Automated Empirical Tuning of Scientific Codes For Performance and Power Consumption*

Shah Faizur Rahman
Univ. of Texas at San Antonio
6900 N Loop 1604 West
San Antonio, TX 78249
srahman@cs.utsa.edu

Jichi Guo
Univ. of Texas at San Antonio
6900 N Loop 1604 West
San Antonio, TX 78249
jguo@cs.utsa.edu

Qing Yi
Univ. of Texas at San Antonio
6900 N Loop 1604 West
San Antonio, TX 78249
qingyi@cs.utsa.edu

ABSTRACT

Automatic empirical tuning of compiler optimizations has been widely used to achieve portable high performance for scientific applications. However, as power dissipation becomes increasingly important in modern architecture design, few have attempted to empirically tune optimization configurations to reduce the power consumption of applications. We provide an automated empirical tuning framework that can be configured to optimize for both performance and energy efficiency. In particular, we extensively parameterize the configuration of a large number of compiler optimizations, including loop parallelization, blocking, unroll-and-jam, array copying, scalar replacement, strength reduction, and loop unrolling. We then use hardware counters combined with elapsed time to estimate both the performance and the power consumption of differently optimized code to automatically discover desirable configurations for these optimizations. We use a power meter to verify our tuning results on two multi-core computers and show that our approach can effectively achieve a balanced performance and energy efficiency on modern CMP machines.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Optimization*

General Terms

Experimentation, Measurement, Performance

Keywords

Compiler Optimizations, Empirical Tuning, Power consumption, Performance

*This research is funded by the National Science Foundation under Grant No. 0833203 and No. 0747357 and by the Department of Energy under Grant No. DE-SC001770

HiPEAC 2011 Heraklion, Crete, Greece

1. INTRODUCTION

In recent years, empirical tuning [4,8,11,13,17,23] has become a de facto approach adopted by both developers and optimizing compilers to achieve portable high performance for scientific applications on a wide variety of modern computing platforms. However, in spite of the immense success of the tuning approach, existing research has rarely investigated how to effectively reduce the power consumption of software based on empirical feedbacks. As a result, software, especially scientific applications, are rarely made to be energy efficient. While the performance of applications is critically important, their power consumption can often be reduced without sacrificing performance.

Heat accumulation and dissipation are important concerns in the design of modern architectures, from extreme-scale supercomputers, to multi-core desktops, to energy efficient laptops, and to embedded chips in cell phones and MP3s. As heat accumulation is closely related to the power consumption of key architectural components, software should be energy-aware and use the full power of computers only when necessary. Most Operating Systems can be configured to automatically scale down the voltage or frequency of microprocessors when a computer has been idle for a while. However, relatively little support has been made available for developers to empirically optimize the energy efficiency of their applications.

The main challenge in empirically tuning user applications for better energy efficiency is the difficulty in estimating the power consumption of user applications during their evaluation. Previous research has estimated the power consumption of various architectural components using simulation [3, 5, 14], offline profiling [7], and real time monitoring of hardware counters [19]. In particular, architectural simulation has been used to study various compiler/Operating System algorithms to improve power efficiency of user applications [3, 20]. However, simulation cannot be effectively used in empirical tuning because the long simulation time, which imposes a prohibitive extra overhead on top of tuning applications for performance.

This paper presents an empirical tuning framework that can be configured to automatically achieve a balanced performance and power efficiency for user applications. Following the work by Singh, Bhadauria and McKee [19], we use real time feedbacks from hardware counters to estimate the energy efficiency of applications and then use the power consumption feedbacks to tune application performance and energy efficiency collectively. We have used POET [25], an

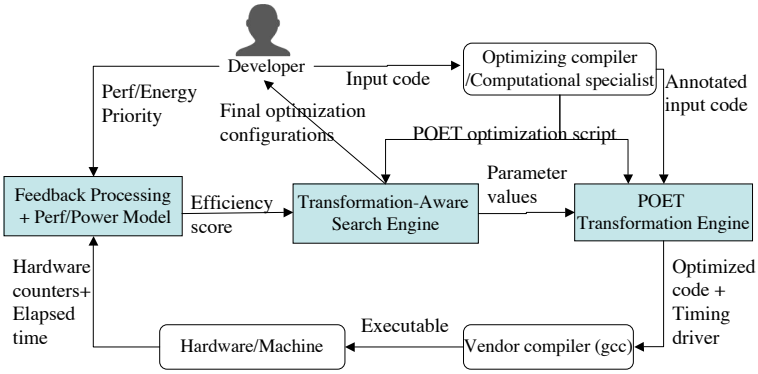


Figure 1: The work flow of our tuning infrastructure

interpreted program transformation language, to extensively parameterize a large collection of compiler optimizations, including OpenMP parallelization, loop blocking, unroll-and-jam, array copying, scalar replacement, strength reduction, and loop unrolling, and have applied these transformations to optimize three matrix computation kernels. We then use our empirical tuning system to automatically discover desirable optimization configurations for these codes, based on both their performance and energy efficiency feedbacks. We use a power meter to verify our tuning results and show that our approach can effectively achieve a balanced performance and energy efficiency on modern CMP machines.

Our research contributions include the following.

- We use hardware counters to effectively guide the automatic empirical tuning of scientific applications to achieve a balanced performance and energy efficiency.
- We demonstrate the impact of a large number of compiler optimizations on both application performance and energy efficiency.
- We present a transformation-aware search algorithm that effectively explores a large optimization space to find desirable optimization configurations.

The rest of the paper is organized as follows. Section 2 introduces our tuning infrastructure. Section 3 presents our search algorithm for exploring the optimization space. Section 4 presents our formulation of application efficiency from empirical feedbacks. Section 5 presents our experimental results. Section 6 presents related work. Section 7 presents our conclusions.

2. TUNING INFRASTRUCTURE

The workflow of our tuning framework is shown in Figure 1 and includes three main components: the POET transformation engine, the transformation-aware search engine, and the feedback processing engine. We use an optimization script, written in an interpreted program transformation language, POET [25], to support the parameterization and empirical tuning of compiler optimizations. The POET optimization script can be automatically generated by an optimizing compiler or manually written by a computational specialist. For this paper, we have used an optimizing compiler to fully automate this process. The script is interpreted by the *POET transformation engine* to optimize the input code based on different parameter configurations supplied

```
void dgemvT(const int M,const int N,const double alpha,
const double *A,const int lda,const double *X,const int
incX, const double beta,double *Y,const int incY)
{
  int i,j;
/*@; BEGIN(nest1=Nest) @*/
  for (i = 0; i < M; i += 1) {
    Y[i] = beta * Y[i];
/*@; BEGIN(nest2=Nest) @*/
    for (j = 0; j < N; j += 1) {
      Y[i] += A[i * lda + j] * X[j];
    }
  }
}
```

Figure 2: Annotated input code for gemv

by the *transformation-aware search engine*, which invokes the *feedback processing engine* internally to evaluate the efficiency of the optimized code based on empirical feedbacks.

To use our framework, a developer needs to provide two inputs: the source code to be optimized and the priority of application performance vs. energy efficiency. An optimizing compiler or a computational specialist can analyze the source code and output the optimization decisions by producing an annotated input code together with a parameterized POET optimization script. An example input code with POET annotations is shown in Figure 2, and a skeleton of the POET script to optimize this code is shown in Figure 3. The TA-aware search engine automatically explores different optimization configurations of the POET script based on efficiency scores returned by the feedback processing engine after invoking the POET transformation engine to optimize the annotated input code with each chosen configuration. Eventually, a final optimization configuration with the best performance/energy efficiency discovered, together with the POET optimization script and the annotated input code, is returned as result to the developer.

The POET language and transformation engine.

POET is an interpreted program transformation language designed for parameterizing general-purpose compiler optimizations for auto-tuning [25]. The POET transformation engine in Figure 1 is essentially a POET language interpreter coupled with a set of transformation libraries and input/output language syntax descriptions. In particular, the POET *opt* library (included at line 1 of Figure 3) contains a large collection of predefined routines which can be invoked to apply a wide variety of source-to-source performance optimizations. To optimize an input program, a POET script needs to be given which specifies exactly which input files

```
1: include opt.pi
2: <trace target/>
3: <input to=target syntax="Cfront.code" from=("rose_dgemvT.C")/>
......
4: <parameter pthread_nest1 type=1.._ default=1
  message="number of threads to parallelize loop nest1"/>
5: <parameter psize_nest1 type=1.._ default=256
  message="number of iterations to run by each thread for nest1"/>
6: <parameter bsize_nest1 type=(INT INT) default=(1 16)
  message="Blocking factor for loop nest nest1"/>
......
7: <eval par_nest1 = DELAY{  <*OMP parallelization*>
  if (pthread_nest1!=1)  { ..... }} />
8: <eval block_nest1 = DELAY{ <*cache blocking *>
  if (bsize_nest1 != (1 1) { ..... }}/>
.......
9: <eval APPLY{par_nest1};
      APPLY{block_nest1};
      APPLY{copy1_nest1};
      APPLY{copy2_nest1};
      APPLY{unrolljam_nest1};
      APPLY{scalar1_nest1};
      APPLY{scalar2_nest1};
      APPLY{unroll_nest2};
      APPLY{cleanup_nest1}/>
10: <output from=(target) syntax=("Cfront.code")/>
```

Figure 3: POET script to optimize Figure 2

to parse using which language syntax descriptions (lines 2-3 of Figure 3), what transformations to apply to the input code after parsing (line 9 of Figure 3), and details of how to apply each transformation (lines 7-9 of Figure 3). Each POET script can be extensively parameterized, where values for the parameters can be flexibly reconfigured via command-line options when invoking the POET transformation engine. In Figure 3, these parameters are declared at lines 4-6 and used to control configurations of optimizations at lines 7-9. The optimized code is then unparsed to standard output at line 10.

The transformation-aware search engine.

The goal of the search engine is to automatically determine values of the optimization parameters for each POET script. It takes a single input, the POET script to tune, and orchestrates the whole tuning process by iteratively determining what parameter values to use to properly configure each POET optimization, invoking the POET transformation engine with the parameter values, compiling the POET optimized source code using a vendor compiler (e.g., gcc), running the compiled code on the hardware machine, and invoking the feedback processing engine to evaluate the empirical feedbacks. In contrast to other generic search engines that look to find the maximal/minimal points in a multi-dimensional generic space, our search engine is domain-specific in that it can read the parameter declarations of each POET script, understand the meaning and context of each optimization parameter, and use built-in heuristics to efficiently explore the configuration space of relevant POET parameters. Details of the search algorithm are presented in Section 3.

The feedback processing engine.

The goal of the feedback engine is to evaluate the efficiency of each POET optimized code. It takes two inputs: the priority preference from the user in terms of how much weight should be put on performance and power consumption respectively, and the hardware counter values and elapsed time statistics collected while evaluating the optimized code. It uses an internal model to estimate the power consumption

Input:
$tuneParams$: tuning parameters declared in POET script;
Output:
$config\_res$: a set of configurations found for $tuneParams$;
Algorithm:
Step1: /* Initialization. */
  Set cur_config = new_configuration($tuneParams$);
  For each parameter $p \in tuneParams$;
    set $cur\_config(p)$ = default_value(p);
  Group $tuneParams$ by the loop nests they optimize;
  Set $opts$ = {loop_blocking, unroll&jam, scalar_repl,
    array_copy, inner_unroll, parallelization};
  Set cur_opt = first_entry(opts);
  Set $config\_res = \{cur\_config\}$;
Step2: /* Set up the current tuning space. */
  Set $cur\_tune = \emptyset$;
  For each $config \in config\_res$:
    For each loop nest L being optimized:
      $cur\_tune = cur\_tune \cup$
        $\{gen\_tune\_space(tuneParams(L), cur\_opt, config)\}$;
  Set $cur\_config = gen\_first\_config(cur\_tune)$;
Step3: /*Apply and evaluate optimization configuration*/
  Invoke POET transformation engine with cur_config;
  Verify the correctness of optimized code;
  Evaluate the optimized code on hardware machine;
  cur_score = efficiency score returned by feedback engine;
Step4: /*Modify $config\_res$ if necessary */
  If cur_score is better or close to those in $config\_res$:
    $config\_res = config\_res \cup \{(cur\_config, cur\_score)\}$;
  If cur_score is better than those in $config\_res$:
    Eliminate weak configurations from $config\_res$;
Step5: /* try the next configuration of cur_tune */
  Set cur_config =
    gen_next_config($cur\_config, cur\_score, cur\_tune$);
  If $cur\_config \neq null$: go to Step3.
Step6: /* try to tune the next optimization*/
  Set cur_opt = next_entry($cur\_opt, opts$);
  If $cur\_opt \neq null$: go to Sep 2;
Step7: /* return result */
  return $config\_res$;

Figure 4: The search algorithm

of the input code based on hardware counter values, combines the estimated energy efficiency with the measured execution time of the optimized code, and then determines an overall efficiency score for the optimized code (for more details, see Section 4). The score is then returned to the transformation-ware search engine and used to determine what other optimization configurations to try next. The whole process repeats until the search algorithm, discussed in Section 3, terminates and returns a final set of desirable optimization configurations.

3. EXPLORING THE OPTIMIZATION SPACE

Our POET transformation scripts currently support the following optimizations, each extensively parameterized so that their configurations can be empirically tuned for better performance or energy efficiency.

- Loop parallelization via OpenMP, where blocks of iterations of an outermost loop are allocated to different threads to evaluate. The optimization is parameterized by the number of threads to run in parallel and the size of each iteration block;
- Loop blocking for cache locality, where iterations of a loop nest are partitioned into smaller blocks so that data can be reused within each block. The blocking factor for each loop dimension is parameterized by the POET script.
- Loop unroll-and-jam, where selected outer loops are unrolled by a small number of iterations, and the unrolled iterations are jammed inside the innermost loop

to promote register reuse. It is parameterized by the number of loop iterations unrolled for each outer loop.

- Array copying and strength reduction, where selected arrays accessed within a blocked loop nest are copied into a separate buffer to avoid cache conflict misses, and strength reduction is applied to reduce the cost of the array address calculations. For each array that can be copied, the optimization is parameterized with a three-way switch which can be used to turn off array copying, strength reduction, or both.
- Scalar replacement combined with strength reduction, where array references are replaced with scalar variables when possible to promote register reuse. The configuration of scalar replacement is similar to array copying.
- Loop unrolling, where an innermost loop is unrolled by a number of iterations to create a larger loop body. The optimization is parameterized by the loop unrolling factor.

The Overall Optimization Space.

For each loop nest in the input code, the overall optimization space is a 7-15 dimensional space (depending on the number of loops and arrays being optimized) where each dimension interacts with others in unpredictable ways. Some optimizations have an inherently large parameterization space, e.g., each loop blocking factor could go from 1 to 512 (depending on the cache size).

It is impractical to exhaustively enumerate the whole optimization space, which may take weeks for a single small kernel. Instead, we use a transformation-aware search algorithm, shown in Figure 4, to effectively explore a subspace of optimization configurations that are likely to produce the best performance or energy efficiency. We adopted the algorithm in Figure 4 because it systematically examines different configurations for each optimization one after another and thus allows us to separately study the impacts of each optimization on the performance and energy efficiency of applications. Our search algorithm also uses systematic space exploration to avoid being stuck at any local minima/maxima, which is a know problem for other generic search algorithms such as hill climbing [17].

The Search Algorithm.

Figure 4 shows the algorithm (implemented using Perl) used by our transformation-aware search engine. The algorithm takes a POET script that has been written to optimize an input code, extracts the tuning parameters from the POET script, and then proceeds to empirically determine the desirable configurations for these parameters. The algorithm output is a collection of tuning parameter configurations that have shown the best efficiency.

Our search algorithm in Figure 4 assumes full knowledge of how each POET parameter is used to control optimizations of the input code and tunes their configuration parameters in a deliberate fashion. In particular, step 1 initializes each optimization parameter with a default value given by the POET script. All the parameters are then grouped by the loop nests that they optimize, and optimizations for the same loop nest are tuned one after another in a predetermined order. In Figure 4, variable *opts* contains the ordering of tuning different optimizations. The ordering is based on the following strategies, and its effectiveness is evaluated in Section 5.5.

- Since parallelization determines the overall size of data operated by each thread, OpenMP parallelization is tuned before all the other sequential optimizations.
- Architectural sensitive optimizations are tuned before predictable optimizations. For example, scalar replacement and loop unrolling are almost always beneficial, while array copying (due to its high overhead) is rarely beneficial. These optimizations are tuned later than other optimizations that have uncertain impacts, e.g., loop blocking and unroll&jam.
- Optimizations that have the most impact are tuned early. For example, loop blocking critically determines whether data can be reused in the cache. Since optimization configurations for out-of-cache computations tend to be significantly different from those for in-cache computations, loop blocking is tuned immediately after tuning parallelization.

Following the predetermined tuning order, the algorithm in Figure 4 tunes each optimization one after another (iterating steps 2-6), keeping a number of best configurations after each tuning phase. In particular, after each optimization is tuned, the algorithm dynamically adjusts the collection of best configurations so far and use the new collection of configurations ($config\_res$) to tune later optimizations. In Figure 4, variable *cur_opt* keeps track of the current optimization being tuned, and $config\_res$ keeps track of the group of best optimization configurations found so far. Step 2 of the algorithm generates a new tuning space (*cur_tune*) by expanding each item in $config\_res$ with a set of new configurations to tune for *cur_opt*. Step 3 invokes the POET transformation engine with each new optimization configuration in *cur_tune* and then collects empirical feedbacks from running the POET optimized code. Step 4 modifies $config\_res$, the set of desirable optimization configurations, based on efficiency feedbacks of running each configuration in *cur_tune*. Step 5 ensures all necessary configurations in *cur_tune* are experimented. Step 6 ensures that all optimizations have been tuned. Finally, Step 7 returns $config\_res$ as the output of the algorithm. Note that both steps 5 and 6 can skip optimization configurations that are known to have a negative impact based on previous experiments.

Since only a small constant number of best configurations are selected after tuning each optimization, the overall tuning time is proportional to the sum of tuning each optimization independently. The algorithm is therefore fairly efficient and requires only a small number of iterations to terminate. Our algorithm is not optimal as it does not fully capture the complex interactions between different optimizations. However, it is sufficient to capture competitive configurations in most cases, as confirmed by our experimental results in Section 5.5.

4. ESTIMATING RUNTIME EFFICIENCY

Figure 5 shows the formula that our feedback processing engine uses to rank the overall efficiency of each optimized code. In particular, it has the user specify how much priority to give execution time and power consumption respectively. After collecting the runtime statistics, the execution time and power consumption rate are then compared against

User input:
  perf_wt: Importance of execution time;
  power_wt: Importance of power consumption;

Runtime statistics of differently optimized code:
  base_time: execution time of the baseline code;
  cur_time: execution time of the current code;
  base_power: power consumed/second by the baseline code;
  cur_power: power consumed/second by the current code;

Efficiency Ranking Formula:
  $perf = base\_time/cur\_time$
  $energy = base\_power/cur\_power$
  $overall\_score = perf * perf\_wt + energy * power\_wt$

Figure 5: Ranking the overall efficiency

Machine-specific statistics:
  base_power: idle state power consumption of the machine;
  a_j (j=1..4): linear weights of different hardware counters;

Runtime statistics of differently optimized code:
  $C(i, j)$: hardware counter j on core i/collecting duration;

Power Estimation Formula:
  $Power(i) = a0 + \Sigma_{j=1,4}(a_j * f(C(i, j)))$
  $total\_power = base\_power + \Sigma_{i=1,4}(5 - i)/4 * Power(i)$

Figure 6: Estimating power consumption for a quad-core AMD Opteron

those of a base line execution (the first optimization configuration evaluated by our search algorithm). The relative execution speedup and energy reduction are then weighted accordingly to compute the final efficiency score.

Following the work by Singh, Bhadauria and McKee [19], we use hardware counter values collected in real time to estimate the power consumed by each processing core of a chip-multiprocessor machine. We used a Watts Up PRO power meter to measure the actual power consumption of the whole system while collecting hardware counter values during the execution of a number of different benchmarks. The whole machine is free of other tasks while running the benchmarks, so the power meter measurement equals to the power required to run each benchmark plus the additional power consumed by the machine to stay idle, which we measured separately using the power meter. We do not distinguish the power consumed by different hardware components and instead correlate hardware counter values with the overall power consumption of the whole system using Spearman's rank correlation algorithm [12].

Figure 6 shows the formula we use to estimate the power consumption rate of a quad-core AMD Opteron machine, where *total_power* is the estimated power consumed by the entire machine, Power(i) ($1 \leq i \leq 4$) is the per-core power consumption estimated for the ith core, and $f(C(i, j))$ is a function (to be determined via regression analysis) of the value of hardware counter j on core i. Our construction of the power consumption model for the quad-core AMD Opteron includes the following three steps.

Step 1: Selection of hardware events to monitor.

On the AMD Opteron, users can select 4 out of 36 different hardware events to monitor simultaneously during the evaluation of user applications. To select which events to monitor, we divide all the events into 4 categories: memory activities, instructions retired, pipeline stalls, and FP (floating point) operations. We then use Spearman's rank correlation algorithm [12] to find a relation between each hardware event and the actual power consumption of the whole system. We then select a leading hardware event from each grouped category based on the correlation value.

Selected events to monitor:
  Memory : L1_CACHE_MISS // L1 cache miss
  FP : FP_OPS // Floating point operations
  Instr. : BR_TAKEN // Branches taken
  Stall : RES_STALL // Cycles stalled
Idle state power consumption: 118W
Power estimation formula:
  if L1_CACHE_MISS/second $< 10^8$ then
    Power(i) = -64.41391051 - 7.28e-09 * (FP_OPS/second)
      + 5.97 * log(BR_TAKEN/second)
      + 1.13e-08 * (RES_STL/second)
      + 2.015e-06 * (L1_CACHE_MISS/second)
  else
    Power(i) = -21.35126692 - 4.481e-10 * (FP_OPS/second)
      + 0.28 * log(BR_TAKEN/second)
      + 3.3e-09 * (RES_STL/second)
      + 4.117372621 * log(L1_CACHE_MISS/second)
  $total\_power = 118 + \Sigma_{i=1,4} Power(i) * (5 - i)/4$

Figure 7: Power consumption model for a quad-core AMD Opteron

Step 2: Estimation of correlation coefficients.

After selecting which hardware events to monitor, we need to determine how to compute the per-core power consumption from combining the hardware counter values. As shown in Figure 6, we divide the value of each hardware counter by the time duration of the collection and then use a piecewise linear function to combine their values into a power consumption rate. We use multivariate Ordinary Least Square (OLS) regression to determine the linear coefficient and the base function for each hardware event based on the empirical results collected from different benchmarks.

Step 3: Putting things together.

To accommodate sharing of resources by different cores, we multiply each $Power(i)$ (power consumed by core i) with a predetermined weight, (5-i)/4, before adding them together with the idle-state power consumption of the machine, so that as more cores are used, the impact of each additional core is decreased.

Figure 7 shows the resulting power model for our AMD Opteron machine. From this model, we see that the power consumption rate increases when user applications incur frequent L1 cache misses, take a large number of branches, or when the CPU is stalled for many cycles. Further, the power consumption rate decreases when the CPU is busy with floating point operations. Our result is consistent with those by Singh et al. [19] and Contreras et al [7].

5. EXPERIMENTAL RESULTS

The goal of our research is to provide automatic empirical tuning support to achieve a balanced performance and energy efficiency on Chip Multi-Processor machines. The following subsections first discuss our experimental design and then present results to confirm each of the following hypotheses.

- Different configurations of compiler optimizations can significantly change the power consumption rate of user applications.
- Hardware counters can be used to accurately estimate the power consumption of applications and thus effectively guide empirical tuning of their energy efficiency;
- Empirical tuning can be used to effectively achieve a balanced performance and energy efficiency for scientific codes.

- Our TA-search algorithm in Section 3 can effectively find desirable optimization configurations in practice.

5.1 Experimental Design

We have evaluated our framework using three matrix computation kernels: gemm (matrix-matrix multiplication), gemv (matrix-vector multiplication), and ger (vector-vector multiplication). Figure 2 shows the original input code for *gemv*. The input code for *gemm* and *ger* are written similarly. We have selected these benchmarks for two reasons.

- All of them are computationally intensive and are known to benefit from the large collection of source-to-source compiler optimizations supported by our framework.
- The three benchmarks vary widely in their computation/data access ratios. In particular, gemm is compute-bound as it reuses every data item a large number of times during evaluation; gemv is memory bound as only a small fraction of data are reused; ger is severely memory-bound as no data item is reused in the computation. Consequently, these benchmarks are expected to be representative of different behaviors demonstrated by scientific codes.

Our tuning framework is presented in Section 2, and the optimizations applied to each benchmark are discussed in Section 3. We tuned each benchmark using 1000∗1000 matrices. To reduce tuning time, when multiple loops are blocked in a loop nest, we assign the same blocking factor to all loop dimensions. Our search engine selects the top 10 configurations after tuning each optimization, a decision confirmed to be effective for our selection of benchmarks in Section 5.5.

We tuned all benchmarks on two multi-core machines: a quad-core machine running Linux with two dual-core 3 GHz AMD Opteron Processors (128KB L1 cache/core), and an eight-core machine running MacOS with two quad-core 2.66 GHz Intel processors (32KB L1 cache/core). We used PAPI to collect values of hardware counters on the AMD Opteron. Because PAPI does not support MacOS, we were not able to collect hardware counter values on the Intel Mac. Therefore, we used the Mac machine to tune for performance only, and used only the AMD Opteron to tune benchmarks for both performance and energy efficiency. We used a *Watts Up PRO Power Meter* to collect the actual power consumption of both machines while running the benchmarks. Since the power meter can log the power consumption of the whole system once every second, we repetitively invoke each optimized routine for 4-5 seconds to accurately record its power consumption rate.

All benchmarks are compiled with -O2 option using gcc 4.2.4 on the AMD machine and gcc 4.0.1 on the Intel machine. We did not use the -O3 option to prevent gcc from applying aggressive loop optimizations to our already heavily optimized code. Each optimized code is first tested for correctness and then linked with its timing driver, which sets up the execution environment, repetitively invokes the optimized routine a pre-configured number of times (typically 3-5 times), and then reports the minimum elapsed time of executing each routine and the median of hardware counter values across multiple runs.

5.2 Impact of Optimizations on Power Consumption

Figure 8 shows the variation of both performance (MFLOPS) and energy efficiency (actual power consumed per second, measured by the Watts up Pro power meter) when the three matrix benchmarks are tuned with different optimization configurations on the two machines. To highlight the degree of variation, both performance and power consumption are normalized to a base line execution which achieves the median performance. All optimization configurations are sorted by the number of threads they use.

From the figures, we see that high performance generally corresponds a high power consumption rate, but the corresponding variation of power consumption is far smaller than that of the performance. Further, many optimization configurations have a low performance level but a high power consumption rate, while some others achieve high performance combined with low power consumption. It is therefore both possible and desirable to find the optimization configurations that are both performance and energy efficient.

From examining the collected data, we found that loop parallelization has the most impact on power consumption, which steadily increases as more threads are used for most cases except for *gemv* and *ger* on the Intel Mac, where the power consumption rate has stayed mostly constant, shown in Figure 8(b). Loop blocking has the second most impact on power consumption. In particular, most of the spikes in the power consumption curve are results of experimenting with different blocking factors. However, there does not seem to be any clear relation between individual loop blocking factors and their power consumption. This is as expected since the relations between loop blocking factors and cache/TLB misses are known to be complex and unpredictable.

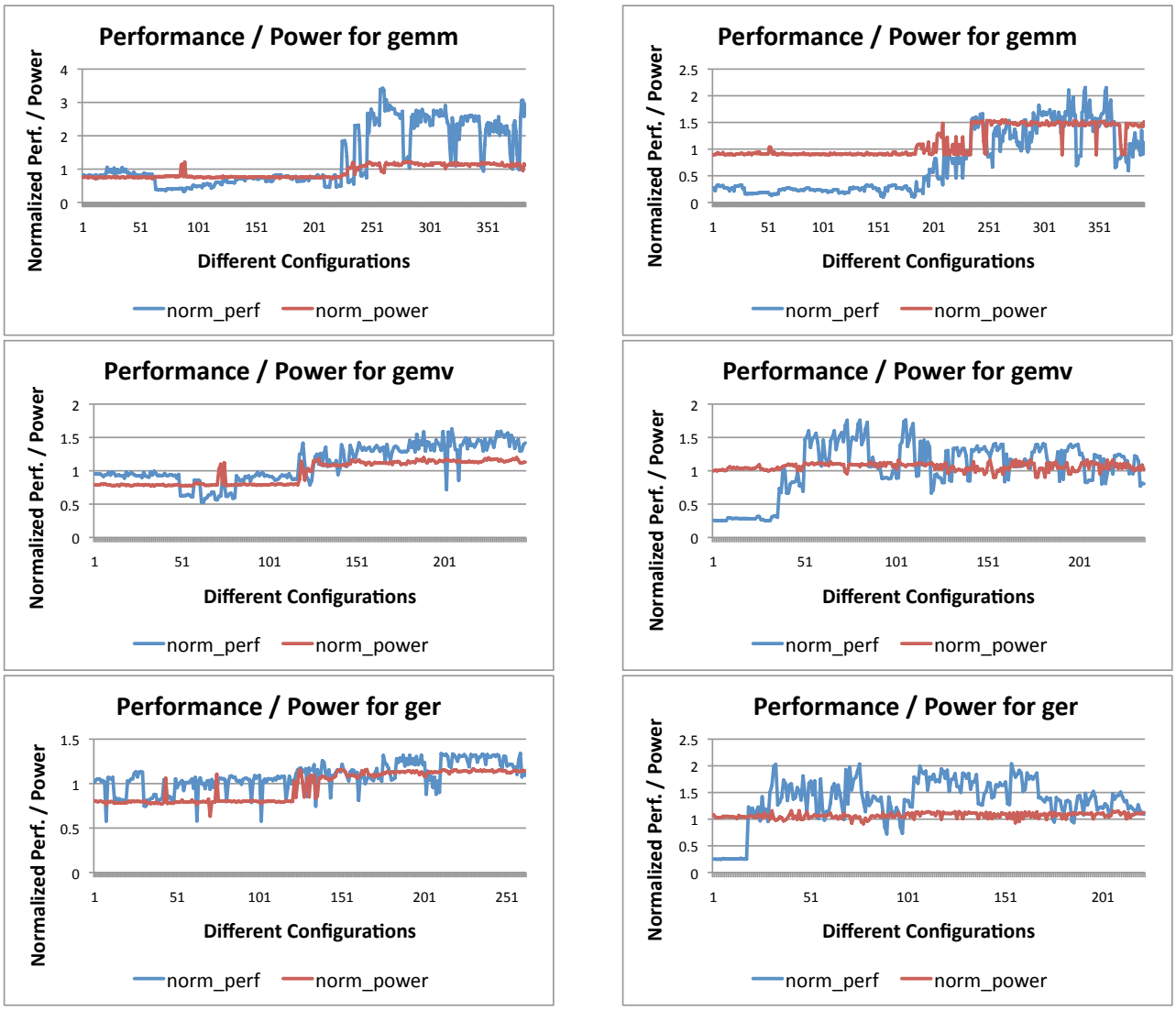
Besides loop parallelization and blocking, loop unrolling is the third optimization that clearly impacts the power consumption rate. Most of the small variations on the power consumption curves are caused by different loop unrolling or blocking factors. In general, as the loop unrolling factor increases, the power consumption goes down. The difference is about 1-2% of the total power consumption. The impact of other optimizations is less significant compared to loop parallelization, blocking, and unrolling.

In summary, to significantly reduce power consumption, a scientific application can use fewer number of threads and then fine-tune the cache blocking and loop unrolling factors to ensure that both the CPUs and the memory hierarchy are being used in an efficient manner. The other optimizations can be tuned for performance only without significantly impacting the power consumption of the hardware.

5.3 Modeling Power Consumption

The model we built to estimate the power consumption rate of the quad-core AMD Opteron machine from hardware counter values is shown in Figure 7. To verify the accuracy of this model, we ran the three matrix benchmarks with different optimization configurations, estimate the power consumption of each optimized code based on the model in Figure 7, and then compare the estimated power with the actual power consumption of the machine measured using a *Watts Up Pro Power Meter* while runing each optimized code.

Figure 9 shows the result of comparing our estimated power consumption rate with the actual system power consumed by the AMD Opteron machine. All graphs show that the estimated power consumption rate closely matches that actually consumed, except for some occasional big spikes.



(a) On the AMD Opteron

(b) On the Intel Mac

* Both performance and power are normalized to those of the execution that achieves the median performance

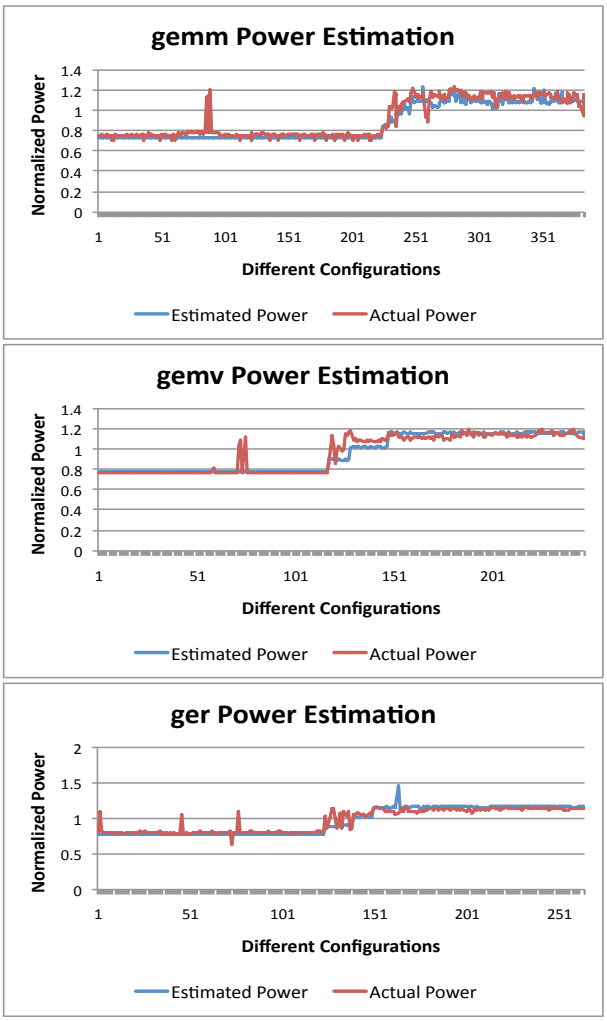
Figure 8: Correlation between performance and power consumption

Most of these spikes are the result of experimenting with different loop blocking factors. It is well-known that different loop blocking factors can have hard-to-predict impact on cache/TLB misses. Since our model can use at most four hardware counters simultaneously to estimate power in real time, it likely failed to capture these spikes due to lack of information on L2 cache and TLB misses. Increasing the number of hardware events to monitor will likely increase the accuracy of our power model, as done by Contreras and Martonosi [7] through offline profiling.

5.4 Tuning For Both Power and Performance

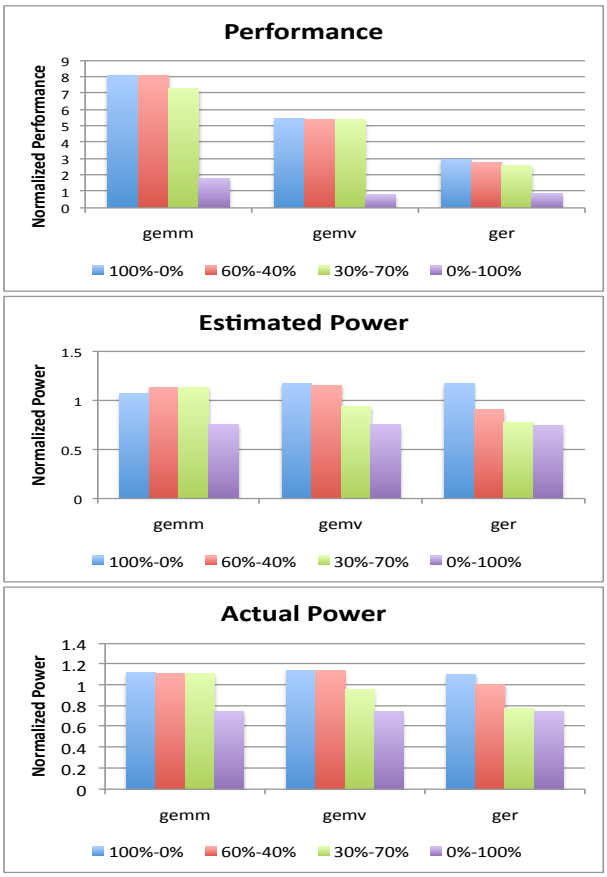
Figure 10 shows the best performance, estimated power consumption rate, and actual power consumption rate achieved when using our power estimation model to tune the three matrix benchmarks using different priorities for performance and power. In particular, each benchmark is tuned by allocating 100%, 60%, 30%, and 0% priority to performance (and thus 0%, 40%, 70%, and 100% priority to power) respectively. Note that when 0% priority is assigned to performance, the benchmarks are tuned solely for reducing power consumption. Both the measured performance and the power consumption rates are normalized to those of the default configuration chosen by the POET script (note that the baseline here is different from the ones used in Figure 8). The goal is to demonstrate that by integrating power consumption as a criteria in auto-tuning, we can find optimizations that achieve a balanced performance/energy efficiency.

From the figures, we see that the power consumption rates of scientific codes can clearly be reduced without sacrificing performance. For both *gemv* and *ger*, when allocating 30% to performance and 70% priority to power, the power consumption rate is reduced by 20-30% with minimal performance degradation. Note that since the variation of performance is significantly larger than that of power consumption, a high priority must be allocated to power consumption



* All power consumptions are normalized to their median consumption rates to allow easy viewing of variations

Figure 9: Accuracy of power estimation on the AMD Opteron



* Both performance and power are normalized to those of the default configuration chosen by the POET script

Figure 10: Tuning for both power and performance with different priorities on the AMD Opteron

in order for it to make a significant impact on the tuning result. For *gemm*, 70% priority on power has not resulted in any power reduction in spite of the performance degradation caused by random factors in the search algorithm.

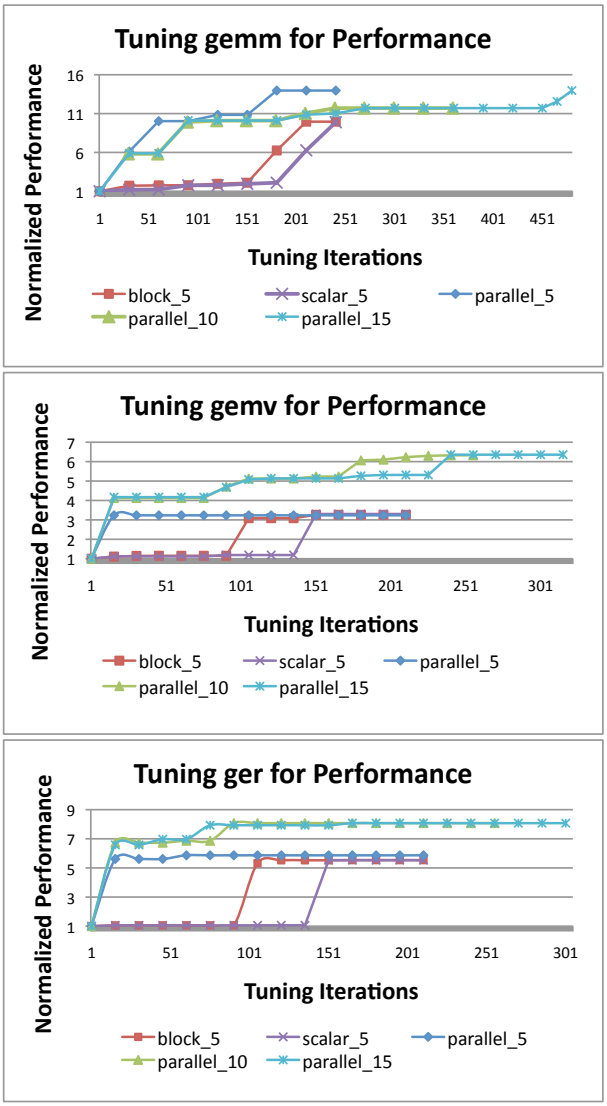
Our search algorithm in Section 3 tunes each optimization separately from the others and thus allows us to more easily study the impact of each optimization. However, it does not sufficiently consider interactions between different optimizations and can lead to suboptimal solutions due to early undesirable decisions. The flaws in the search algorithm can limit the effectiveness of collectively tuning for both performance and power consumption, as demonstrated by the less-than-ideal tuning result for *gemm* in Figure 10. Our tuning approach, however, is independent of specific search algorithms and can easily collaborate other existing alternative generic search algorithms [6, 13, 21].

5.5 Exploring The Search Space

To further evaluate the effectiveness of our search algorithm in Section 3, we experiment with tuning optimizations in different orders to determine its sensitivity to the search starting point. Further, we vary the number of top optimization configurations dynamically selected after tuning each optimization to determine whether significant performance improvement can be achieved by increasing the number of different configurations being experimented.

Figure 11 shows the best performance achieved at different iterations of the search algorithm when tuning the three matrix benchmarks on the eight-core Intel Mac. For each benchmark, the performance (MFLOPS) is normalized to that of a baseline execution of the default optimization configuration given by each POET script. Three different tuning orders, *block_5* (tune loop blocking first), *scalar_5* (tune scalar replacement first), and *parallel_5* (tune loop parallelization first), are experimented while selecting the top 5 configurations after tuning each optimization. Further, *parallel_10* tunes parallelization first while selecting 10 best configurations after tuning each optimization, and *parallel_15* selecting the top 15 configurations.

From Figure 11, the search algorithm is relatively insensitive to the different ordering of optimizations except for *gemm*, where tuning loop parallelization first has resulted in a 50% performance improvement over tuning other optimizations first. The dramatic performance difference is due to the large number of processing cores available on the Intel machine and the relatively small data size being tuned. When operating on $1000 * 1000$ matrices on a single thread,



* Performance is normalized to that of the default configuration chosen by the POET script

Figure 11: Tuning on the Intel Mac using different optimization ordering

the *gemm* routine is memory-bound and loop blocking is necessary to enable data reuse within caches. However, when using 8 threads, each thread only needs to operate on a $128 * 128$ sub-matrice which can fit in cache without any blocking. Therefore the loop blocking factors chosen before parallelization has resulted in suboptimal performance. The performance difference for *gemv* and *ger* is not as significant as neither can significantly benefit from loop blocking due to limited data reuses. When tuning scalar replacement first, the best performance achieved is very similar to that achieved by tuning blocking first, and the performance difference is within 2%.

When the number of top configurations selected increases from 5 to 10 after tuning each optimization, the best performance found for *gemm* remained similar, while the best performance found for *gemv* and *ger* increased significantly. This indicates that selecting top 5 configurations is not sufficient for the algorithm to remain stable. Therefore, we choose 10 as the default number of top configurations to keep after tuning each optimization. When the number of top configurations increase to 15, the best performance found has remained stable, indicating that selecting 10 top configurations is good enough for most cases.

Since our search algorithm tunes different optimizations relatively independently of each other, and it assigns the same blocking factor to all dimensions of a loop blocking optimization, the search terminates fairly quickly. For example, when the top 5, 10, and 15 configurations are selected for *gemm*, the search terminates after about 250, 360, and 500 iterations respectively. Most of our tuning sessions finish in less than 2 hours when selecting 10 top configurations.

6. RELATED WORK

Empirical tuning has been used in the development of many successful scientific libraries, such as ATLAS [23], PHiPAC [4], OSKI [22], FFTW [8], SPIRAL [16], among others, to achieve portable high performance. Recent research has automated empirical tuning with *iterative compilation*, where configurations of compiler optimizations are empirically modified based on performance feedbacks of the optimized code [1, 9, 13, 15, 17]. Existing auto-tuning research has mostly focused on achieving portable high performance in terms of execution time. In contrast, this research focuses on collectively modeling both the performance and the power efficiency of applications.

The POET transformation engine was initially developed by Yi et al [25] to parameterize both domain-specific and general-purpose compiler optimizations for empirical tuning. The language has been used to successfully optimize a number of linear algebra routines and have achieved comparable performance as those achieved by ATLAS [26]. The focus of this paper is to explore the configuration space of a large number of POET-implemented compiler optimizations for both performance and energy efficiency. Instead of using an existing generic search technique [6, 13, 21], we have developed a transformation-aware search engine to explore the optimization space.

Previous work on optimizing the energy efficiency of software applications has mostly focused on different instruction scheduling algorithms in compilers [10,14,18,24] and thread-allocation algorithms in Operating Systems for CMP [2, 3, 19]. Power consumption of software applications has been estimated using architectural simulation [3,5,14], offline application profiling [7], and real time monitoring of hardware counters [19]. Our modeling of software power consumption uses real-time monitoring of hardware counters, following the work by Singh, Bhadauria, and McKee [19]. In contrast to previous work, this paper investigates power consumption as a concern in the empirical tuning of scientific applications and models the impact of a much larger collection of complier optimizations on the energy efficiency of applications.

7. CONCLUSIONS

We present an empirical tuning framework that automatically achieves a balanced performance and energy efficiency for scientific codes on modern multi-core architectures. Our framework has accurately estimated the power consumption of chip multiprocessors in real time and used the information to guide the configuration of a large number of compiler optimizations. We show that a number of compiler opti-

mizations can significantly impact the energy efficiency of applications and that the energy consumption of scientific codes can be reduced without sacrificing their performance.

8. REFERENCES

[1] N. Baradaran, J. Chame, C. Chen, P. Diniz, M. Hall, Y.-J. Lee, B. Liu, and R. Lucas. Eco: An empirical-based compilation and optimization system. In *International Parallel and Distributed Processing Symposium*, 2003.

[2] D. Bautista, J. Sahuquillo, H. Hassan, S. Petit, and J. Duato. A simple power-aware scheduling for multicore systems when running real-time applications. In *Proc. 22nd IEEE/ACM International Parallel and Distributed Processing Symposium*, Apr. 2008.

[3] Y. Ben-Itzhak, I. Cidon, and A. Kolodny. Performance and power aware cmp thread allocation modeling. In *International Conference on High Performance Embedded Architectures and Compilers*, pages 232–246, Jan. 2010.

[4] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel. Optimizing matrix multiply using phipac: a portable, high-performance, ansi c coding methodology. In *Proc. the 11th international conference on Supercomputing*, pages 340–347, New York, NY, USA, 1997. ACM Press.

[5] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *Proc. 27th annual international symposium on Computer architecture*, pages 83–94, New York, NY, USA, 2000. ACM.

[6] C. Chen, J. Chame, and M. Hall. Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy. In *International Symposium on Code Generation and Optimization*, San Jose, CA, USA, March 2005.

[7] G. Contreras and M. Martonosi. Power prediction for intel xscale®processors using performance monitoring unit events. In *International symposium on Low power electronics and design*, pages 221–226, New York, NY, USA, 2005. ACM.

[8] M. Frigo and S. Johnson. FFTW: An Adaptive Software Architecture for the FFT. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, volume 3, page 1381, 1998.

[9] T. Kisuki, P. Knijnenburg, M. O'Boyle, and H. Wijsho. Iterative compilation in program optimization. In *Compilers for Parallel Computers*, pages 35–44, 2000.

[10] C. Lee, J. K. Lee, T. Hwang, and S.-C. Tsai. Compiler optimization on instruction scheduling for low power. In *ISSS '00: Proceedings of the 13th international symposium on System synthesis*, pages 55–60, Washington, DC, USA, 2000. IEEE Computer Society.

[11] J. Moura, J. Johnson, R. Johnson, D. Padua, M. Puschel, and M. Veloso. Spiral: Automatic implementation of signal processing algorithms. In *Proceedings of the Conference on High-Performance Embedded Computing*, MIT Lincoln Laboratories, Boston, MA, 2000.

[12] J. L. Myers and A. D. Well. *Research Design and Statistical Analysis*. Lawrence Erlbaum Associates, Inc., 2003.

[13] Z. Pan and R. Eigenmann. Fast automatic procedure-level performance tuning. In *Proc. Parallel Architectures and Compilation Techniques*, 2006.

[14] A. Parikh, S. K. Kim, M. Vijaykrishnan, and M. J. N. Irwin. Instruction scheduling for low power. *Journal of VLSI Signal Processing Systems For Signal Image And Video Technology*, (1):129–149, 2004.

[15] G. Pike and P. Hilfinger. Better tiling and array contraction for compiling scientific programs. In *SC*, Baltimore, MD, USA, November 2002.

[16] 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. *Proceedings of the IEEE, special issue on Program Generation, Optimization, and Adaptation*, 93(2), 2005.

[17] A. Qasem and K. Kennedy. Profitable loop fusion and tiling using model-driven empirical search. In *Proceedings of the 20th ACM International Conference on SuperComputing (ICS06)*, June 2006.

[18] W.-T. Shiue. Retargetable compilation for low power. In *Proc. the 9th international symposium on Hardware/software codesign*, pages 254–259, New York, NY, USA, 2001. ACM.

[19] K. Singh, M. Bhadauria, and S. A. McKee. Real time power estimation and thread scheduling via performance counters. In *Proc. Workshop on Design, Architecture, and Simulation of Chip Multi-Processors, Como, IT.*, November 2008.

[20] V. Tiwari, S. Malik, A. Wolfe, and M. T.-C. Lee. Instruction level power analysis and optimization of software. *The Journal of VLSI Signal Processing*, 13(2-3):223–238, 1996.

[21] M. J. Voss and R. Eigenmann. High-level adaptive program optimization with adapt. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 93–102, 2001.

[22] R. Vuduc, J. Demmel, and K. Yelick. OSKI: An interface for a self-optimizing library of sparse matrix kernels, 2005. `bebop.cs.berkeley.edu/oski`.

[23] R. C. Whaley, A. Petitet, and J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1):3–25, 2001.

[24] H. Yang, G. R. Gao, and C. Leung. On achieving balanced power consumption in software pipelined loops. In *CASES '02: Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 210–217, New York, NY, USA, 2002. ACM.

[25] Q. Yi, K. Seymour, H. You, R. Vuduc, and D. Quinlan. POET: Parameterized optimizations for empirical tuning. In *Workshop on Performance Optimization for High-Level Languages and Libraries*, Mar 2007.

[26] Q. Yi and C. Whaley. Automated transformation for performance-critical kernels. In *ACM SIGPLAN Symposium on Library-Centric Software Design*, Montreal, Canada, Oct. 2007.