant reduction in the switching activity of the memory bus can be obtained by reordering memory bus transactions over a small fixed window of memory accesses. The techniques described above are intended to determine an approximation of the very best that we can achieve given dependences between instruction and data accesses. Based on our encouraging initial results, we are investigating hardware structures to do reordering with compiler support.

Our initial technique, called *cache line reordering* (CLR), considers a small window of accesses. In this scheme, a reordering vector is associated with every cache line that indicates the order in which individual cache line words should be accessed. The memory controller must be able to access memory in an arbitrary order within a cache line and to burst transfer the data to/from the memory system. We reorder accesses within a cache line to find the order that minimizes the transitions for a given line. The CLR scheme has the advantage that it does not have to build a dependence graph dynamically, as the previous technique does.

In this section, we use CLR with a bidirectional bus that has tristate drivers and supports burst transfers. Such a bus is only driven by the CPU or the external memory controller during a cache line transfer. Between transfers, the bus is not driven. Therefore, we do not reorder between cache lines because the switching overhead is effectively always the same when starting a new transfer, regardless of the previous one. However, within a transfer (i.e., cache line), it is possible to reorder elements to minimize switching.

Because a cache line may contain many words (e.g., a 32-byte line has 8 32-bit words), we only reorder within a subline of 4 words. Subline reordering reduces the number of possible combinations of words from $8!$ to $2 \cdot 4!$ for a 32-byte line. Although there is a performance loss (i.e., reduction in bit flips) due to reordering over a smaller set than a full cache line, it is more practical to search $4!$ combinations than $8!$ dynamically in hardware.

CLR is applicable to both data and instructions. The first set of bars in Figure 9 shows the effect of *ideal CLR*. In this case, the hardware always knows the best order in which to read or write every cache line. Unlike the previous experiments, the numbers in the figure were collected using execution-driven simulation that does not have any knowledge about execution paths. The benchmarks in the figure were run to completion, except for *go* and *perl* which were halted after 50 million instructions. The figure shows that CLR can reduce the number of signal transitions by 17–31%, with an average of 25%.

Although ideal CLR is not practical for the data cache (note, however, that we can use profile-feedback optimization to help for data accesses), it is practical for the instruction stream. In this case, the compiler knows at compile time the binary representation of instructions and can reorder instructions within a cache line. To use this optimization, the compiler needs to know the architecture of the memory-processor interface and to convey to the hardware the reordering vectors associated with every cache line.

For the data stream, we can track the best order for each cache line to determine how to transfer it to/from main memory. To do the reordering, the hardware computes the best order for each line whenever it is written back to memory (i.e., a dirty line replacement) or loaded from memory for the first time. The best order for each line is stored in a buffer that is indexed by cache line address. When loading a line, the memory management unit requests the line from main memory in the order specified by the reorder vector in the buffer. The hardware manages the allocation of buffer space.

The second set of bars in Figure 9 (labelled “real CLR”) shows the effect of compiler-driven reordering of the instruction stream and hardware dynamic reordering of the data stream. The figure shows that a practical CLR scheme can reduce the overall number of signal transitions significantly. Reordering with a hardware scheme reduces switching activity nearly as well as cache line ordering with perfect knowledge. In this case, the reduction is 15–31% with an average of 21%.

In typical systems, the memory bus can consume a 15–30% of the memory hierarchy’s energy [10]. In this case, cache line reordering may reduce energy consumption by 2–9%.

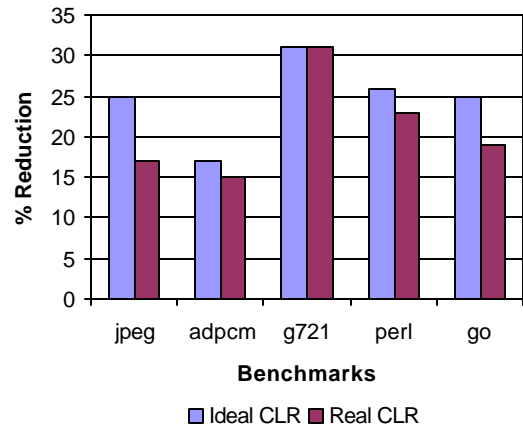


Figure 9: Percentage reduction in switching activity for CLR. For ideal CLR, the processor always knows the best order in which to read or write cache lines. For real CLR, the hardware reorders instruction cache lines with compiler assistance and reorders data cache lines dynamically.

Although the reduction is small, we believe that like compiler optimizations, you must apply all the tricks in your bag to substantially reduce overall energy consumption. Cache line reordering is such a technique that can be applied in concert with other energy reduction strategies. Our initial results are promising, and we are continuing to explore techniques for reducing the energy consumption of the memory hierarchy. A future paper will describe our architecture and implementation of cache line reordering.

6. Related Work

There has been much research on improving power consumption by minimizing processor switching activity. Within a processor, switching has been studied in the context of registers and functional units. Tiwari, Malik, and Wolfe [17,24] perform power-aware instruction selection to reduce the switching effects during execution. Register allocation has also been studied with the aim of reducing switching activity [6].

In order to reduce off-chip activity, most previous work focuses on improving cache structures to reduce accesses to external memory. Cache strategies have been analyzed in the context of power consumption [1]. Cooper and Harvey proposed a compiler-controlled on-chip buffer [7] to hold spilled register values to reduce traffic to main memory. Other techniques such as loop [2], filter [15], and victim caches [14] reduce switching by buffering values between processing elements (e.g., between the CPU and on-chip caches or between the CPU and off-chip memory). Our work differs because it aims to reduce switching cost for a given memory hierarchy and cache design.

In [9], the authors reduce the power consumption of the external bus by reordering complex data structures in memory using a cost function. The previous work that relates most to ours is that of [23]. In their work, the switching of off-chip accesses is reduced by encoding addresses with gray codes.

Using gray codes, consecutive addresses differ only by a single bit. Gray coding was applied to instruction fetches, which are sequential within a basic block. Our work considers both instruction and data accesses to reduce switching. Also the compiler uses prior knowledge of bus activity during instruction fetches to reorder the instructions.

7. Summary and Future Work

This paper studies reordering bus transactions to reduce the switching activity of the external memory interface. We have developed a tool set called MPOWER for studying the impact of bus structure and transaction reordering on power consumption. In the paper, we use MPOWER to investigate the effectiveness of transaction reordering and in what situations to apply it. We found that for an ideal case, switching activity can be reduced by an average of 53% for a 32-bit non-multiplexed bus. This paper also briefly sketches a feasible hardware scheme for bidirectional buses that can reorder the instruction and data streams with compiler assistance. Our hardware scheme reduces switching activity by 15–31%, leading to a possible energy savings of 2–9%. We are continuing to investigate memory bus transaction reordering. In particular, we are:

- extending MPOWER to determine energy consumption for both memory and the processor;
- developing and implementing a hardware scheme for reordering the elements in a cache line to reduce the energy consumption of the memory bus;
- and developing compiler techniques for reordering the instruction stream with hardware support, and investigating the use of profile-feedback optimizations for reordering the data stream (to reduce hardware complexity).

References

- [1] Bahar R. I., Albera G. and Manne S., “Power and performance trade-offs using various Caching Strategies”, *Int’l. Symp. on Low-Power Electronics and Design*, pp. 64–69, Monterey, CA, August 1998.
- [2] Bellas N., Hajj I., and Polychronopoulos, “Architectural and compiler support for energy reduction in the memory hierarchy of high performance microprocessors”, *Int’l. Symp. on Low-Power Electronics and Design*, pp. 70–75, Monterey, CA, August 1998.
- [3] Bunda J., Fussell D., Athas W. C., Jenevein R., “16-bit vs. 32-bit instructions for pipelined microprocessors”, *ACM/IEEE Int’l. Symp. on Computer Architecture*, pp. 237–246, San Diego, CA, May 1993.
- [4] Burger D. and Austin T. M., “The SimpleScalar tool set, version 2.0”, Technical Report #1342, Computer Science Department, University of Wisconsin-Madison, June 1997.
- [5] Chandrakasan A. and Brodersen R., *Low Power Digital CMOS Design*, Kluwer Academic Publishers, 1995.
- [6] Chang J. and Pedram M., “Register allocation and binding for low power”, *32nd ACM/IEEE Design Automation Conference*, pp. 29–35, 1995.
- [7] Cooper K. D. and Harvey T. J., “Compiler-controlled memory”, *Proc. of the 8th Int’l. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*, pp. 2–11, October 1998.
- [8] Cuppu V., Jacob B., Davis B., and Mudge T., “A performance comparison of contemporary DRAM architectures”, *ACM/IEEE Int’l. Symp. on Computer Architecture*, pp. 222–233, Atlanta, GA, May 1–5, 1999.
- [9] Diguett J., Wuytack S., Cathoor F. and Man H. D., “Formalized methodology for data reuse exploration in hierarchical memory mappings”, *Int’l. Symp. on Low-Power Electronics and Design*, pp. 30–35, Monterey, CA, August 1997.
- [10] Fromm R., Perissakis S., Cardwell N., et al., “The energy efficiency of IRAM architectures”, *ACM/IEEE Int’l. Symp. on Computer Architecture*, pp. 327–337, Denver, CO, June 1997.
- [11] Gowan M. K., Biro L. L., and Jackson D. B., “Power considerations in the design of the Alpha 21264 microprocessor”, *Proc. of the 1998 Design and Automation Conference*, pp. 726–731, San Francisco, CA, June 1998.
- [12] Hunt D. and Lesartre G., “PA-8500: The continuing evolution of the PA-8000”, *42nd IEEE Int’l. Computer Conference*, San Jose, CA, February 1997.
- [13] Kessler R. E., “The Alpha 21264 microprocessor”, *IEEE Micro*, Vol. 19, No. 2, pp. 24–36, March/April 1999.
- [14] Jouppi N., “Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers”, *ACM/IEEE Int’l. Symp. on Computer Architecture*, pp. 364–373, May 1990.
- [15] Kin J., Gupta M., and Mangione-Smith W. H., “The filter cache: A energy efficient memory structure”, *IEEE/ACM 30th Int’l. Symp. on Microarchitecture*, pp. 184–193, Research Triangle Park, NC, December 1997
- [16] Lee C., Potkonjak M., and Magione-Smith W. M., “MediaBench: A tool for evaluating and synthesizing multimedia and communication systems”, *IEEE 30th Int’l. Symp. on Microarchitecture*, pp. 330–335, RTP, NC, December 1997.
- [17] Lee M., Tiwari V., Malik S., and Fujita M., “Power analysis and low-power scheduling techniques for embedded DSP software”, *IEEE/ACM Int’l. Symp. on Systems Synthesis*, pp. 110–115, Cannes, France, September 1995.
- [18] Lefurgy C., Piccininni E., and Mudge T., “Evaluation of a high performance code compression method”, *IEEE/ACM 32nd Int’l. Symp. on Microarchitecture*, pp. 93–102, Haifa, Israel, November 1999.
- [19] M-CORE MMC2001 Reference Manual, Motorola Semiconductor Product Sector, Austin, Texas, www.motorola.com/SPS/MCORE, 1998.
- [20] Motorola PowerPC MPC7400 User’s Manual, document number MPC7400UM/D, Motorola Semiconductor Product Sector, Austin, Texas, www.motorola.com/SPS, 2000.

- [21] Quan M., "Chip makers reap benefits of cell-phone boom", *EE Times*, www.eetimes.com, Nov. 18, 1999.
- [22] Santhanam S., "StrongARM SA110: A 160 MHz 32b 0.5W CMOS ARM processor", *HotChips VIII*, pp. 119–130, Stanford, CA, August 1996.
- [23] Su C-L, Tsui C-Y, and Despain A. M., "Low power architecture design and compilation techniques for high-performance processors", *IEEE Computer Conference (COMPCON)*, February 1994.
- [24] Tiwari V., Malik S. and Wolfe A., "Compilation techniques for low energy: An overview", *Int'l. Symp. on Low-Power Electronics*, October 1994