Performance/Energy Optimization of DSP Transforms on the XScale Processor

Paolo D'Alberto, Markus Püschel, and Franz Franchetti
Carnegie Mellon University
Department of Electric and Computer Engineering
{*pdalbert, pueschel,franzf*}@*ece.cmu.edu*

September 25, 2006

Abstract

The XScale processor family provides user-controllable independent scaling configuration of CPU, bus, and memory frequencies. This feature introduces another handle for the code optimization with respect to energy consumption or runtime performance. We quantify the effect of frequency configurations on both performance and energy for three signal processing transforms: DFT, FIR filters, and WHT.

To do this, we use SPIRAL, a program generation system for signal processing transforms. For a given transform to be implemented, SPIRAL searches over different algorithms to find the best match to the given platform w.r.t. the chosen performance metric (usually runtime). In this paper we use SPIRAL to generate different implementations for different frequency configuration, optimized for runtime and energy consumption (physically measured). In doing so we show that first, each transform achieves best performance/energy consumption for a different system configuration; second, the best code depends on architecture configuration, problem size and algorithm; third, the fastest implementation is not always the most energy efficient; fourth, we introduce dynamic (i.e., during execution) reconfiguration in order to further improve performance/energy. Finally, we benchmark SPIRAL generated code against Intel's vendor library routines. We show competitive results as well as 20% performance improvements or energy reduction for selected transforms and problem sizes.

1 Introduction

The rapidly increasing complexity of computing platforms keeps application developers under constant pressure to rewrite and re-optimize their software. A typical

micro-architecture may feature one or multiple processors with several levels of memory hierarchy, special instruction sets, or software-controlled caches. One of the recent additions to this list of features is software-controlled scaling of the CPU core frequency. The idea is to enable the user (or the operating system) to scale up or down the CPU frequency and the supply voltage to save energy; this is especially important for devices operating on limited power sources such as batteries. Frequency scaling is available for different processors, such as Athlon 64, Pentium M, and the XScale processor family (a fixed-point processor targeted for embedded applications).

XScale systems provide more reconfigurability options, namely the independent selection (i.e., to a certain degree) of CPU, bus, and memory frequency. Reconfigurability complicates the code generation and optimization process because different configurations correspond to different platforms. However, reconfigurability is crucial in the high-performance/power-aware signal-processing domain.

Contribution of this paper. We consider three linear transforms: finite impulse response (FIR) filter, the discrete Fourier transform (DFT), and the Walsh-Hadamard transform (WHT). Our test platform is a SITSANG board with an XScale PXA255 fixed-point processor with no voltage scaling. To perform the experiments, we integrated frequency scaling in the automatic code generation and optimization framework SPIRAL [1]. Using SPIRAL, we generated code tuned for different frequency settings or to a dynamic frequency scaling strategy.

In this work, we show: First, code adaptation to one specific or the best setting can yield up to 20% higher performance or energy reduction than using an implementation optimized for a different setting (e.g., the fastest CPU vs. the fastest memory). Second, there are algorithms and configurations that achieve the same performance but have a 20% different energy consumption. For example, the fastest code–configuration can consume 5% more than the most energy efficient code–configuration. Third, we apply dynamic scaling and we are able to reduce energy consumption; however, this technique is not helpful for performance. Finally, we show that SPIRAL's codes compares favorably against the hand-tuned Intel's vendor library IPP, which is oblivious to the frequency configuration.

Related work. Different frequency settings yield memory hierarchies with different characteristics. Thus, a code generation tool that enables the tuning of codes to the architecture's characteristics is an ideal solution. Examples of such tools include for linear algebra kernels ATLAS [2] and FLAME [3], for the DFT and related transforms FFTW [4], and for general linear signal transform SPIRAL, which is used in this paper.

Other approaches to optimization in the presence of reconfigurable features or frequency scaling consider more general code and larger scale problems, in contrast to the small but crucial transform kernels considered here. Examples include

compiler techniques [5], power modeling [6] or software-hardware application assists driving runtime adaptations [7].

Hsu and Kremer [8] and Xie et al. [9] present a compile-time algorithm for the intra-application of dynamic voltage scaling, however the application remains practically unaware and unchanged. We consider the possibility that the code adapts and changes for each different configuration in response of the different system configurations.

Organization of the paper. In Section 2, we provide details of our platform and an overview of our code generator. In Section 3, we introduce the specific framework used to collect our results. In Section 4, we present experimental results for DFT, FIR and WHT. We conclude in Section 5.

2 Background

In this section, we describe the reconfigurable features of the XScale architecture and the inner works of the SPIRAL framework.

2.1 Intel XScale PXA255

The Intel XScale architecture targets embedded devices. One crucial feature is the hardware support for energy conservation and high burst performance. Specifically, applications may control the frequency settings of the machine's CPU, bus, and memory. In this paper, we consider the PXA255, a fixed-point processor in the XScale family [10] with no voltage scaling, which we call simply **XScale**.

Frequency configuration. A frequency configuration is given by a memory frequency m (one of 99 MHz, 132 MHz, or 165 MHz), a bus multiplier α (one of 1, 2, or 4) and, a CPU multiplier β (one of 1, 1.5, 2, or 3). When we choose a configuration triple (m, α, β), the memory frequency is set to m, the bus frequency to $\alpha m/2$ and the CPU frequency to $\alpha\beta m$. Out of 36 possible choices for (m, α, β), not all are suggested nor necessarily stable. In this paper, we consider a representative set of 13 configurations that are stable for the DSP transforms considered. The configurations are summarized in Table 1. The frequencies are given in MHz and each setting is assigned a mnemonic name that specifies the CPU frequency, and the ratio of memory and bus frequency to the CPU frequency, respectively. For example, 530-1/4-1/2 means that the memory runs at a quarter, and the bus at half of the CPU speed.

A change of configuration is not instantaneous and it is done by writing appropriate configuration bits to a control register (CCCR [10]); we have measured an average penalty of 530 μs.

CPU	Memory	Bus	Name
597	99	99	597-1/6-1/6
530	132	265	530-1/4-1/2
530	132	132	530-1/4-1/4
497	165	165	497-1/3-1/3
398	99	199	398-1/4-1/2
398	99	99	398-1/4-1/4
331	165	165	331-1/2-1/2
298	99	49	298-1/3-1/6
265	132	132	265-1/2-1/2
199	99	99	199-1/2-1/2
165	165	82	165-1-1/2
132	132	66	132-1-1/2
99	99	49	99-1-1/2

Table 1: PXA255 Configurations: The frequencies are in MHz and 398-1/4-1/4 is the startup setting.

From the prospective of a software developer, the problem is at least two-fold. First, different configurations correspond to different platforms and thus code optimized for one platform may be suboptimal for another. Second, the platform choice is not straightforward. For example, if the highest performance is desired, there are three candidate settings: 597-1/6-1/6 (fastest CPU), 497-1/3-1/3 (fastest memory), and 530-1/4-1/2 (fastest bus). Energy constraints may further complicate the selection.

2.2 SPIRAL

SPIRAL is a code generator for linear signal transforms such as the discrete Fourier transform (DFT), the Walsh-Hadamard transform (WHT), the discrete cosine and sine transforms, FIR filters (FIR), and the discrete wavelet transform. The input to SPIRAL is a formally specified transform (e.g., DFT of size 245), the output is a highly optimized C program implementing the transform. SPIRAL generates fixed-point code for platforms such as XScale.

In the following, we first provide some details on transforms and their algorithms, then we explain the inner workings of SPIRAL.

Transforms and algorithms. We consider three transforms: FIR filters, DFT and WHT; each transform is defined by a matrix, which multiplied by an input

vector x yields an output vector y; for example, the DFT matrix follows:

$$\mathbf{DFT}_n = [\omega_n^{k\ell}]_{0\le k,\ell<n}, \quad \omega_n = e^{-2\pi i/n} \tag{1}$$

The input and output length of DFT is n.

Algorithms for these transforms are sparse structure factorizations of the transform matrix. For example, the Cooley-Tukey fast Fourier transform (FFT) follows:

$$\mathbf{DFT}_{km} = (\mathbf{DFT}_k \otimes I_m) D (I_k \otimes \mathbf{DFT}_m) P, \tag{2}$$

Here, I_m is the $m \times m$ identity matrix; D is a diagonal matrix, and P is a permutation matrix, both depending on k and m (see [11] for details).

Algorithms for WHT and filters can be described similarly; for filters may include different choices of blocking, Karatsuba, and frequency domain methods [12, 13].

How SPIRAL works. In SPIRAL, a decomposition like (2) is called a *rule*. For a given transform, SPIRAL recursively applies these rules to generate one algorithm represented as a formula. This formula is then structurally optimized using a rewriting system and finally translated into a C program (for computing the transform) using a special purpose compiler. The C program is further optimized and then a native compiler is used to generate an executable. Its runtime is measured and fed into a search engine, which decides how to modify the algorithm; that is, the engine changes the formula, and thus the code, by using a dynamic-programming search. Eventually, this feedback loop terminates and outputs the fastest program found in the search. The entire process is visualized in Figure 1 (see [1, 14] for a complete description).

Note that there is a large degree of freedom in creating a formula, or algorithm, for a given transform due to the choices of decomposition in each step. For example, a DFT of size 16 could derive three different factorizations using (2) such as $16{=}2{\times}8$, or $4{\times}4$, or $8{\times}2$. Similar choices apply recursively to the smaller transforms obtained after each decomposition.

3 Extension of SPIRAL

In this work, our goal is to generate automatically transform codes for the XScale. These codes are optimized specifically to every frequency configuration (see Table 1). As optimization metric, we use both runtime performance and energy consumption. To achieve our goal, we extended SPIRAL in two directions. First, we included a framework that embeds frequency scaling at the formula level and thus into code. Second, we let SPIRAL run using the energy consumption as measure for its feedback loop (see Figure 1).

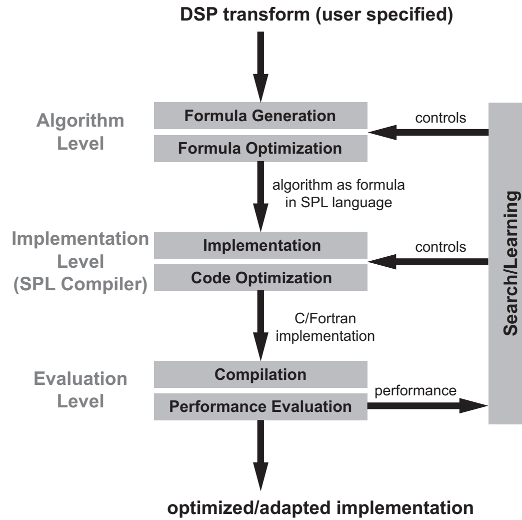

Figure 1: The code generator SPIRAL.

3.1 Frequency scaling in SPIRAL

Static frequency scaling. We enable SPIRAL to generate code for different frequency configurations by a transform-transparent tagging framework that starts at the formula level. The basic idea is simple. Any formula F generated by SPIRAL can be tagged with a frequency configuration, for example 497-1/3-1/3, written as

$$[F]_{497-1/3-1/3} \tag{3}$$

Next, we extended the SPL compiler (see Figure 1) to understand these tags and translate them into the appropriate code. In the example (3), the entire formula would be executed at 497-1/3-1/3 with a potential[1] switch at the the beginning and at the end. We call this *static* frequency scaling.

Dynamic frequency scaling:. The same technique is used to perform *dynamic* frequency scaling; that is, to perform different parts of the formula at different configurations. This is explained next, starting with a motivation. Consider the decomposition rule for the DFT in (1). First, the input vector x is multiplied by $(I_k \otimes \mathbf{DFT}_m)$. The definition of the tensor product implies that this corresponds to a loop with k iterations. The loop body calls $\mathbf{DFT}_m$ on a contiguous subvector of x of length m. This access pattern yields good cache utilization and, thus, high performance. It is *compute-bound*.

[1]We never perform unnecessary switches as it is very cheap to check whether the processor already runs at the desired configuration.

The second part, $(\mathbf{DFT}_k \otimes I_m)$ is also a loop, but with m iterations. Further, in the loop body $\mathbf{DFT}_k$ accesses a k-element long subvector of x, but at stride m. If m is sufficiently large, this is effectively equivalent to reducing the cache size (unless the cache is fully associative) and may lead to cache thrashing. The computation thus becomes *memory-bound*.

The basic idea is now to run both parts at different settings. Using tags and an example, this can be expressed as

$$[(\mathbf{DFT}_k \otimes I_m)]_{165-1-1/2} \cdot [(I_k \otimes \mathbf{DFT}_m)]_{530-1/4-1/2} \quad (4)$$

The tag on the right has a higher CPU and bus speed and the tag on the left has a higher memory speed. SPIRAL will generate the corresponding code for easy evaluation. The question is how to distribute the tags in the formula. This is explained next.

TagIt$(F, Csize, c, m)$

Require: F is WHT/DFT, c base configuration, m alternative conf.

1: **if** $|F| \leq Csize$ **then**
2:     **return** $[F]_c$
3: **else**
4:     $\forall k \leq \ell$, s.t. $k * \ell = |F|$. (e.g., DFT rule (1))
5:     $f = [F_k \otimes I_\ell]_m (I_k \otimes$ **TagIt**$(F_\ell, Csize, c, m))$
6:     **return** f
7: **end if**

Table 2: Algorithm to determine F tagged.

Algorithm. We included an algorithm (see Table 2) for tagging a given formula into SPIRAL. The algorithm applies to WHTs and DFTs. FIR filters are structured differently; they are compute-bound for all input sizes. The input to the tagging algorithm is the cache size, two frequency configurations c and m to be assigned to memory and compute-bound formula parts respectively, and a formula F. The algorithm recursively descends the formula expression tree and assigns tags. The c tag is assigned once a subformula has an input that fits into the cache.

In the experiments, this algorithm is combined with search over the different formulas of the transform.

3.2 Performance Measurement

We installed SPIRAL on a desktop (host machine) and connected the XScale board to the local network. On the host, SPIRAL generates tagged formulas, translates

them into code, cross-compiles for the XScale, and builds a loadable kernel module (LKM). Even though, floating-point codes run on this processor, they are inefficient because software emulated. To speed up the computation, SPIRAL generates fixed-point code directly (i.e., 14-bit precision on native 32-bit integers). We measure runtime or energy as explained next.

Runtime. We upload the LKM into the board. We first execute the code once (hence, we "warm up" the caches), and then measure a sufficient number of iterations. Finally, we return the runtime to the host and to SPIRAL's search engine to close the feedback loop.

Energy. The XScale board has a 3.5V battery ($U = 3.5V$) as its power source. To measure energy, we unplug all external sources and we measure, sample, and collect the out-coming battery current through a digital multi meter (DMM).[2] The energy is measured using the following procedure: First, we measure the transform execution time, t, (as explained previously) and determine the number of iterations sufficient to let the board run the transform for about 10 seconds. Second, we turn off all peripherals power supplies (e.g., LCD) and we take 512 samples 2 ms apart (a sampling period of about one second) of the battery current, then we compute the average current I. Third, we determine the energy by the formula $E = UIt$. Notice that we assume that the battery voltage is anchored to its nominal value. This energy value is sent back to the host system and SPIRAL to close the feedback loop.

4 Experimental Results

We consider the following transforms: DFT, WHT, and 8-tap and 16-tap FIR filters. For each transform, we use SPIRAL in separate searches for each configuration to find the code optimized for runtime or energy. Runtime and energy measurements are performed as explained in Section 3.2. We use *gcc 3.4.2* to compile all the codes and and we used *crosstool* to build the cross compiler. In the following figures, we show performance for seven (of the 13) configurations.

The performance is reported as the $operations/runtime$ ratio and measured in pseudo Mop/s (million operations per second). We exclude from the operations count the index computations and we assume $5n\log_2(n)$ for the DFT, $n\log_2(n)$ for the WHT, and $n(2d-1)$ for a d-tap filter. The energy is reported as the $operations/energy$ ratio and measured in pseudo Mop/J (million operations per Joule). These metrics (performance and energy efficiency) preserves runtime or energy relations.

[2] We use an Agilent 34401A.

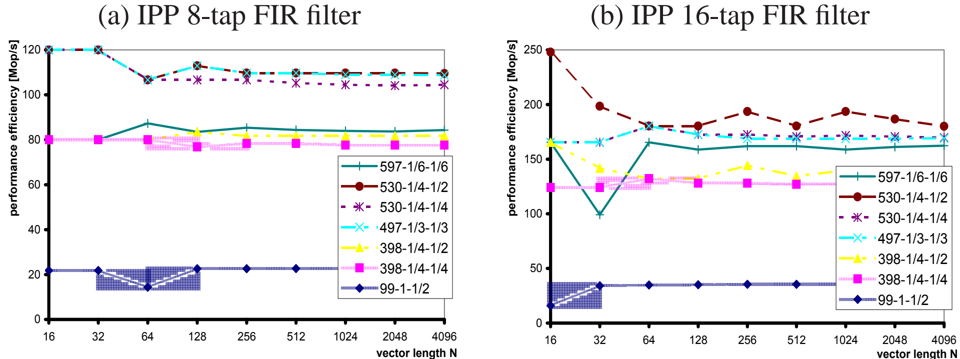

Figure 2: Performance (in pseudo Mop/s) of IPP FIR filters: (a) 8-tap FIR filter, and (b) 16-tap FIR filter.

We use the Intel vendor library IPP 4.1 as benchmark except for the WHT (not provided) and for DFT of sizes larger than 2^{12} (suggested range for IPP). IPP provides one implementation, which is oblivious of any given configurations; in contrast, SPIRAL generates specific codes for each configuration.

4.1 Runtime Performance Results

General behavior. We achieve peak performance for DFT (Figure 3.(a) and Figure 4) and WHT (Figure 3.(b)) for problems fitting the cache and we have a clear slowdown for larger problems. This slowdown is because the transforms presents a limited data locality and strided access pattern, which produces cache thrashing (see also the discussion in Section 3.1). For FIR filters (Figure 3.(c) and (d) and Figure 2), in contrast, the performance is roughly constant across sizes due to the local, consecutive access of the input.

Best configuration. For DFT and FIR, there is only one best configuration independently of the problem size (Figure 3.(a), (c), and (d); Figure 4, and Figure 2)). For WHT, the best configuration is a function of the problem size; that is, whether or not the problem fit into cache (Figure 3.(b)). The difference, however, is less than 10%. For DFT, 530-1/4-1/2 (high bus speed) is the best configuration, whereas for FIR filters it is 597-1/6-1/6 (fast CPU speed). This is consistent with the access patterns of the transforms already discussed previously. Note that 597-1/6-1/6 performs poorly with DFT and WHT.

SPIRAL vs. IPP. In Figure 4.(a), we present the relative speed of the DFT code generated by SPIRAL (Figure 3.(a)) versus the one by IPP (Figure 4.(b)); the higher a line is, the faster SPIRAL's code is. For problem sizes $N = 64$, 128

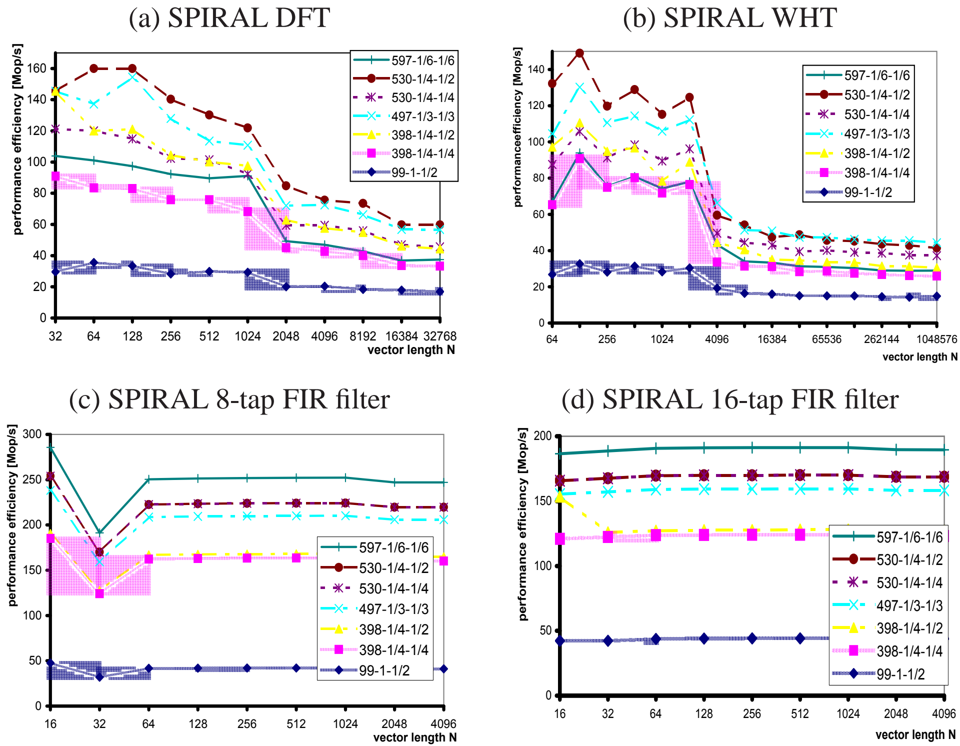

Figure 3: Performance (in pseudo Mop/s) of SPIRAL generated code: (a) DFT, (b) WHT, (c) 8-tap FIR filter, and (d) 16-tap FIR filter.

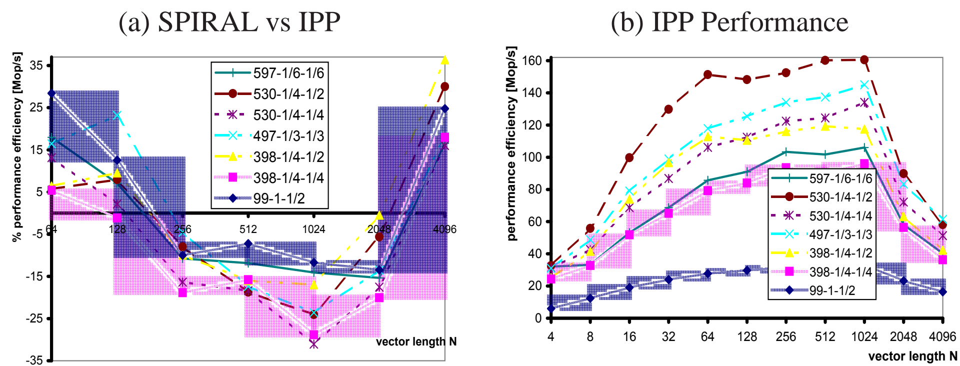

Figure 4: (a) Relative performance of SPIRAL DFT / IPP DFT. The higher a line is, the faster SPIRAL is. (b) IPP DFT performance (in pseudo Mop/s).

and 4096, SPIRAL code clearly is faster; in contrast, it is slower for $N = 256$, . . . , 2048. For most problem sizes, the relative speed of SPIRAL versus IPP varies more than 10% points for different configurations. For example, for $N = 128$, SPIRAL generated code is as fast as the IPP code in configuration 398-1/4-1/4, but almost 25% faster in configuration 497-1/3-1/3.

SPIRAL's codes for 8-tap FIR filters (Figure 3.(c)) outperforms the respective IPP's routine (Figure 2.(a)) by a factor of two. In the case of 16-tap FIR filters, SPIRAL (Figure 3.(d)) and IPP (Figure 2.(b)) have equivalent performance.

IPP does not provide a WHT library function.

4.2 Energy Results

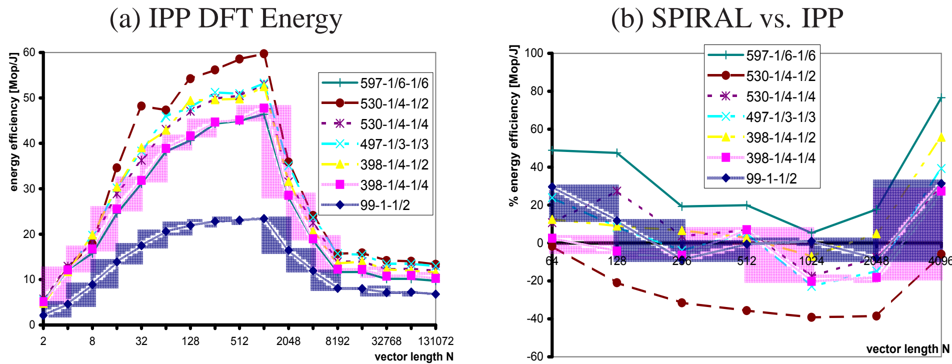

Figure 5: (a) IPP DFT energy efficiency (in pseudo Mop/J). (b) Energy gain and loss of SPIRAL DFT / IPP DFT. Higher means SPIRAL is more frugal.

For SPIRAL's DFTs, the configuration with the highest energy efficiency (Figure 6.(a)) depends on the problem size and it is a compromise among the speed of CPU, bus and memory. For example, 398-1/4-1/2 is clearly best for sizes 8, 16, and 32. For the SPIRAL's WHT (Figure 6.(b)) we observe a in-cache behavior (530-1/4-1/4) and out-cache behavior (398-1/4-1/2). For SPIRAL's and IPP's FIR filters (Figure 6.(c),(d) and Figure 7), there is only one *good* configuration.

SPIRAL vs. IPP. In Figure 5.(b), we present the relative energy efficiency of SPIRAL generated DFT code (Figure 6.(a)) versus the IPP DFT library function (Figure 5.(b)). SPIRAL's DFTs relative energy to the IPP DFT library function is similar to the relative performance (Figure 5.(b) to Figure 4.(a)).

SPIRAL's 8-tap FIRs (Figure 3.(c)) have a threefold energy efficiency w.r.t. IPP's FIRs (Figure 2.(a)). The case of 16-tap FIR filters shows that the performance of SPIRAL's FIRs (Figure 3.(d)) is equal to IPP's (Figure 2.(b)); however,

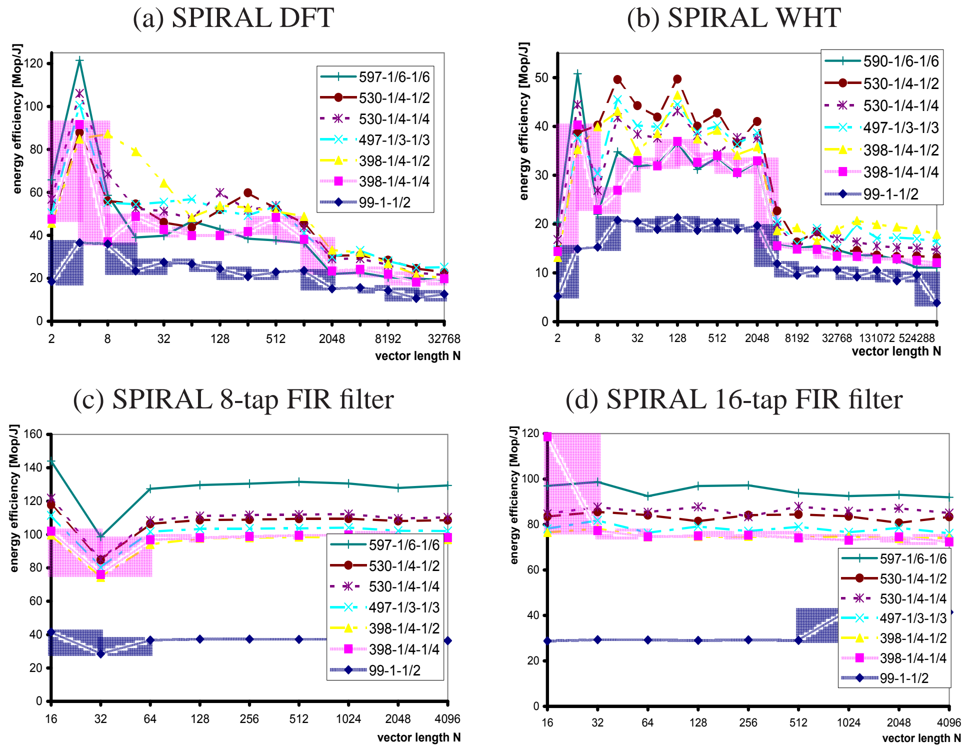

Figure 6: Energy efficiency (in pseudo Mop/J) of SPIRAL generated code: (a) DFT, (b) WHT, (c) 8-tap FIR filter, and (d) 16-tap FIR filter.

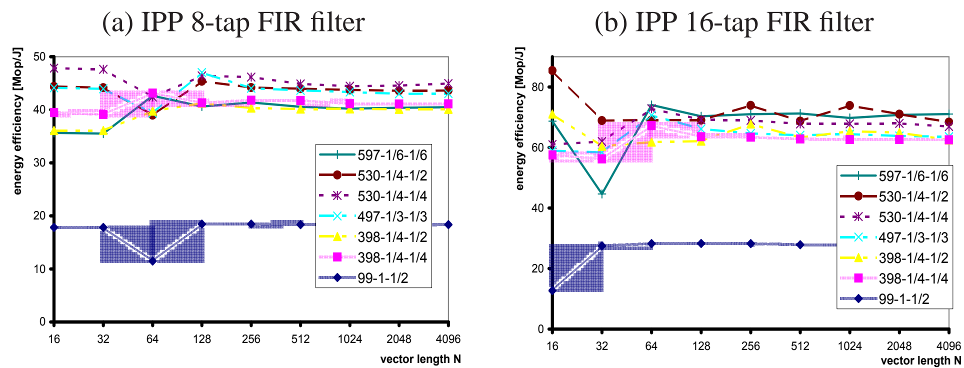

Figure 7: Energy efficiency (in pseudo Mop/J) of IPP FIR filters: (a) 8-tap FIR filter, and (b) 16-tap FIR filter.

SPIRAL's FIRs outperform IPP's by 20% in energy efficiency (Figure 6.(d) and Figure 7.(b)).

4.3 Dynamic Frequency Scaling (DFS)

For the application of DFS, there must be two configurations for which there is an advantage to switch as a function of the problem size. This is the case only for the WHT (Figure 3.(b) and Figure 6.(b)). Briefly, we first choose two configurations. Then, for all problem sizes that do not fit into the cache ($n \geq N = 2^{16}$) we apply Algorithm 2. This way, SPIRAL finds a WHT implementation that switches between the chosen configurations and automatically determines the trade off between the overhead and the performance/energy gains obtained by switching, thus the overall performance for the given metric.

We found that DFS is not helpful for performance because of the long switching overhead (530μs). However, DFS is helpful for energy.

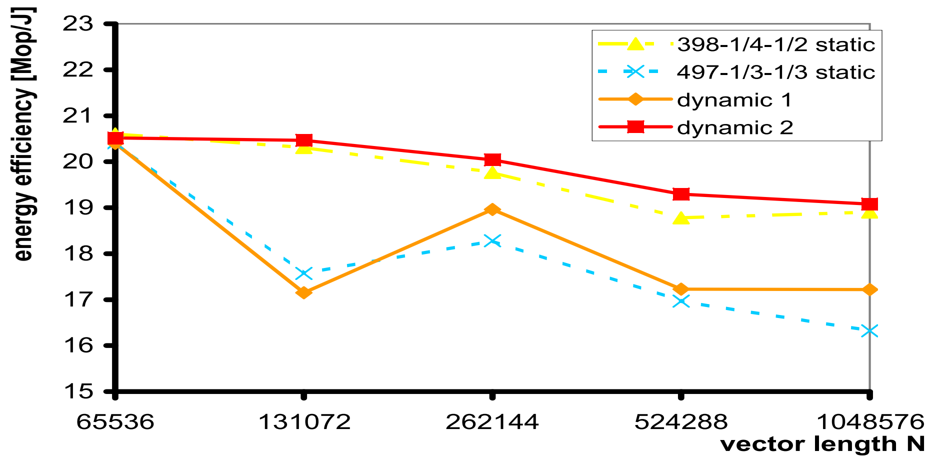

Figure 8: Dynamic frequency scaling of SPIRAL WHT, switching between 497-1/3-1/3 and 398-1/4-1/4.

By direct feedback, we have found two leading configurations: 398-1/4-1/2 and 497-1/3-1/3. We start using the most energy efficient configuration 398-1/4-1/2 as base configuration (i.e., "398-1/4-1/2 static" in Figure 8). We switch to the faster configuration 497-1/3-1/3 for all sub-formulae of shape $\mathbf{WHT}_{2^3} \otimes I_{2^n}$ (i.e., "dynamic 1" in Figure 8). We improve the energy because we improve the execution time. In fact, the computation $\mathbf{WHT}_{2^3} \otimes I_{2^n}$ trashes the small 2-way *mini cache* and a faster CPU/memory shortens the cache miss penalty.

If we start using the fastest configuration 497-1/3-1/3 as base configuration (i.e., "497-1/3-1/3 static" in Figure 8) we switch to the slower configuration 398-1/4-1/2 for all sub-formulae of shape $\mathbf{WHT}_{2^2} \otimes I_{2^n}$ (i.e., "dynamic 2" in Figure 8). We can slow down the CPU, memory and bus with an energy gain (with a small performance loss).

5 Conclusions

We show how a feed-back based framework as SPIRAL can be used to generate and optimize efficient DSP applications such as DFT, WHT, and FIR filters on the XScale embedded platform. We support frequency scaling and thus automatically generate and optimize codes for different configurations and codes. Our experiments show that the best configuration depends on the DSP kernel, metric and (sometimes even on) the problem size.

References

[1] M. Püschel, J.M.F. Moura, J. Johnson, D. Padua, M. Veloso, B.W. Singer, J. Xiong, F. Franchetti, A. Gačić, Y. Voronenko, K. Chen, R.W. Johnson, and N. Rizzolo, "SPIRAL: Code generation for DSP transforms," *Proc. of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, vol. 93, no. 2, 2005.

[2] R.C. Whaley, A. Petitet, and J.J. Dongarra, "Automated empirical optimization of software and the ATLAS project," *Parallel Computing*, vol. 27, no. 1–2, pp. 3–35, 2001, Note #147, UT-CS-00-448, 2000.

[3] P. Bientinesi, J.A. Gunnels, M.E. Myers, E.S. Quintana-Ortí, and R.A. van de Geijn, "The science of deriving dense linear algebra algorithms," *ACM Transactions on Mathematical Software*, vol. 31, no. 1, Mar 2005.

[4] M. Frigo and S. Johnson, "The design and implementation of FFTW3," *Proc. of the IEEE, special issue on "Program Generation, Optimization, and Adaptation*, vol. 93, no. 2, pp. 216–231, 2005.

[5] A. Halambi, A. Shrivastava, N. Dutt, and A. Nicolau, "A customizable compiler framework for embedded systems," in *Proc. of the 5th International Workshop on Software and Compilers for Embedded Systems*, 2001.

[6] G. Contreras and M. Martonosi, "Power prediction for intel XScale; processors using performance monitoring unit events," in *ISLPED '05: Proceedings*

of the 2005 international symposium on Low power electronics and design, New York, NY, USA, 2005, pp. 221–226, ACM Press.

[7] L. Singleton, C. Poellabauer, and K. Schwan, "Monitoring of cache miss rates for accurate dynamic voltage and frequency scaling," in *Proc. of the 12th Annual Multimedia Computing and Networking Conference*, SanJose, Jan, 2005.

[8] C. Hsu and U. Kremer, "The design, implementation, and evaluation of a compiler algorithm for cpu energy reduction," in *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, New York, NY, USA, 2003, pp. 38–48, ACM Press.

[9] F. Xie, M. Martonosi, and S. Malik, "Compile-time dynamic voltage scaling settings: Opportunities and limits," in *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, New York, NY, USA, 2003, pp. 49–62, ACM Press.

[10] Intel, *Intel XScale Microarchitecture*, 2001.

[11] C. Van Loan, *Computational Framework of the Fast Fourier Transform*, SIAM, 1992.

[12] A. Gačić, M. Püschel, and J.M.F. Moura, "Fast automatic implementations of FIR filters," in *Proc. International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, 2003, vol. 2, pp. 541–544.

[13] A. Gačić, *Automatic Implementation and Platform Adaptation of Discrete Filtering and Wavelet Algorithms*, Ph.D. thesis, Electrical and Computer Engineering, Carnegie Mellon University, 2004.

[14] F. Franchetti, Y. Voronenko, and M. Püschel, "Formal loop merging for signal transforms," in *Proc. of Programming Language Design and Implementation*, Chicago, Jun. 2005.