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
Pittsburgh, PA, USA
{pdalbert,pueschel,franzf}@ece.cmu.edu

Abstract. The XScale processor family provides user-controllable independent 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: the discrete Fourier transform (DFT), finite impulse response (FIR) filters, and the Walsh-Hadamard Transform (WHT).

To do this, we use SPIRAL, a program generation and optimization 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 with respect to the chosen performance metric (usually runtime). In this paper we use SPIRAL to generate implementations for different frequency configurations and optimize for runtime and physically measured energy consumption. 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 the chosen 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 microarchitecture 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

* This work was supported by DARPA through the Department of Interior grant NBCH1050009 and by NSF through awards 0234293 and 0325687.

K. De Bosschere et al. (Eds.): HiPEAC 2007, LNCS 4367, pp. 201–214, 2007.

different processors, such as AMD's Athlon 64, Intel's XScale (fixed-point processors targeted for embedded applications) and Core processor families.

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

Contribution of this paper. We consider three linear signal transforms: the discrete Fourier transform (DFT), finite impulse response (FIR) filters, and the Walsh-Hadamard transform (WHT). Our test platform is a SITSANG board with an XScale PXA255 fixed-point processor. The platform provides the above mentioned frequency scaling but 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 configuration can consume 5% more energy than the most energy efficient configuration. Third, we apply dynamic scaling (i.e., during execution) and are able to reduce energy consumption; however, this technique does not improve runtime performance. Finally, we show that SPIRAL generated code compares favorably with the hand-tuned Intel's vendor library IPP, which is oblivious to the frequency configuration.

Related work. Optimization for frequency and voltage scaling typically targets large-scale problems and more general codes. Recent work introduces compiler techniques [2], power modeling [3], and software/hardware monitoring of applications [4] to aid adaptation of frequency/voltage settings. [5] and [6] present a compile-time algorithm for the dynamic voltage scaling within an application, inserting switching points chosen by static analysis.

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 [7] and Sparsity [8], for the DFT and related transforms FFTW [9], and for general linear signal transform SPIRAL, which is used in this paper.

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

2 Background

In this section, we first describe the XScale architecture including its reconfigurability features and then the SPIRAL program generation 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 platforms'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. We refer to this platform simply as XScale throughout the paper.

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 recommended or 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 530 MHz CPU speed.

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

For a software developer the problem is at least two-fold. First, different configurations correspond in effect to different platforms and thus code optimized for one configuration may be suboptimal for another. Second, the choice of configuration is not straightforward. For example, if the highest performance is desired, there are three

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

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

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 program generator for linear signal transforms such as the DFT, the WHT, the discrete cosine and sine transforms, FIR filters, 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 can generate 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 in this paper: the DFT, FIR filters, and the WHT. Each transform is a matrix-vector multiplication $y = Mx$, where M is the transform matrix. For example, for input size n, the DFT is defined by the matrix

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

The output of the DFT is also of size n.

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

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

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). Most importantly, the Kronecker, or tensor product, is defined as

$$A \otimes B = [a_{k,\ell} B]_{k,\ell}, \quad \text{for } A = [a_{k,\ell}]_{k,\ell}. \tag{3}$$

If one of the tensor factors A, B is the identity matrix, as in (2), then $y = (A \otimes B)x$ can be implemented simply as a loop. For example, $y = (I_k \otimes B)x$ is a loop with k iterations. In each iteration, B is multiplied to a contiguous chunk of x to yield the corresponding chunk of y. $y = (A \otimes I_m)x$ is a loop with m iterations, but in this case, A is multiplied to subvectors of x extracted at stride m.

The WHT is a real transform defined recursively by $\mathbf{WHT}_2 = \mathbf{DFT}_2$, and

$$\mathbf{WHT}_{2^n} = (\mathbf{WHT}_{2^k} \otimes I_{2^m})(I_{2^k} \otimes \mathbf{WHT}_{2^m}), \quad n = k + m. \tag{4}$$

It only exists for two-power sizes. (4) also serves as algorithm for the WHT, similar to (2).

Algorithms for FIR filters can be described similarly; this includes 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 out of many possible

algorithms 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 standard C 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 Fig. 1 (see [1,14] for a complete description).

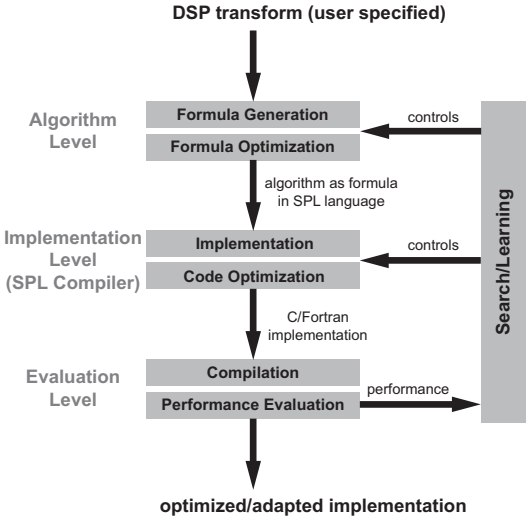

Fig. 1. The program generator SPIRAL

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, for computing a DFT of size 16 three different factorizations of 16 in (2) can be used: 2×8, 4×4, or 8×2. Similar choices apply recursively to the smaller transforms obtained after each decomposition.

3 Extension of SPIRAL

In this work, our goal is to automatically generate implementations of transforms for the XScale platform. These programs are optimized specifically for 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 frequency scaling into SPIRAL's framework. Second, we enabled SPIRAL to run with physically measured energy consumption as performance measure in the feedback loop (see Fig. 1).

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

$$[F]_{497-1/3-1/3}. \tag{5}$$

Next, we extended the SPL compiler (see Fig. 1) to understand these tags and translate them into the appropriate code. In the example (5), 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 WHT in (4). First, the input vector x is multiplied by $(I_{2^k} \otimes \mathbf{WHT}_{2^m})$. As explained after (3), this corresponds to a loop with 2^k iterations. The loop body calls $\mathbf{WHT}_{2^m}$ on contiguous subvectors of x of length 2^m. This access pattern yields good cache utilization and, thus, high performance. It is *compute-bound*.

The second part, $(\mathbf{WHT}_{2^k} \otimes I_{2^m})$ is also a loop, but with 2^m iterations. Further, in the loop body $\mathbf{WHT}_{2^k}$ accesses a 2^k-element long subvector of x, but at stride 2^m. If 2^m is sufficiently large, this is effectively equivalent to reducing the cache size (unless the cache is fully associative), since the elements of x are mapped to the same cache set. The consequence is cache thrashing. The computation 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

$$[(\mathbf{WHT}_{2^k} \otimes I_{2^m})]_{497-1/3-1/3} \cdot [(I_{2^k} \otimes \mathbf{WHT}_{2^m})]_{530-1/4-1/2}. \tag{6}$$

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.

Algorithm. We included an algorithm (see Table 2) for tagging a given formula into SPIRAL. The algorithm in principle applies to WHTs and DFTs but is shown only for WHT for simpler presentation. FIR filters are structured differently; they are compute-bound for all input sizes. The input to the tagging algorithm is the cache size N, 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.

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

Table 2. Algorithm for assigning tags to a formula (example WHT). **Input:** Cache size N; tags c, m for compute-bound and memory-bound sub-formulas, respectively; a formula F for a $\mathbf{WHT}_{2^{k+\ell}}$ of the form $(\mathbf{WHT}_{2^k} \otimes I_{2^\ell})(I_{2^k} \otimes \mathbf{WHT}_{2^\ell})$ where $\mathbf{WHT}_{2^k}, \mathbf{WHT}_{2^\ell}$ are further expanded. **Output:** F tagged.

TagIt(F, N, c, m)
1: **if** $2^{k+\ell} \leq N$ **then**
2: **return** F_c
3: **end if**
4: **if** $2^\ell \leq N$ **then**
5: **return** $[(\mathbf{WHT}_{2^k} \otimes I_{2^\ell})]_m[(I_{2^k} \otimes \mathbf{WHT}_{2^\ell})]_c$
6: **else**
7: **return** $[(\mathbf{WHT}_{2^k} \otimes I_{2^\ell})]_m(I_{2^k} \otimes \mathbf{TagIt}(\mathbf{WHT}_{2^\ell}))$
8: **end if**

3.2 Performance Measurement

We installed SPIRAL on a desktop computer (host machine) and connected the XScale board through the local network. On the host, SPIRAL generates tagged formulas, translates them into fixed-point code, cross-compiles for the XScale, and builds a loadable kernel module (LKM). 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 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 above 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 $E = UIt$, where $U = 3.5V$. 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 generate the programs 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 generated

[2] We use an Agilent 34401A.

programs and and we used *crosstool* to build the cross compiler. In the following figures, we show performance for seven out of the thirteen configurations in Table 1.

The performance is reported in pseudo Mop/s (million operations per second). We exclude from the operations count the index computations and, for input size n, 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 performance is reported in pseudo Mop/J (million operations per Joule). Both metrics (runtime performance and energy efficiency) preserve the runtime and energy relation, respectively.

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

4.1 Runtime Performance Results

General behavior. We achieve the best performance for the DFT (Fig. 2(a)) and WHT (Fig. 2(b)) for problems fitting in the cache. For larger sizes, ther performance drops. This is a property of these transforms as the structure of their algorithms produces strided memory access and hence cache thrashing (see also the discussion in Section 3.1).

For FIR filters (Figs. 2(c) and (d)), in contrast, the performance remains roughly constant across sizes due to the consecutive access of the input.

Best configuration. For the DFT and FIR filters, there is only one best configuration independently of the problem size, namely 530-1/4-1/2 (highest bus speed) for the DFT and 597-1/6-1/6 (highest CPU speed) for FIR filters. This again shows that FIR filters are compute-bound, whereas DFT and WHT are memory-bound. Note that the configuration 597-1/6-1/6 performs poorly with both DFT and WHT.

For the WHT, the best configuration depends on the problem size, namely, whether or not the problem fits into cache. For in-cache sizes, 530-1/4-1/2 is best, for out-of-cache sizes 497-1/3-1/3 is best. The difference, however, is less than 10%.

SPIRAL vs. IPP. In Fig. 4(a), we show the performance of IPP's DFT for different configurations. The relative speed of SPIRAL generated DFT over IPP is shown in Fig. 4(b). Here, IPP is the base line constant to zero and the performance improvement (in percent) of SPIRAL over IPP is shown. For problem sizes $n = 64$, 128, and 4096, SPIRAL generated code is faster; in contrast, it is slower for sizes $n = 256, \ldots, 2048$.

Further, the relative speed of SPIRAL over IPP may vary by 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 generated code for 8-tap FIR filters (Fig. 2(c)) outperforms the respective IPP routines (Fig. 3(a)) by a factor of two. In the case of 16-tap FIR filters, SPIRAL (Fig. 2(d)) and IPP (Fig. 3(b)) have roughly equal performance.

IPP does not provide a WHT library function.

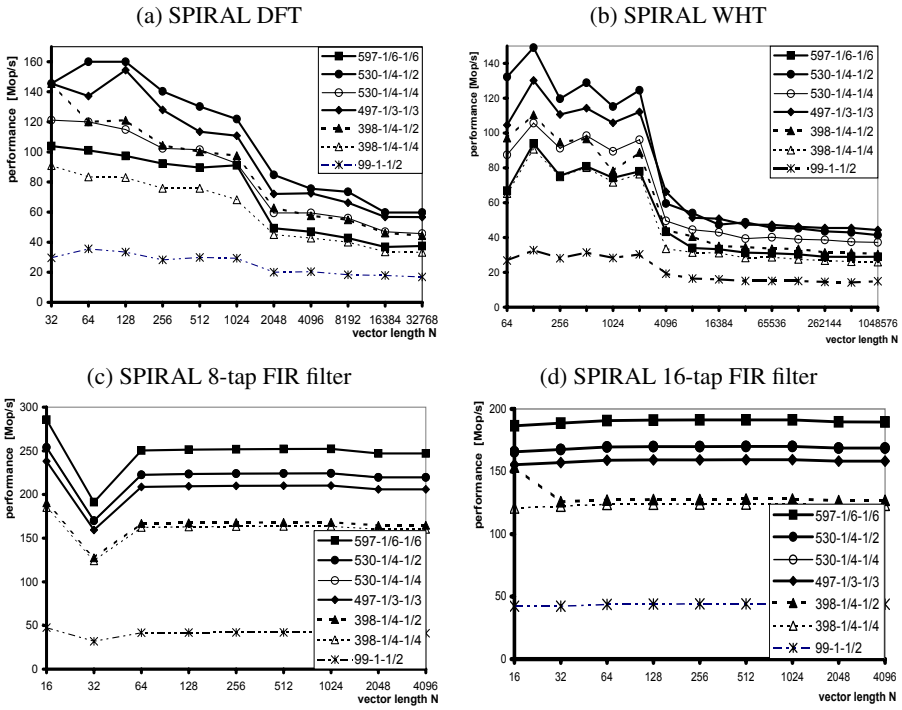

Fig. 2. Performance (in pseudo Mop/s) of SPIRAL generated code: (a) DFT, (b) WHT, (c) 8-tap FIR filter, and (d) 16-tap FIR filter

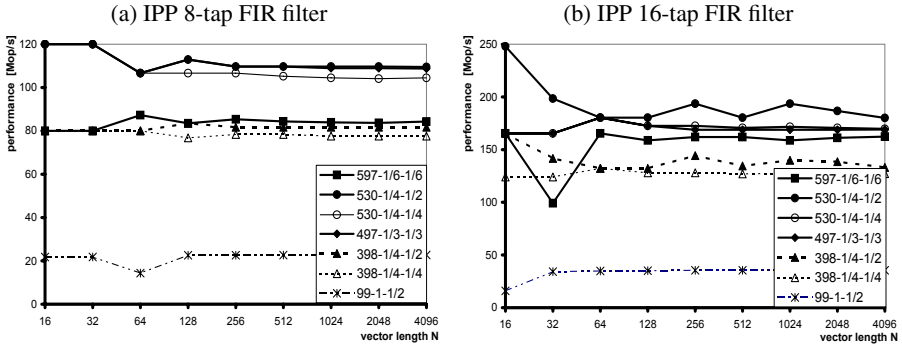

Fig. 3. Performance (in pseudo Mop/s) of IPP FIR filters: (a) 8-tap FIR filter, and (b) 16-tap FIR filter

4.2 Energy Results

General behavior. The energy efficiency of FIR filters (for both SPIRAL and IPP) does not depend on the problem size (see Figs. 5(c) and (d) and Fig. 6). In contrast, the

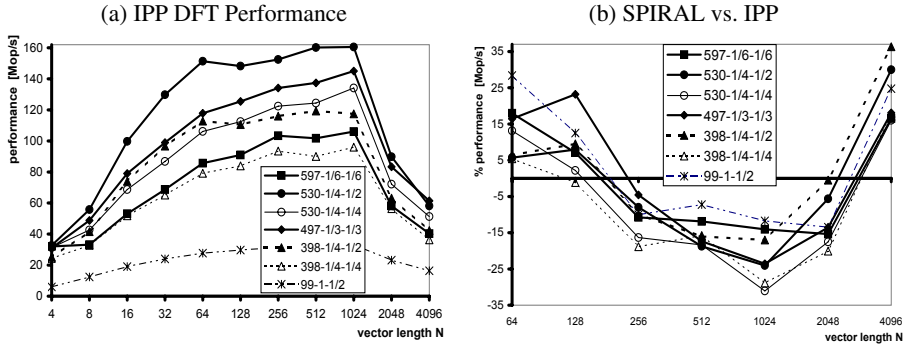

Fig. 4. (a) DFT performance of Intel's IPP (in pseudo Mop/s); (b) Relative performance of SPIRAL generated DFT over IPP's DFT. The percentage is the performance improvement of SPIRAL over IPP.

energy efficiency of the DFT (Fig. 5(a) and Fig. 7(b)) and of the WHT (Fig. 5(b)) is high for in-cache problem sizes and significantly lower for large problem sizes.

Best configuration. For SPIRAL generated DFT code, the configuration with the highest energy efficiency (Fig. 5(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 generated WHT code (Fig. 5(b)) the best configuration in-cache is 530-1/4-1/4 and out-of-cache 398-1/4-1/2. Note that these configuration are different from the ones optimal for performance (see Section 4.1).

SPIRAL vs. IPP. In Fig. 7(a) we show the energy efficiency of IPP's DFT code for different configurations. The relative efficiency of SPIRAL generated code over IPP code is shown in Fig. 7(b). Qualitatively, the plot is similar to Fig. 4(b).

SPIRAL generated 8-tap FIR filter code Fig. 5(c) gains threefold over IPP (Fig. 6(a)) in energy efficiency. For 16-tap FIR filters, SPIRAL still gains about 20% (Fig. 5(d) versus Fig. 6(b)) even though the performance is roughly equal (Section 4.1).

4.3 Dynamic Frequency Scaling

Finally, we investigate the potential of dynamic frequency scaling; that is, the dynamic switching among configurations during the computation (see Section 3.1). For this technique to make sense, the best configuration for in-cache and out-of-cache sizes has to differ. This is the case only for the WHT. Both performance (Fig. 2(b)) and energy efficiency (Fig. 5(b)) are candidates for dynamic frequency scaling.

For both metrics we first choose two configurations. Then, for all problem sizes that do not fit into cache ($N \geq 2^{16}$) we apply the tagging algorithm given in Table 2 to the fastest formulae. This way, SPIRAL finds a WHT implementation that switches between the chosen configurations and automatically trades the switching overhead and

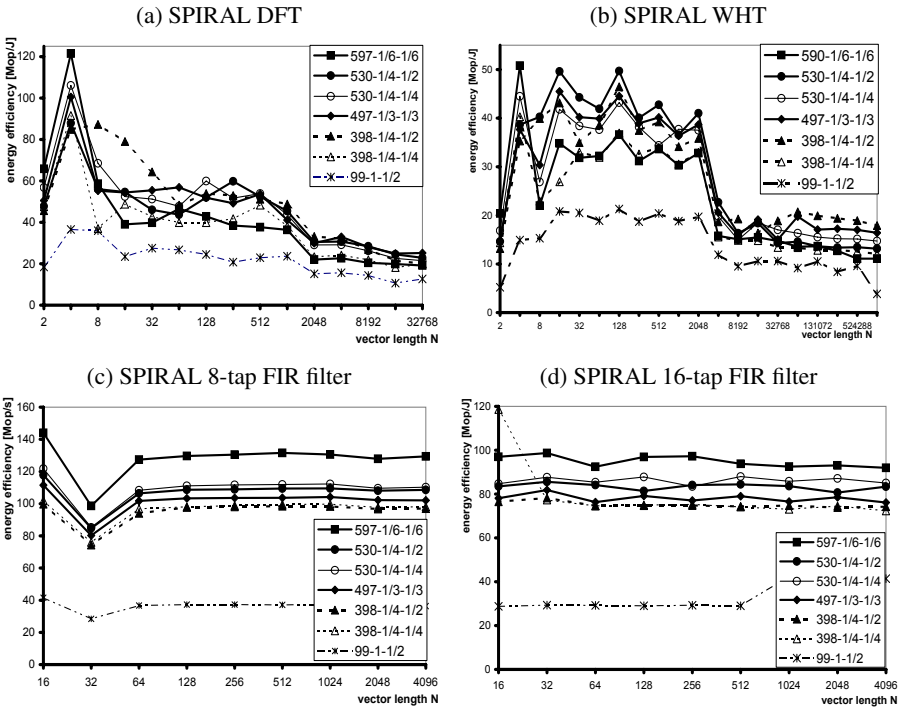

Fig. 5. Energy efficiency (in pseudo Mop/J) of SPIRAL generated code: (a) DFT, (b) WHT, (c) 8-tap FIR filter, and (d) 16-tap FIR filter

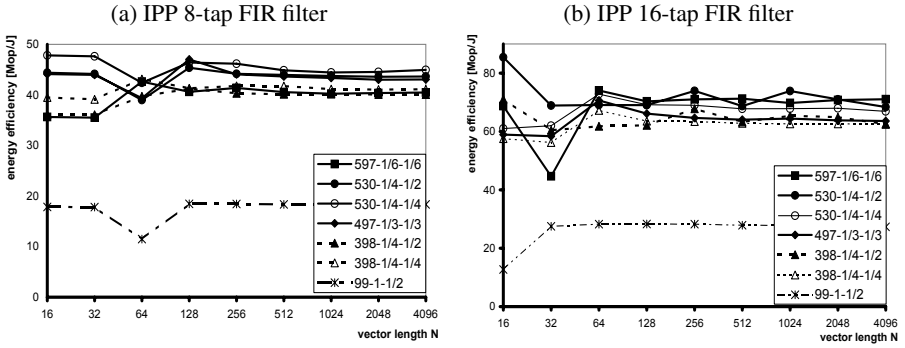

Fig. 6. Energy efficiency (in pseudo Mop/J) of IPP FIR filters: (a) 8-tap FIR filter, and (b) 16-tap FIR filter

the performance or energy efficiency gains obtained by switching, overall optimizing for the given metric.

Runtime performance. Based on Fig. 2(b) we find two configuration candidates to switch between: 1) 530-1/4-1/2 is the fastest configuration for small problem sizes and

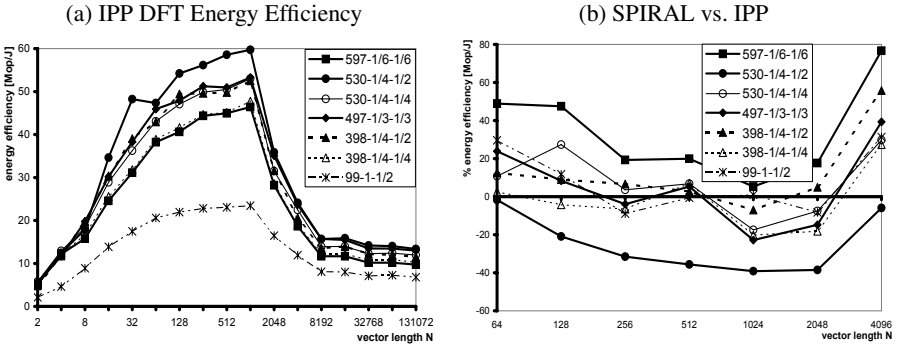

Fig. 7. (a) IPP's DFT energy efficiency (in pseudo Mop/J). (b) Energy efficiency gain or loss of SPIRAL generated DFT over IPP's DFT in percent. Higher means SPIRAL is more efficient.

thus the candidate for CPU bound parts, and 2) 497-1/3-1/3 for memory bound parts of the computation. However, due to the large switching overhead ($530\mu s$) SPIRAL finds that not switching at all leads to the highest performance.

Energy efficiency. Optimizing for energy efficiency leaves more room for the successful application of dynamic frequency scaling, as energy depends both on runtime and power, which in turn both depend nonlinearly on the CPU, bus, and memory frequencies.

Our experiments indicate that switching between 398-1/4-1/2 and 497-1/3-1/3 yields the most energy efficient implementations. Thus, we investigate switching between these configurations further detailing two approaches.

Starting from the statically most efficient configuration 398-1/4-1/2 (line "398-1/4-1/2 static" in Fig. 8) we can gain efficiency by switching to the faster configuration 497-1/3-1/3 for all sub-formulae of shape $\mathbf{WHT}_{2^3} \otimes I_{2^n}$ (line "dynamic 1" in Fig. 8). The efficiency gain is due to higher bandwidth requirements for $\mathbf{WHT}_{2^3} \otimes I_{2^n}$ as this formula trashes the mini cache and at the same time is able to fully utilize the CPU's 8 registers and thus the higher CPU frequency.

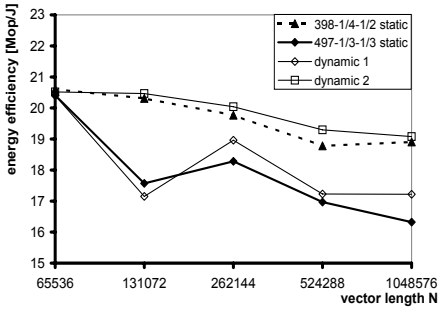

Fig. 8. Dynamic frequency scaling of SPIRAL generated WHT, switching between 497-1/3-1/3 and 398-1/4-1/4

Starting from the statically second most efficient configuration 497-1/3-1/3 (line "497-1/3-1/3 static" in Fig. 8) we can gain efficiency by switching to the slower configuration 398-1/4-1/2 for all sub-formulae of shape $\mathbf{WHT}_{2^2} \otimes I_{2^n}$ (line "dynamic 2" in Fig. 8). In the case of $\mathbf{WHT}_{2^2} \otimes I_{2^n}$ the CPU is not fully utilized and the mini cache is not trashed. Thus, we can slow down the CPU frequency without hurting runtime and can even slow down the memory and bus and still gain energy efficiency. However, this approach (line "dynamic 2" in Fig. 8) cannot compete with the first approach (line "dynamic 1" in Fig. 8).

Overall, dynamic frequency scaling between 398-1/4-1/2 and 497-1/3-1/3 yields a slight gain in energy efficiency with respect to *both* baseline configurations, however, due to different reasons.

5 Conclusions

We show how a program generation framework as SPIRAL can be used to produce efficient DSP kernels such as the DFT, the WHT, and FIR filters on the XScale embedded platform. We support frequency scaling and thus automatically generate and optimize programs tuned for different configurations. Our experiments show that the best configuration depends on the DSP kernel, the metric and sometimes even on the problem size.

References

1. Püschel, M., Moura, J.M.F., Johnson, J., Padua, D., Veloso, M., Singer, B.W., Xiong, J., Franchetti, F., Gačić, A., Voronenko, Y., Chen, K., Johnson, R.W., Rizzolo, N.: SPIRAL: Code generation for DSP transforms. Proc. of the IEEE **93**(2) (2005) 232–275 Special issue on *Program Generation, Optimization, and Adaptation*.
2. Halambi, A., Shrivastava, A., Dutt, N., Nicolau, A.: A customizable compiler framework for embedded systems. In: Proc. Workshop on Software and Compilers for Embedded Systems. (2001)
3. Contreras, G., Martonosi, M.: Power prediction for Intel XScale processors using performance monitoring unit events. In: Proc. International Symposium on Low Power Electronics and Design (ISLPED). (2005) 221–226
4. Singleton, L., Poellabauer, C., Schwan, K.: Monitoring of cache miss rates for accurate dynamic voltage and frequency scaling. In: Proc. Multimedia Computing and Networking Conference. (2005)
5. Hsu, C., Kremer, U.: The design, implementation, and evaluation of a compiler algorithm for CPU energy reduction. In: Proc. Conference on Programming Language Design and Implementation (PLDI). (2003) 38–48
6. Xie, F., Martonosi, M., Malik, S.: Compile-time dynamic voltage scaling settings: Opportunities and limits. In: Proc. Conference on Programming Language Design and Implementation (PLDI). (2003) 49–62
7. Whaley, R.C., Petitet, A., Dongarra, J.J.: Automated empirical optimization of software and the ATLAS project. Parallel Computing **27**(1–2) (2001) 3–35
8. Im, E.J., Yelick, K., Vuduc, R.: Sparsity: Optimization framework for sparse matrix kernels. Int'l J. High Performance Computing Applications **18**(1) (2004)

9. Frigo, M., Johnson, S.G.: The design and implementation of FFTW3. Proc. of the IEEE **93**(2) (2005) 216–231 Special issue on *Program Generation, Optimization, and Adaptation*.
10. Intel: Intel XScale Microarchitecture. (2001)
11. Van Loan, C.: Computational Framework of the Fast Fourier Transform. SIAM (1992)
12. Gačić, A., Püschel, M., Moura, J.M.F.: Fast automatic implementations of FIR filters. In: Proc. International Conference on Acoustics, Speech, and Signal Processing (ICASSP). Volume 2. (2003) 541–544
13. Gačić, A.: Automatic Implementation and Platform Adaptation of Discrete Filtering and Wavelet Algorithms. PhD thesis, Electrical and Computer Engineering, Carnegie Mellon University (2004)
14. Franchetti, F., Voronenko, Y., Püschel, M.: Loop merging for signal transforms. In: Proc. Programming Language Design and Implementation (PLDI). (2005) 315–326