

DSIMD: Dynamically Exploiting DLP in SMT Processors

Patrick Joseph Eibl

Andrew Shell Waterman

Department of Electrical and Computer Engineering
Duke University
{pje2, asw9}@ee.duke.edu

Abstract

With power dissipation being a significant design constraint in modern microprocessors, techniques that can reduce power consumption with minimal degradation in performance are of particular importance. We propose one such technique, Dynamic SIMD (DSIMD), which is aimed at reducing instruction fetch power in simultaneously multithreaded (SMT) processors, which are already considered an area of interest in energy efficiency.

In multithreaded workloads that happen to have independent threads that execute the same code, but not quite simultaneously, DSIMD will stall the leading thread(s) until they run in lockstep with those that lag behind. The benefits of this are twofold: first, instruction fetch only needs to be performed once for all threads in lockstep, with the unused fetch units clock gated to save power; second, contention for the instruction cache is reduced, lowering the penalty cycles due to misses and thereby boosting performance. We present simulation results for DSIMD showing consistently high fetch savings (and therefore fetch power savings) of 24% to 46%, while at the same time improving performance. We also demonstrate a means of implementing DSIMD with minimal additions to existing fetch hardware.

1 Introduction

Simultaneous multithreading (SMT) is a technique whose potential to greatly increase utilization of resources and overall performance has been demonstrated by a significant body of research [1] [2]. SMT achieves this benefit by making it possible to issue instructions from multiple independent threads each cycle, making more instructions available to issue at any time and keeping the processor busy with useful work. The viability of SMT has been proven with its implementation in recent commercial processors [4] [5].

A subset of SMT research has been directed at power efficiency [3]. Power is a paramount issue in modern CPU design both in the mobile market, where battery life is a concern, and in high-performance computing, which pushes the limits of heat-dissipating packaging technology. Because of its ability to both utilize processor resources more fully and to reduce misspeculation, SMT is inherently power efficient.

SMT can be considered among a number of architectural solutions to the problem of maximizing processor performance on workloads with significant data-level parallelism (DLP). Most such solutions use some combination of SIMD instructions (with dedicated vector processing hardware) and multithreading. With multithreading, the workload is split into several independent threads, which will have the best combined performance running on a chip multi-processor (CMP) or an SMT processor. We propose a technique called Dynamic SIMD (DSIMD) that aims to combine the instruction fetch efficiency of SIMD with the high-performance multithreading achieved by SMT. Unlike SIMD ISA extensions, such as SSE [5], DSIMD does not rely on modifications to the exposed architecture or compiler. Instead, it exploits DLP intrinsic to existing binaries.

The inspiration for DSIMD is a characteristic of many of the multithreaded workloads mentioned above: often, multiple threads are executing the same code, and in many situations their control flows are identical for long stretches. Their PCs may be offset by some constant, though, as the threads are likely to be at different points in the code. If we force these threads to run in lockstep by stalling the leading threads, instruction fetch would only need to be performed once for all of the threads in lockstep, since they will be fetching identical instructions until they diverge at a conditional branch, misspeculation, or data cache miss. This leads to an increase in apparent fetch bandwidth for a constant number of fetch units. Therefore, with minimal performance implications, one or more fetch units can be disabled with a clock gating mechanism while threads are in lockstep, resulting in a reduction in power.

In this paper, we investigate the potential of DSIMD to reduce power and energy consumption, and its implications for performance and chip area. Section 2 details our hardware platform and the necessary modifications to implement DSIMD. Section 3 describes our simulation environment and methodology then details the results of our experiments. Section 4 examines some of the interesting implications of DSIMD that become apparent in our results. Section 5 discusses possibilities for extending and improving on the technique we present.

2 Hardware

2.1 Base Hardware Platform

We model a processor similar to the canonical SMT implementation presented in [1]: an out-of-order, superscalar core modified minimally to implement simultaneous multithreading. The core is two-threaded, and the issue width is varied between 2, 4, and 8. The processor contains 16KB, 4-way set-associative L1 data and instruction caches; its unified L2 cache is 1MB.

SuperEScalar (SESC) [8], the cycle-accurate architectural simulator we use to model this processor, does not implement an advanced SMT fetch policy like ICOUNT. Instead, it supports round-robin fetch. Using the nomenclature in [1], we model the fetch schemes RR.2.1, RR.2.2, and RR.2.4 for our 2-issue, 4-issue, and 8-issue configurations, respectively: for issue width k , each of the two thread contexts can fetch up to $k/2$ instructions every cycle. Thread shortage is not an issue, in spite of these configurations, because all simulations are two-threaded. We deliberately choose not to model the better-performing fetch scheme that selects up to k instructions per thread per cycle (e.g. RR.2.8 for 8-issue) because such a configuration is inherently power-inefficient and is thus contrary to our goals.

2.2 Hardware Modifications

We propose the following generic implementation of Dynamic SIMD, generalized for an arbitrary number of thread contexts. Each thread's current PC is inserted, on every cycle, into a searchable FIFO queue unique to that thread. On each cycle, each thread determines if any other thread's PC appears in its queue. If so, there exists an opportunity to line the current thread up with another thread that lags behind it. The current thread is then stalled (its fetch unit is turned off via clock gating) until the trailing thread "catches up", i.e. the trailing thread's PC matches the head of the current thread's queue. At that point, the two threads begin to run in lockstep: the current thread is unstalled, its fetch unit remains disabled, and its instruction source becomes the output of the other thread's fetch unit. This continues until the two threads' PCs become unequal, at which point they resume normal execution.

Some subtleties exist in this design. Deadlock can occur, for example, if the stalled thread holds a lock that a lagging thread is waiting for. Forward progress can easily be guaranteed, however, by setting a maximum stall time. We propose that this stall timeout be proportional to the instruction's position in the queue at the time of stall. Furthermore, it is necessary to prevent all threads from stalling simultaneously in the event of a circular dependence—another form of deadlock.

We present a sample hardware implementation for two thread contexts in Figure 1. Each thread can be in one of three states: normal operation (FetchSrc = 0 and Stall = 0), stalled (FetchSrc = 0 and Stall = 1), or lockstep (FetchSrc = 1 and Stall = 1). During normal operation, a given thread fetches from the instruction cache every cycle, and the multiplexer passes the instructions fetched from the cache port that corresponds to that thread. During a stall, a thread's fetch unit is disabled via clock gating; no instructions are decoded. During lockstep, a thread's fetch unit is disabled, and the multiplexer passes the other thread's instruction stream.

A thread enters the stall state from the normal state if the other thread's PC appears in its queue, the current PCs are unequal, and the other thread is not currently stalled (to prevent deadlock). A thread enters the lockstep state from the stall state if the current PCs are equal and remains in this state as long as this condition is met. (There is no legal transition from the normal state directly to the lockstep state in this implementation, but this optimization is certainly conceivable.) Normal operation resumes if a stall times out or if, during lockstep, the PCs diverge.

In this implementation, the stall timeout is implemented using a saturating down counter whose initial value is the position of the other thread's PC in the queue times some constant. Since this constant is likely to be a small power of two, as we will demonstrate later, no multiplication need be performed. The searchable queue is implemented as a content-addressable shift register, which we will refer to as a FIFO-CAM.

It is apparent that the complexity of this design is quadratic in the number of thread contexts, k . Since the contents of each queue must be checked against the PC of each other thread, each FIFO-CAM must have $k-1$ read ports. The size (and delay) of each FIFO-CAM grows as the number of read ports. An obvious optimization to mitigate the resulting area and power increase is to only store several of the least-significant bits of each PC in the FIFO-CAMs; false-positive lineup opportunities would be rare, since the most-significant bits of the PC do not change often, especially with respect to the queue lengths of interest. The amount of control logic at each thread also increases with k , since it is no longer implicit which thread is catching up to a given stalled thread, but the area and power consumption of the FIFO-CAMs is expected to dominate that of the control logic.

Fortunately, DSIMD is still implementable for more than two threads. As we will later demonstrate, the queues only need to have a handful of entries. More importantly, the only logic added to the critical path is the fetch source multiplexer, whose delay only scales as the logarithm of the number of threads.

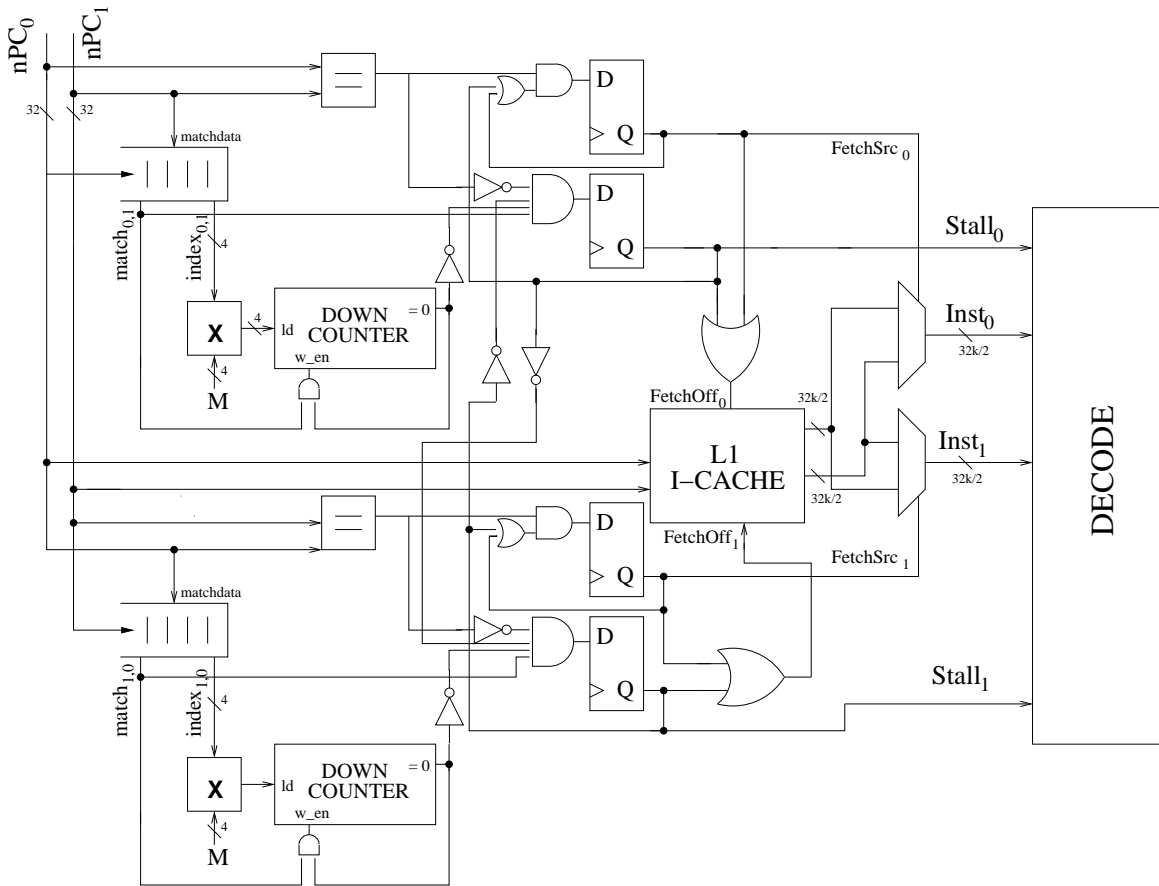


Figure 1: A sample DSIMD hardware implementation

3 Experimental Simulation

3.1 Framework

To explore the performance implications and power savings of DSIMD, we vary the two configuration parameters of our proposed design: the queue size and the stall timeout multiplier. Preliminary simulations indicated that performance fell sharply for queue sizes 32 and above: the probability of two threads’ execution paths lining up decreases, and the penalty of stalling increases. As such, we simulated queue sizes of 4, 8, and 16. For each queue size, we also considered stall timeout multipliers of 1, 2, and 4; the penalty of stalling increases unacceptably for longer multipliers, and most code of interest should retire at least one instruction every four cycles, anyway. Furthermore, in the event of pathological code, the upper-bound on stall penalty (the queue size times the timeout multiplier) must be such that DSIMD does not unacceptably degrade performance.

We simulated each of these DSIMD configurations with two thread contexts on two-issue, four-issue, and eight-issue superscalar cores. Although it is unlikely that power-conscious designs will be eight-wide, such cores

must pay careful attention to heat dissipation, so reducing fetch power is still of relevance.

We execute programs on a modified version of SESC that implements DSIMD as described above. SESC’s SMT fetching policy was also modified to implement the round-robin fetch policies described in the Section 2.1. By default, it implements a greedy round-robin fetch algorithm that does not model realistically implementable behavior.

Simulations were performed on multithreaded programs from the SPLASH-2 benchmark suite [9]. The benchmarks selected were OCEAN, an ocean current simulator; FMM, an N-body interaction simulator; and RADIX, an integer radix sort kernel. We compiled these programs with a gcc cross-compiler built to produce Alpha binaries. The programs were linked against SESC’s thread library but were otherwise unmodified. Initialization routines and the first iteration of each benchmark were discarded from our analysis. The number of dynamic instructions actually considered was over 100 million for each benchmark.

3.2 Results

In this section, we show the effectiveness of DSIMD with regard to three metrics: percentage of fetches saved, speedup, and percentage reduction in energy-delay product, with each of these being a comparison against the

same processor configuration without DSIMD. The number of fetches saved is directly proportional to the reduction in energy, as each saved fetch corresponds to a cycle where the fetch unit is clock-gated. Speedup is a simple method for evaluating the performance implications. The energy-delay product has been proposed as a useful metric for evaluating efficiency [7], as it takes into account both energy savings and their effects on performance. Whenever these metrics are averaged, the geometric mean of their corresponding ratios is used.

In order to calculate energy-delay, we were forced to choose a value for the percentage of processor power that is used in fetching. In the extreme case, fetching power can approach as much as 27% [6]. In the interest of being a bit conservative for our estimate, we use a value of 15%.

In our figures, we use the nomenclature Q_iM_j as shorthand for a DSIMD configuration with queue size i and stall timeout factor (multiplier) j .

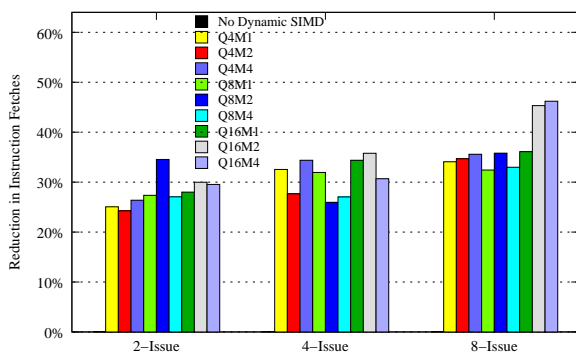


Figure 2: DSIMD fetch bandwidth reduction for various configurations

The results for mean fetch savings across our three benchmarks (Figure 2) show some correlation between issue width and percentage savings. Between 24% and 46% of fetches were saved, with trends being difficult to identify due to individual benchmark differences. For mean speedup (Figure 3), we find that there is not a large deviation from the configurations without DSIMD (between 0.98 and 1.07). Interestingly, we see much larger deviations on 2-issue than on 4- or 8-issue.

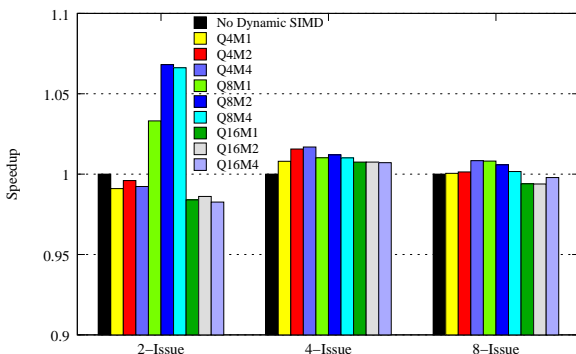


Figure 3: Performance impact of DSIMD for various configurations

The mean energy-delay product reduction (Figure 4) is found to be positive in all cases, ranging from about

2% to 12%. As with speedup, it is rather consistent for 4- and 8-issue (6%), but varies greatly for 2-issue.

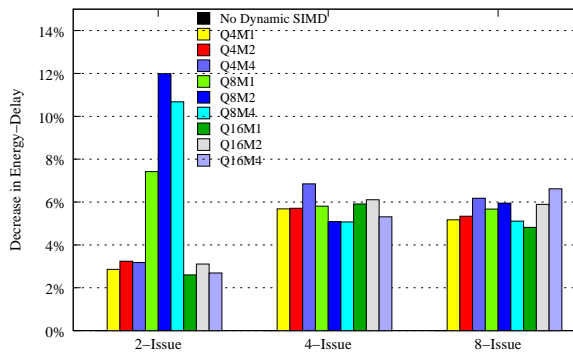


Figure 4: Reduction of energy-delay product for various DSIMD configurations

4 Analysis

The results for fetch savings come as no surprise. Rather, they serve to confirm one of our original assumptions that it is quite common for independent threads to be executing instructions that are very close in their PCs, and that they can be brought into lockstep for extended amounts of cycles. The slight correlation of issue width and fetch savings can be explained by the fact that a greater width gives the processor more choice in executing instructions. Therefore, threads are less likely to fall out of lockstep due to data cache misses and other high-latency events that would differ between the threads.

Although one might intuitively expect all-around performance degradation with a technique based on stalling threads, our results show that this is not actually the case. There are a few reasons for this. The greatest of these is that threads running in lockstep need not contend for blocks in the instruction cache. One commonly cited disadvantage of multithreading is cache contention, with which different threads are likely to be accessing different areas of memory and thrashing a conservatively sized cache [2]. In the case of SMT fetching, two threads that are distant in their PCs will fight for space in the cache, whereas threads in lockstep will harmoniously attempt to fetch the same instruction each cycle. The resulting decrease in the instruction cache miss rate can boost performance enough to overcome fetches lost due to stalling and then some.

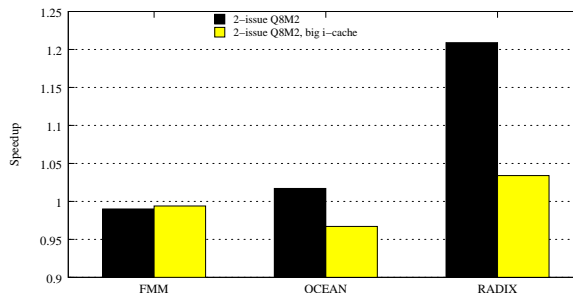


Figure 5: DSIMD performance with a large instruction cache

We tested this explanation by running simulations with an impractically large instruction cache of 256KB, as compared to our baseline value of 16KB. The results in Figure 4 show that for configurations with which there was originally a significant speedup, the speedup is greatly reduced with the expanded cache.

The implications of these results are twofold. First, they confirm our hypothesis that reduced L1 instruction cache contention was the dominant factor in the performance increase we observed. Second, and more importantly, they indicate that DSIMD increases power efficiency in three distinct ways: first, by reducing the power consumed by the instruction cache; second, by reducing the power consumed by the L2 cache (since it need not be accessed as often); and finally, by decreasing execution time.

4.1 Benchmark-Specific Analysis

Even within the set of embarrassingly-parallel, identically-threaded workloads for which DSIMD is appropriate, there is significant variation in its efficacy. Although for all three benchmarks we observed a reduction in the energy-delay product, the difference between them is cause for further analysis.

4.1.1 FMM

FMM’s reduction in energy-delay product is much less than observed for the other benchmarks. Execution time is virtually unaffected for this benchmark; since it spends the majority of its time in a small amount of code, there is little to gain from reduced instruction cache contention. DSIMD’s modest reduction in fetch bandwidth for this benchmark still allows for some reduction in power consumption, however. In particular, the Q8M2 configuration performs consistently well.

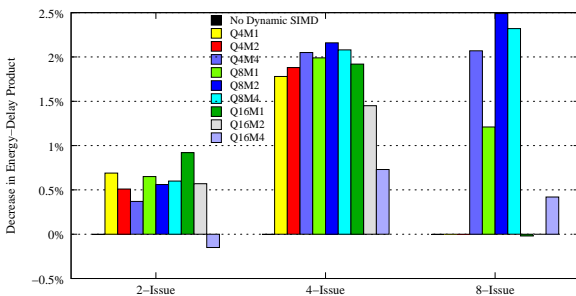


Figure 6: Energy-delay product reduction for FMM

It is interesting to note the nonlinear correlation between fetch savings and energy-delay product in this case. While performance is virtually unaffected by most DSIMD configurations, the excessive stall penalty imposed by the Q16M4 configuration does cause a performance reduction. (In the event one thread stalls but the two do not ultimately line up, as many as 64 cycles may be wasted.) Thus, although the maximum number of fetches are avoided in this case, the energy-delay product actually falls. This effect makes configurations like

Q8M2 quite attractive: they provide a tighter bound on possible performance degradation.

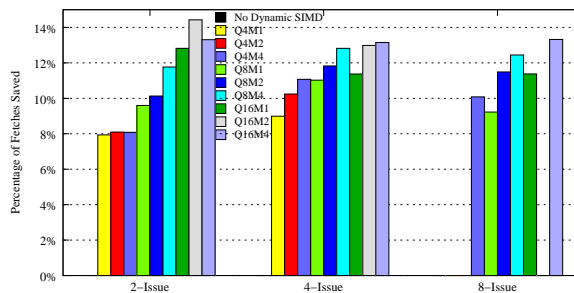


Figure 7: Fetch bandwidth reduction for FMM

4.1.2 OCEAN

OCEAN fares considerably better than FMM. Because the frequently-accessed code size in this benchmark is larger, it receives a speedup (see Figure 8) in every DSIMD configuration tested on the two-wide and four-wide cores, and on all configurations with a queue size of less than 16 on the 8-wide core. Interestingly, the amount of fetches saved varies little with the queue size and multiplier; across the board, 40% of fetches are obviated. Since, in a two-thread configuration, the maximum fetch bandwidth reduction is 50%, we conclude that OCEAN’s code is inherently suited to DSIMD—it has few data-dependent branches and good data cache performance—so threads naturally tend to line up for many cycles, regardless of configuration.

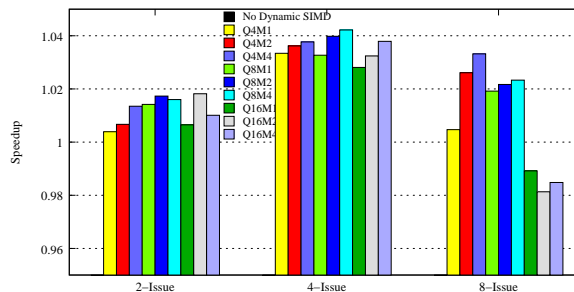


Figure 8: OCEAN speedup

As Figure 9 indicates, the handy performance boost and considerable fetch bandwidth reduction cause OCEAN’s energy-delay product to decrease by as much as 10%. While the power savings increases monotonically with the stall timeout factor M, it is still hard to recommend a configuration such as Q16M4 because of the potential for performance reduction on some programs. Q8M2 again performs well across the board.

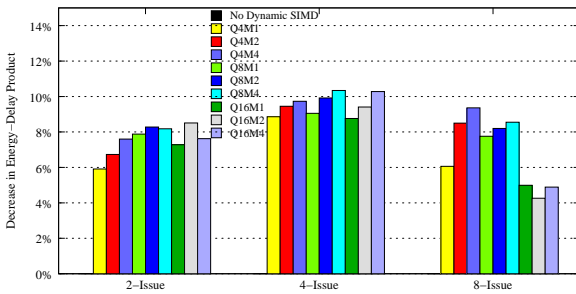


Figure 9: Energy-delay product reduction for OCEAN

4.1.3 RADIX

DSIMD improves the energy-delay product on the radix sort quite well in general. The fetch bandwidth savings are particularly compelling: on the eight-issue core, nearly every possible fetch is obviated, with a savings of 49.4% across the board. It is of interest that the fetch savings is considerably greater for the widest core; we reason that increased opportunities for out-of-order scheduling allow the processor to better hide the effects of data cache misses, so the threads are able to remain in lockstep for longer stretches (and if they do get out of lockstep, they are closer together).

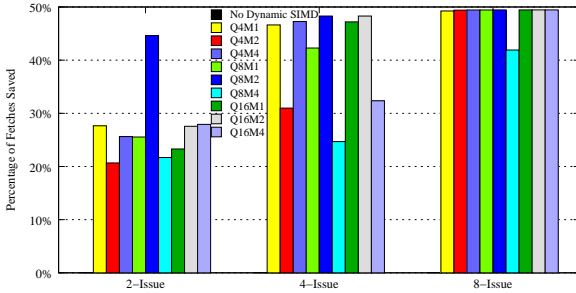


Figure 10: Fetch bandwidth reduction for RADIX

The high fraction of fetches saved results in nearly the maximum power savings possible under our model. In turn, we see a considerable decrease in energy-delay product across the board. We are forced to conclude, however, that the considerable performance increase in the two-issue Q8 configurations (which accounts for the energy-delay product decreases of 13% and greater) is an outlier and is not predictive of performance on many real workloads. As we demonstrated earlier, the speedup in this case is attributable to DSIMD reducing cache thrashing—the speedup falls to unity with a large L1 instruction cache—but we do not expect typical performance gains of this magnitude. The radix sort does represent something of a toy benchmark.

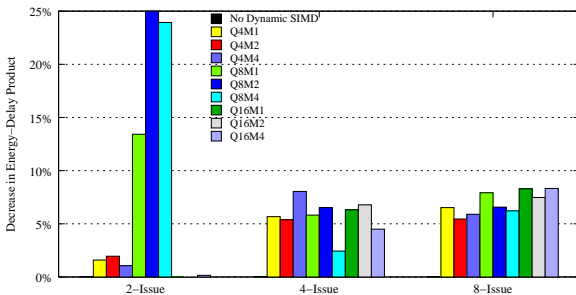


Figure 11: Energy-delay product reduction for RADIX

5 Extending DSIMD

With this being a first attempt at examining the usefulness of DSIMD, there exists much potential for extensions and improvements on the techniques we present. For one, we were only able to get a peek at the space of processor configurations and benchmarks. Relationships of DSIMD performance to parameters of relevant structures, such as the instruction cache, the decode logic, and the issue units should be explored. The benchmarks and their execution can be examined in greater detail to see specifically what types of code blocks DSIMD functions best with.

Our current method of determining when to stall a thread is rather simple: if two threads differ by a small constant in their PC, stall the leading thread up to a maximum of some multiple of that difference to try to bring the threads into lockstep. One can envision more intelligent methods of determining when to stall, including examination of instruction traces and dynamic enabling of DSIMD based on how well it is working over a particular stretch. These will, of course, incur a greater hardware overhead, and exploring this tradeoff could be an interesting area of research.

With a mind toward saving power, we can also consider what other work is being duplicated when threads are in lockstep besides fetching, such as decode. It is possible that the power savings could be further enhanced by not duplicating this other work, especially on CISC architectures in which decode is a power-hungry, multi-stage operation.

It would also be interesting to extend this work beyond SMT processors with two contexts. Although most existing SMT implementations use two-way, including the Pentium 4 and the POWER5, it would be relevant to consider the effect of having more contexts—especially since the fraction of avoidable fetches increases with the number of identical threads, and the impact of cache contention increases, too. However, such an analysis must also consider the power implications of supporting more than two thread contexts; the power efficiency of SMT, in general, may not be monotonic in the number of threads.

6 Conclusion

Power dissipation has become a very significant design constraint in modern microprocessors. High-performance processors must be able to reach their peak activity levels without overheating to the point of failure, and there is a strong demand for energy-efficient mobile devices that will give long battery life. Therefore, techniques that can reduce power consumption with minimal negative effect performance are especially useful for today’s microarchitectures.

We have shown that DSIMD can be considered in this category. For the benchmarks used, performance losses due to stalled threads were compensated by a decrease in instruction cache contention, in many cases to the point of having a net performance gain. Total fetches, and by

extension, average fetching power, were reduced by 24% at minimum. These power reductions and potential performance gains can be achieved with a minimal increase in hardware.

References

- [1] Tullsen, Dean and Eggers, Susan and Emer, Joel and Levy, Henry and Lo, Jack and Stamm, Rebecca. *Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor* May, 1996.
- [2] Eggers, Susan and Emer, Joel and Levy, Henry and Lo, Jack and Stamm, Rebecca and Tullsen, Dean. *Simultaneous Multithreading: A Platform for Next-generation Processors* September, 1997.
- [3] Seng, John and Tullsen, Dean and Cai, George. *Power-Sensitive Multithreaded Architecture* September, 2000.
- [4] Kalla, Ron and Sinharoy, Balaram and Tendler, Joel. *IBM POWER5 Chip: A Dual-Core Multithreaded Processor* March, 2004.
- [5] Hinton, Glenn et al. *The Microarchitecture of the Pentium 4 Processor* February, 2001.
- [6] Montanaro, James et al. *A 160-MHz, 32-b, 0.5-W CMOS RISC Microprocessor* November, 1996.
- [7] Gonzalez, Ricardo and Horowitz, Mark. *Energy Dissipation In General Purpose Microprocessors* September, 1996.
- [8] Renau, Jose and Fraguera, Basilio and Tuck, James and Liu, Wei and Privulovic, Milos and Ceze, Luis and Sarangi, Smruti and Sack, Paul and Strauss, Karin and Montesinos, Pablo. *SESC Simulator*. <http://sesc.sourceforge.net>. 2002.
- [9] Woo, Steven and Ohara, Moriyoshi and Torrie, Evan and Singh, Jaswinder and Gupta, Anoop. *The SPLASH-2 Programs: Characterization and Methodological Considerations* 22nd International Symposium on Computer Architecture, June, 1995.