Parameterization and Search-space Exploitation of Loop Fusion

Yuan Zhao* Qing Yi† Ken Kennedy* Dan Quinlan ‡ Richard Vuduc‡

∗ Rice University, Houston, TX, USA
† University of Texas at San Antonio, TX, USA
‡ Lawrence Livermore National Laboratory, Livermore, CA, USA

Abstract. Traditional compilers are limited in their ability to optimize applications for different architectures because statically modeling the effect of specific optimizations on different hardware implementations is difficult. Recent research has been addressing this issue through the use of *empirical tuning*, which uses trial executions to determine the optimization parameters that are most effective on a particular hardware platform. In this paper, we investigate empirical tuning of loop fusion, an important transformation for optimizing a significant class of real-world applications. In spite of its usefulness, fusion has attracted little attention from previous empirical tuning research, partially because it is much harder to configure than transformations like loop blocking and unrolling. This paper presents novel compiler techniques that extend conventional fusion algorithms to parameterize their output when optimizing a computation, thus allowing the compiler to formulate the entire configuration space for loop fusion using a sequence of integer parameters. The compiler can then employ an external empirical search engine to find the optimal operating point within the space of legal fusion configurations and generate the final optimized code using a simple code transformation system. We have implemented our approach within our compiler infrastructure and conducted preliminary experiments using a simple empirical search strategy. Our results convey new insights on the interaction of loop fusion with limited hardware resources, such as available registers, while confirming conventional wisdom about the effectiveness of loop fusion in improving application performance.

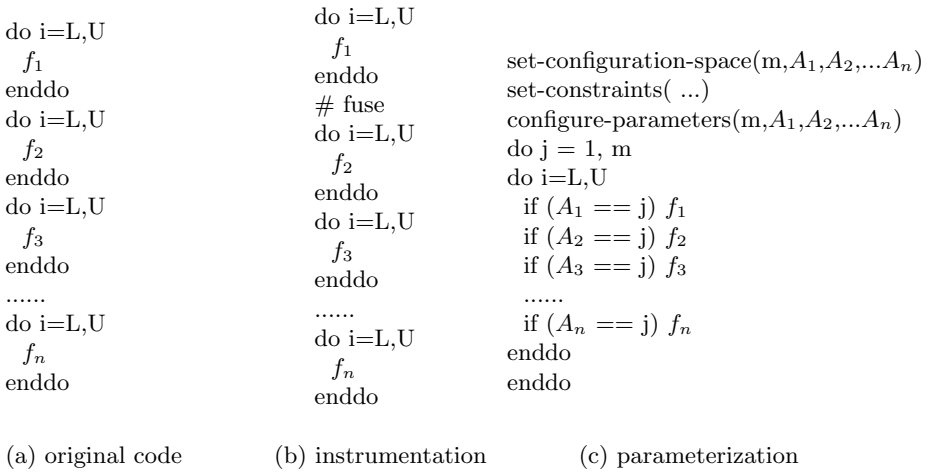
1 Introduction

The evolving technology has driven computer architectures into increasingly complex designs, manifested by a wide range of advanced processors and deep memory hierarchies. Such complex machine architectures pose significant challenge to compilers in estimating the runtime behavior of applications. Mispredictions by compilers often lead to poor application performance and a low percentage utilization of peak computational power.

As an alternative to static modeling of machine behavior, empirical tuning of applications has recently become popular in the research community when applying compiler optimizations. The success of empirical tuning relies on the

availability of an optimization search space. Specifically, compilers must be able to provide a clear mapping between the performance of applications and the configuration of optimizations.

Among the many compiler transformations, loop fusion has been established as critical in optimizing a significant class of real-world applications [16, 11, 15]. However, previous empirical tuning research has mostly focused on loop blocking and unrolling. Little effort has been devoted to tuning loop fusion, partially because the transformation is much more difficult to configure and explicitly parameterize. To illustrate the complexity of this problem, we consider the pseudocode in Figure 1(a).



(a) original code (b) instrumentation (c) parameterization

Fig. 1. Pseudo code to illustrate parameterize of loop fusion

Suppose all the loops in Figure 1(a) can be fused into a single loop without changing the meaning of the original program. But due to register pressure and other factors such as cache conflict misses, it might not be beneficial to fuse all the loops together. As we may not be able to precisely model the runtime environment, we would like to empirically determine the best transformation.

To experiment with different transformations, we must be able to automatically accept different configurations and perform loop fusion accordingly. One straightforward approach is to implement a simple translator that automatically instruments the original program with different annotations. The compiler can then extend conventional loop fusion algorithms to understand the annotations (*e.g.* LoopTool [22]). Figure 1(b) illustrates this approach, where "#fuse" is an annotation inserted by the instrumentation translator to instruct the compiler that f_1 and f_2 should be fused into a single loop. The instrumentation approach could work to some extent but has the following limitations.

– Although the instrumentation translator can recognize loops in the original program and insert annotations between loops, it cannot use information from the compiler, such as the safety and profitability of performing various transformations, to guide the empirical search process.
– Although the instrumentation translator can instruct the compiler to fuse loops that are next to each other in the original code, it cannot exploit fusion transformations that reorder the loops. For example, suppose the best transformation for Figure 1(a) is to fuse loops f_1,f_3 and f_4, but leave f_2 out of the fusion group. Without help from the compiler, the instrumentation translator cannot recognize it is safe to do so and therefore cannot perform the reordering transformation to enable the best annotation.

To overcome the above limitations, we believe that the compiler must be able to pass information to the empirical tuning system. Instead of having the empirical tuning system blindly instructing a compiler what to do, we make the compiler generate all the possible transformations. The tuning system can then choose which versions to use based on profiling information and predictions from the compiler. As it is overly expensive and impractical to explicitly generate all equivalent versions of a program, we must devise a compact way to represent different fusion transformations.

This paper presents a new code generation technique explicitly for this purpose. Our goal is to allow loop fusion to be parameterized just as loop blocking is parameterized with blocking factors. Specifically, we have formulated the configuration space of loop fusion with a sequence of integer parameters. As different values are assigned to each parameter, a parameterized code template can be instantiated to generate the corresponding transformation. Given an arbitrary input code, the parameterized code template, combined with the configuration space of the parameters, can potentially represent the results of applying all possible loop fusion transformations.

Given Figure 1(a) as input, our parameterization of loop fusion transformations is shown in Figure 1(c). Specifically, we separate the sequence of original loops into a collection of groups, where each group is fused into a single loop after applying fusion. The first statement in (c) specifies the parameters of the configuration space, which include the number of fusion groups (m in Figure 1(c)), and a group id ($A_1, A_2, ..., A_n$ in (c)) for each original loop, indicating which fusion group the original loop belongs to. The *set-constraints* statement in (c) specifies a collection of constraints that must be satisfied by the parameters in order to produce a valid transformation. The constraints are passed to an empirical search engine (implemented separately from the compiler). We then invoke the *configure-parameters* function in (c), which asks the search engine to determine the values of configuration parameters. Finally, we use the code template in (c) to generate the result of the corresponding loop fusion transformation. The parameterized code in Figure 1(c) can be used to represent all possible fusion transformations of the original code. Section 2 describes this formulation in more detail.

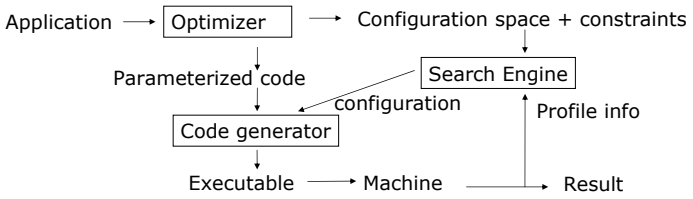


Fig. 2. Empirical tuning of loop fusion

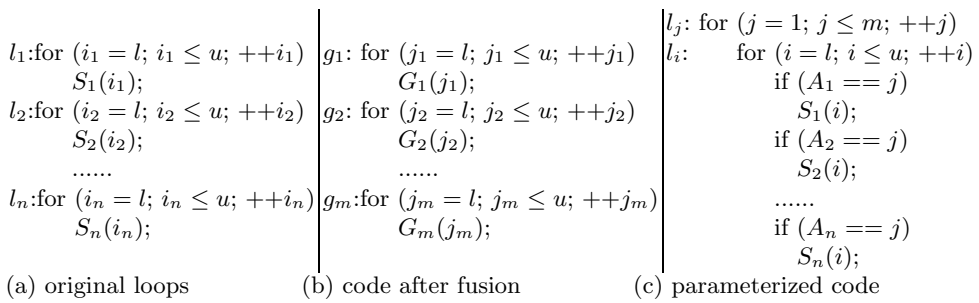
We have implemented our parameterization of loop fusion through modifications of the loop transformation algorithms by Yi and Kennedy [30] in the ROSE compiler infrastructure [23]. To exploit the search space of loop fusion, we also implemented a simple empirical search engine that exhaustively search a significant subspace of the entire fusion configuration space. In addition to using our parameterized code as a template for generating output of different fusion configurations, we also treated our parameterized code as a dynamically adaptable executable by compiling it directly and linking it with the search engine, in which case we dynamically applies different fusion transformations at runtime.

Section 3 presents our preliminary results of exploiting the empirical search space of loop fusion both at installation time and at runtime. Our experimental results are based on applying loop fusion to a kernel computation, *Riemann*, extracted from a real-world scientific application. Our results have provided new insight as well as confirming previous conclusions in applying loop fusion to improve performance of applications.

2 Empirical Tuning of Loop Fusion

This section presents our techniques for empirically applying loop fusion to optimize scientific kernels. Figure 2 shows our overall framework, which includes the following three components.

- The optimizer. A source-to-source compiler that reads in an application, performs dependence analysis to determine opportunities of applying loop fusion, then generates two outputs: a parameterized version of the transformed code, and the configuration space of the transformation. A template of the parameterized code is shown in Figure 1. The configuration space includes a sequence of integer variables used in the parameterization, and a collection of constraints on their values.
- The search engine. An empirical configuration generator that chooses integer values for the loop fusion parameters based on two inputs: the configuration space generated by the optimizer and the collected performance measurements of applications.
- The code generator. A simple configuration applicator that takes the parameterized code generated by the optimizer and the configuration of parameter



(a) original loops (b) code after fusion (c) parameterized code

Fig. 3. Parameterization of Loop Fusion

values by the search engine, instantiates the parameters with their corresponding values, and invokes the vendor compiler to produce an executable of the transformed code. The performance of the executable is measured on the target machine and the results of measurements are recorded and used by the search engine to generate the next configuration, until a satisfactory configuration is found.

In our framework, we apply the optimizer only once to generate the parameterized code and the configuration space. The result of entire compiler analysis is encoded within the parameterization. Consequently, the search engine and the the code generator do not need to perform any additional dependence analysis. Our iterative empirical tuning process thus avoids recompiling the application every time a new version is generated. In the following we describe each component of our framework in more detail.

2.1 Parameterization of Loop Fusion

Given as input a sequence of loops, as shown in Figure 3(a), our parameterization of loop fusion transformations is shown in Figure 3(c). The parameterization is based on the observation that each loop fusion transformation partitions the input loops l_1,l_2,...,l_n into a sequence of groups, G_1, G_2, ..., G_m, where $1 \le m \le n$, so that all the statements in each group are fused into a single loop. Figure 3(b) shows a template of the transformed code after applying a conventional loop fusion, where G_i $(i = 1, ..., m)$ represents the bodies of all the loops that belong to the fusion group. Each clustering of the original loops thus uniquely determines a single loop fusion transformation.

In order to parameterize the results of applying arbitrary loop fusion transformations, we need to explicitly model the relation between the clustered groups of a fusion transformation (G_1,...,G_m in Figure 3(b)) and the collection of statements in the original code (S_1,...,S_n in Figure 3(a)). In Figure 3(c), we use a sequence of integer variables, A_1,A_2,...,A_n, to model this relation. Specifically, $\forall i = 1, ..., n$, if $A_i = j$, then the original statement S_i in Figure 3(a) belongs to the clustered group G_j in (b). All possible values of A_1,...,A_n therefore comprise the configuration space of different fusion transformations.

As shown in Figure 3(c), our parameterized code has an outermost loop l_j, which enumerates the clustered groups of an arbitrary loop fusion transformations. Each iteration of loop l_j enumerates the code of the clustered group G_j in Figure 3(b). Specifically, the fused loop of group G_j is loop l_i in Figure 3(c), and the body of the fused loop includes every statement S_k $(k = 1, ..., n)$ such that $A_k == j$; that is, statement S_k is assigned to the fusion group G_j by the current fusion configuration.

The code in Figure 3(c) contains $n+1$ parameters: m, the number of clustered groups by loop fusion, and A_k $(k = 1, 2, ..., n)$, the index of the fusion group that statement S_k belongs. When these parameters are instantiated with different integer values, the code in (c) is equivalent to the conventional output of different loop fusion transformations. For instance, if $m = A_1 = A_2 = ... = A_n = 1$, the code in Figure 3(c) is equivalent to fusing all the loops in Figure 3(a) together. If $m = n$, and $A_k = k$ $\forall k = 1, ..., n$, the code in (c) is equivalent to the code in (a), where none of the loops are fused. If $m = n/3$, and $A_k = k/3$ $\forall k = 1, ..., n$, the code in (c) is equivalent to fusing every three loops in the original code in (a).

Because different instantiations of the fusion parameters can represent the entire configuration space of partitioning individual loops into different groups, the parameterized code in Figure 3(c) can be used to represent outputs of arbitrary fusion transformations on the loops in (a). Instead of applying heuristics to determine how to optimize application performance through loop fusion, we have implemented our loop fusion transformation to generate the parameterized output in (c) instead. To guarantee that only valid transformations are applied, our compiler additionally outputs a collection of constraints on the values that can be assigned to the parameters.

2.2 Configuration Space

As shown in Figure 2, after our optimizer generates the parameterized code and the configuration space of applying loop fusion transformations, a search engine tries to find a configuration that both satisfies the parameter constraints and provides the best performance.

If the loops in Figure 3(a) can be fused in arbitrary order, then the number of configurations that fuse n statements into m loops can be modeled using the following recursive formula: $F(n, m) = F(n-1, m-1) + m \times F(n-1, m)$, where $F(i, i) = F(i, 1) = 1$.[1] The total number of fusion configurations is therefore $\Sigma_{m=1}^{n} F(n, m)$, and the lower and upper bounds are $F(n, 2) = 2^{n-1} - 1$ and $n!$ respectively. The entire configuration space of loop fusion for n loops is therefore at least exponential.

In real applications, however, the number of valid configurations is much smaller because of the dependences between statements. Suppose that the dependence constraints between statements in a sequence of n loops require that all the loops be fused in exactly the order they appear in the original code; that

[1] This is the number of ways to partition n objects into m nonempty subsets

is, if S_i and S_j $(i \leq j)$ in Figure 3(a) are fused together, then all statements in $\{S_k | i < k < j\}$ are also fused with S_i and S_j. In this case, the number of configurations that fuse n loops into m loops is $\binom{n-1}{m-1}$. Figure 4 lists the number of different configurations for both reordering fusion transformations (statements can be arbitrarily reordered) and order-preserving fusion transformations (no reordering is allowed between statements).

n	1	2	3	4	5	6	7	8	9	10
Reordering	1	2	5	15	52	203	877	4140	21147	115975
Order-preserving	1	2	4	8	16	32	64	128	256	512

Fig. 4. Number of configurations

2.3 Code Generation

As shown in Figure 2, after the search engine selects a configuration, the code generator translates the parameterized code into the equivalent output of a conventional loop fusion, shown in Figure 3(b). We refer to this output as the *conventional output* of loop fusion. The translated code is then compiled into executable by invoking the vendor compiler on the target machine.

Our code generator is a simple source-to-source translator that requires no dependence analysis. Specifically, given the parameterized code in Figure 3(c), it unrolls the outer loop l_j, substitutes all parameters with their corresponding values, then removes all conditional branches inside l_i that evaluates to false after parameter substitution. The translated code is identical to the result that would be generated by a conventional loop fusion transformation.

2.4 Directly Executing Parameterized Code

Our parameterized code in Figure 3(c) can be compiled into executable even without the translation step described in Section 2.3. This feature allows us to measure dynamically the performance of different fusion configurations at runtime. Specifically, we can invoke the vendor compiler to generate a single executable from the parameterized code in Figure 3(c), and then use the search engine to set the values of the parameters in (c) at runtime. Directly executing the parameterized code avoids generating and compiling a different code for each fusion configuration.

We have built our framework so that loop fusion configurations can be tuned both at runtime, where the parameterized code are directly compiled and measured, and at installation time, where the parameterized code is first translated into conventional output before being measured. Section 3 presents our experimental results of applying both tuning processes.

3 Experimental Results and Discussion

We have implemented the techniques described in Section 2 and have applied the techniques to a computational kernel, *Riemann*, extracted from a computational fluid dynamics application. We collected data both from tuning the conventional loop fusion output (shown in Figure 3(b) and discussed in Section 2.3) and from directly tuning the parameterized output (shown in Figure 3(c) and discussed in Section 2.4). We organize our discussion around two key aspects of automatic empirical tuning of loop fusion.

- *Code generation*: In considering the use of parameterized output, we illustrate the trade-off between the accuracy of directly tuning the parameterized code and the overhead of re-compiling each version of conventional loop fusion output.
- *Search*: We provide a preliminary characterization of the configuration space of loop fusion and suggest heuristics to prune the search space based on, for instance, estimated register pressure. Our results suggest the utility of static models augmented by local search.

These results are preliminary since we consider only a single kernel and two platforms. Nevertheless, since we analyze data collected from an exhaustive search, these data serve as a useful and suggestive guide for future studies.

3.1 Experimental Setup

We have implemented the framework illustrated in Figure 2 using the fusion implementation developed by Yi and Kennedy [30], available both in D System (for Fortran) at Rice and in ROSE (C/C++) at LLNL. We use a simple implementation of the search engine that exhaustively enumerates all configurations.

We applied loop fusion to *Riemann*, a kernel which is a part of a piecewise parabolic method (PPM) code [9]. PPM is an algorithm for solving hyperbolic equations, typically from simulations of computational fluid dynamics and hypersonic flows, and is part of the benchmark suite used in procurement evaluations for the Advanced Simulation and Computing (ASC) Program within the U.S. Department of Energy. PPM is moderate in size (the 1D kernel consists of about 50 loops when unfused), compute-bound, can be completely fused (at the cost of excessive register pressure), and has been optimized by others using manual and semi-automated techniques that include fusing all loops [29].

We considered fusing just the core relaxation component that dominates the overall running time of *Riemann*. This subcomputation has 8 loops which can be completely fused, and these loops are iterated 5 times. (The kernel also contains an additional 10 pre-processing loops before relaxation, and 27 post-processing loops.) Focusing on just the 8 loops permits an exhaustive study of the search space which consists of $2^{8-1} = 128$ order-preserving fusion configurations. Because there are many data dependences among these 8 loops, we believe the order-preserving configurations represent a good subspace of all possible configurations. To make our presentation intuitive, we describe the configuration space

using bit strings of length 7, $b_1 b_2 \cdots b_7$, where $b_i = 1$ if and only if loops i and $i+1$ are fused. The bit strings are translated into corresponding integer values of the configuration parameters (shown in Figure 3(c)) in our implementation.

Since we use bit strings of length 7 to represent configurations of applying loop fusion to *Riemann*, the configuration space is a 7-dimensional space. To ease the presentation of raw data we linearize this space (*e.g.*, see Figure 5) using a Gray code ordering, where neighboring points in the linearization differ by only 1 bit, *i*.e., differ in only 1 pair of loops being fused or not fused. The specific Gray ordering we use is a binary-reflected Gray code [5]; we take the first point to be linear index 0 and configuration 0000000 (all unfused), and the last point to be linear index 127 and configuration 1000000. The all-fused case, 1111111, is linear index 85.

We performed experiments on two architectures: a 2.4 GHz Intel Xeon with 8 floating point registers, 8KB L1 cache, 512KB L2 cache; and a 650 MHz Sun UltraSPARC IIe with 32 registers, 16KB L1 cache, 512KB L2 cache. We use the Intel C/C++ compiler 9.0 with "-O3" on the Xeon, and the Sun Workshop 9.0 C/C++ compiler with option "-O5" on the UltraSPARC.

For each configuration we observed hardware counters using PAPI [6], recording process-specific cycles, instructions, cache and TLB misses, and branch mispredictions. To reduce the sensitivity of our reported counts to system noise, we measure each metric for each fusion configuration 9 times and report the minimum. The time to run the kernel is roughly a few tenths of a second on the Xeon and a few seconds on the UltraSPARC.

3.2 Parameterized *vs.* Conventional Output

There is a potential time and accuracy trade-off in choosing to use either parameterized or conventional output. Empirical tuning using the parameterized output is more flexible in that we need to produce only a single executable which is then dynamically adapted for each configuration at runtime, whereas using conventional output needs to repeatedly invoke both the code generator in Figure 2 and the vendor compiler to generate executable for each configuration. However, the parameterized code has a much more complex control structure due to the configuration logic. If we choose to use the parameterized form to search for best configuration, we must ask how accurate this search will be.

We compare the accuracy of using parameterized and conventional output in Figure 5, which shows the running time (in cycles) of parameterized output and conventional output for the 128 fusion configurations on the Xeon platform. Observe that the two types of implementations qualitatively track one another. However, the parameterized codes take up to 25% more time to execute due to their complex control structure. We can further quantify the similarity of the two curves using statistical correlation, where a correlation close to 1 indicates a strong increasing linear relationship, -1 indicates a strong decreasing linear relationship, and 0 indicates no linear relationship. On the Xeon, this correlation is 0.85, while on the UltraSPARC it is 0.75, both indicating moderate-to-strong

linear relationships. However, these relationships are not perfect and so there may be performance estimation inaccuracy when using parameterized output.

Figure 6 shows the extent to which the parameterized codes reproduce other features of the execution behavior (like cache misses) of the conventional codes on the Xeon. Specifically, we show the ratio of measurements for the parameterized codes to those of the conventional codes. Cycles, store instructions, and L2 and TLB misses—expected to be the most expensive misses—are well-reproduced, showing errors of 25% or less. However, there are uniformly many more load instructions, and in several cases, many more L1 misses. The excess loads can be explained by two characteristics of the parameterized output: (1) the need for additional integer variables to describe the configuration, and (2) an inability of the vendor compiler to apply scalar replacement to reduce memory accesses. The spikes in L1 misses tend to occur in configurations that introduce a considerable degree of fusion. Nevertheless, we expect L1 misses to be relatively cheaper than L2 and TLB misses, so L1 miss behavior may be less necessary to reproduce accurately. Results on the UltraSPARC (not shown) are qualitatively similar except that the largest ratios are smaller (less than 1.7).

We also report on the turnaround time to evaluate each point in the search space. On the Xeon, we observed an average time of 1.89 seconds to generate and compile a conventional output, and 0.18 seconds to execute it, compared to an average of 0.67 seconds just to execute the parameterized version. The overhead of the parameterized output is due to extra control logic, the function call to the runtime configuration generator, and the missed optimization opportunities from the vendor compiler. On the UltraSPARC, it takes 3.0 seconds to generate and compile the conventional output, and 1.3 seconds to execute, while it takes 3.5 seconds for the parameterized version to execute. Thus, the relative costs will depend on the application itself and the underlying architecture.

3.3 Properties of the fusion configuration space

We present a preliminary characterization of the fusion configuration space, based on our exhaustive search data, focusing exclusively on the conventional output. We describe several properties of the search space, each implying a possible search space pruning technique.

The distribution of performance in the configuration space can be summarized as follows. On Xeon, the speedup over the completely unfused case is between 0.88 and 1.06, with a standard deviation (σ) equal to .039. There are 90 configurations within 5% of the best speedup and 112 points within 10%. On Sun Sparc, the speedup is between 0.97 and 1.11, with $\sigma = .026$. There are 19 points within 5% of the best speedup and 94 points within 10%. Although these data indicate there are many implementations with reasonably good performance, we note that the penalty for making a mistake can be significant: on the Xeon, choosing a poor configuration leads to a "speedup" as low as .88.

The Gray ordering reveals relationships among fusion configurations that could be exploited by a guided search. Figure 5 gives the running time in cycles,

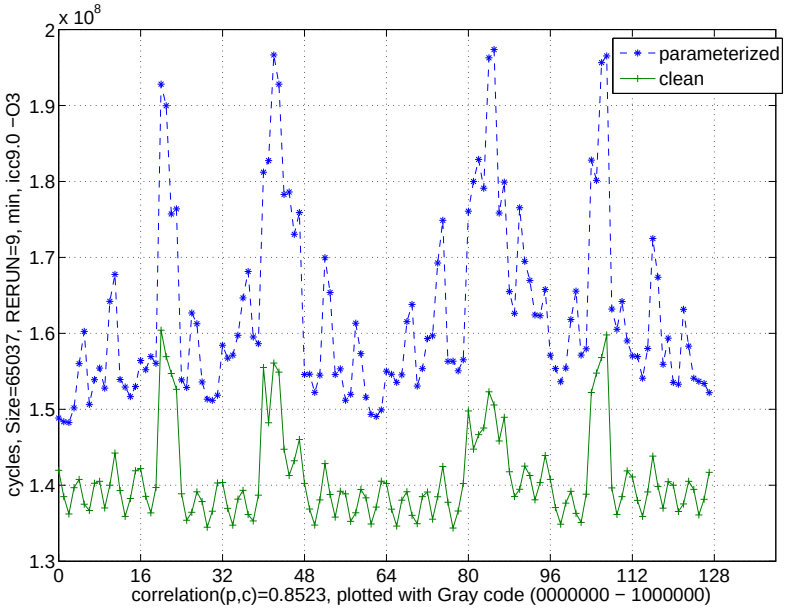


Fig. 5. Correlation of cycles on Intel Xeon

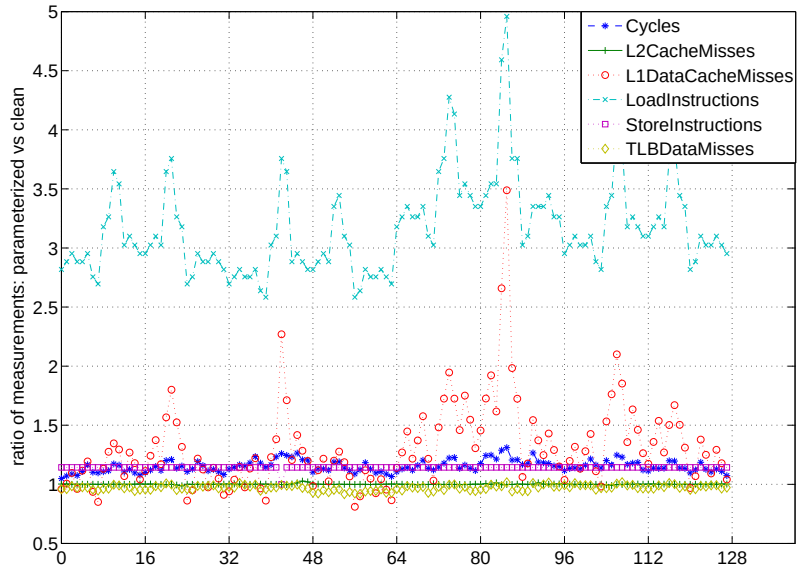


Fig. 6. Ratio of measurements on Intel Xeon: parameterized vs clean version

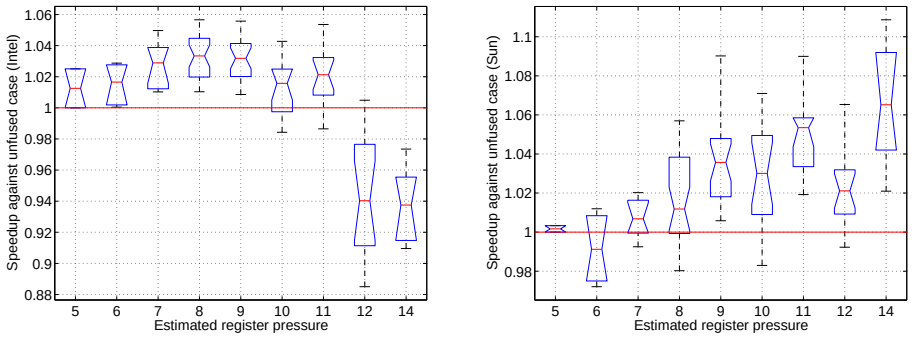


Fig. 7. Speedup *vs.* register pressure on Xeon (*left*) and UltraSPARC (*right*)

with all configurations linearized along the x-axis by a binary-reflected left-to-right Gray code (see Section 3.1). There is a qualitative mirror-symmetry in several places, indicating which fused pairs do not affect performance too much. For example, comparing configurations 0–63 to 127–64 indicates that fusing loops 1 and 2 (bit b_1) has little effect on the running time. However, segments 0–15 and 31–16, which differ only in bit 3, are not mirror symmetric, suggesting that fusing loops 3 and 4 has a non-trivial effect on performance. In much larger search spaces, it should be possible to detect these interactions quickly by sampling.

Though the curves are not smooth in Figure 5, many of the "peaks" and "valleys" are localized (*e.g.*, see the grouped peaks at 20–23, 40–43, 80–87, 104–107). This observation suggests gradient-descent type methods with random perturbations may be an effective search technique.

We consider pruning the search space using a purely static estimation of register pressure. We estimate the number of registers needed to keep every array reference in a register for the purpose of data reuse as described by Carr and Kennedy for unroll-and-jam [7]. We assign to each configuration the maximum estimated register pressure among all loops after fusion. Of course, many configurations may have the same register pressure estimate.

We show the distributions of speedup as a function of this estimated register pressure using box-plots, as shown in Figure 7 for the Xeon (*left*) and the UltraSPARC (*right*). At each register pressure value, we show the minimum, maximum, median by short horizontal lines, and the 25th and 75th percentiles of points by trapezoids. On the Xeon, which has only 8 scalar floating point registers, the best speedup is achieved when the register pressure is 8; when the register pressure exceeds 11, the performance drops. By contrast, the UltraSPARC, which has 32 registers, achieves a maximum speedup when there is a high-degree of fusion (in this case, all loops fused). This confirms our intuition that fusing too much can cause performance degradation due to the limited resources such as registers. However, observe also that the register pressure estimate is not enough to identify the optimal configuration since there is generally a wide spread in performance at each register pressure value. In the absence of other possible static properties of the code, this observation suggests that the register pressure estimate could be combined with some form of local search.

4 Related Work

Loop fusion is one of the many compiler transformation techniques (*e.g.*, blocking, distribution, interchange, skewing, and index-set splitting [1, 2]) that can significantly improve the memory hierarchy performance of scientific applications. However, optimal fusion is NP-complete in general [11] and heuristics must be applied. For example, Kennedy and McKinley [16] proposed a *typed fusion* heuristic, which can achieve maximal fusion for loops of a particular type in linear time. This paper extends previous loop fusion transformation algorithms to parameterize their output so that an external empirical search engine can take advantage of a compiler's knowledge.

Previous work in empirical tuning of applications can be roughly divided into two categories: *domain-specific approaches* which use specialized kernel-specific code generators and search engines that exploit problem-specific knowledge and representations, and *compiler-based approaches* which extend traditional compiler infrastructures with parameterized transformations and search engines. These approaches are complimentary since the domain-specific methods frequently output high-level source and rely on a compiler to perform the final machine-code generation, as noted in a recent survey [28, Sec. 5].

The domain-specific approach has been applied to a broad variety of problem areas, including linear algebra [12] and signal processing [13, 20], among others [28, Sec. 5]. These systems all feature special parameterized code generators which take as input a desired kernel and specific parameter values, and output a kernel implementation (typically in Fortran or C). Some generators permit users to specify the desired kernel operation in a high-level domain-specific notation [20, 3, 4]. The parameters express algorithmic or implementation details thought to be machine- or program input-specific, as determined by the experience and domain-knowledge of the generator writer. A separate search phase, occurring at build-time or at run-time (or both), selects the parameter values.

The Tensor Contraction Engine (TCE) synthesizes and tunes entire parallel quantum chemistry programs from a specification based on tensor notation [3]. TCE performs fusion among other transformations, but does so to reduce memory usage rather than to enhance locality or parallelism.

Compiler-based tuning approaches belong broadly to the area of feedback-directed optimization [24], which includes superoptimizers, profile-guided and iterative compilation, dynamic optimization, and empirical tuning of the compiler itself. Earlier work on *iterative compilation*, of which this work is an instance, has studied the tuning of tile sizes and unrolling factors [17, 19, 22, 25, 14], and more recently, the search space pruning of loop fusion [21]. Previous work as listed above did not focus on finding a suitable parameterization of the transformations. Other approaches consider augmenting the internal static models and decision-making processes within a compiler through collection of empirical data when the compiler is installed for a target architecture [18, 26, 27, 10].

Though this paper does not focus on search itself, fusion tuning should benefit from prior tuning work that uses combinatorial search methods [28, Sec. 5]. These techniques include simple exhaustive search, random search, simulated an-

nealing, statistical methods, evolutionary algorithms, and hybrid empirical and analytical modeling combined with local search techniques [31, 8]. Hybrid models should be effective for fusion tuning, as argued by Qasem and Kennedy [21] and further supported by this paper's experimental results.

5 Conclusions and Future Work

Our proposed parameterization technique provides traditional compiler infrastructures with a compact interface for communicating the space of loop fusion transformations to an external search engine, thereby enabling empirically tuned fusion for general programs. We are pursuing the development of parameterizations for several other individual loop and data layout transformations, and for combinations of these transformations.

References

1. R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, San Francisco, October 2001.
2. D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, 1994.
3. G. Baumgartner, A. Auer, D. E. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X. Gao, R. J. Harrison, S. Hirata, S. Krishnamoorthy, S. Krishnan, C.-C. Lam, Q. Lu, M. Nooijen, R. M. Pitzer, J. Ramanujam, P. Sadayappan, and A. Sibiryakov. Synthesis of high-performance parallel programs for a class of *ab initio* quantum chemistry models. *Proc. IEEE*, 93(2), 2005.
4. P. Bientinesi, J. A. Gunnels, M. E. Myers, E. Quintana-Orti, and R. van de Geijn. The science of deriving dense linear algebra algorithms. *ACM TOMS*, 31(1):1–26, Mar. 2005.
5. P. E. Black. Gray code. `www.nist.gov/dads/HTML/graycode.html`.
6. S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In *SC*, Nov. 2000.
7. S. Carr and K. Kennedy. Improving the ratio of memory operations to floating-point operations in loops. *ACM TOPLAS*, 16(6):1768–1810, 1994.
8. C. Chen, J. Chame, and M. Hall. Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy. In *CGO*, Mar. 2005.
9. P. Colella and P. Woodward. The piecewise parabolic method for gas-dynamical simulations. *J. Comp. Phys.*, 54(174), 1984.
10. K. D. Cooper, D. Subramanian, and L. Torczon. Adaptive optimizing compilers for the 21st century. *Journal of Supercomputing*, 23(1):7–22, 2002.
11. A. Darte. On the complexity of loop fusion. *Parallel Computing*, 26(9):1175–1193, 2000.
12. J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, A. Petitet, R. Vuduc, C. Whaley, and K. Yelick. Self adapting linear algebra algorithms and software. *Proc. IEEE*, 93(2), 2005.
13. M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proc. IEEE*, 93(2), 2005.

14. G. Fursin, A. Cohen, M. O'Boyle, and O. Temam. A pratical method for quickly evaluating program optimizations. In *Proceedings of the International Conference on High Performance Embedded Architectures and Compilers*, Nov. 2005.
15. K. Kennedy. Fast greedy weighted fusion. In *ICS*, Santa Fe, NM, May 2000.
16. K. Kennedy and K. S. McKinley. Typed fusion with applications to parallel and sequential code generation. Technical Report TR93-208, Dept. of Computer Science, Rice University, Aug. 1993. (also available as CRPC-TR94370).
17. T. Kisuki, P. M. Knijnenburg, and M. F. O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. In *PACT*, Oct. 2000.
18. N. Mitchell, L. Carter, and J. Ferrante. A modal model of memory. In *ICCS*, volume 2073 of *LNCS*, pages 81–96, San Francisco, CA, May 2001. Springer.
19. G. Pike and P. Hilfinger. Better tiling and array contraction for compiling scientific programs. In *SC*, Baltimore, MD, USA, Nov. 2002.
20. 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. IEEE*, 93(2), 2005.
21. A. Qasem and K. Kennedy. A cache-conscious profitability model for empirical tuning of loop fusion. In *LCPC*, Oct. 2005. (to appear).
22. A. Qasem, K. Kennedy, and J. Mellor-Crummey. Automatic tuning of whole applications using direct search and a performance-based transformation system. In *LACSI*, 2004.
23. D. Quinlan, M. Schordan, Q. Yi, and A. Saebjornsen. Classification and untilization of abstractions for optimization. In *The First International Symposium on Leveraging Applications of Formal Methods*, Paphos, Cyprus, Oct 2004.
24. M. D. Smith. Overcoming the challenges to feedback-directed optimization. In *Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization (Dynamo)*, Boston, MA, USA, Jan. 2000.
25. M. Stephenson and S. Amarasinghe. Predicting unroll factors using supervised classification. In *CGO*, San Jose, CA, USA, Mar. 2005.
26. M. Stephenson, S. Amarasinghe, M. Martin, and U.-M. O'Reilly. Meta optimization: improving compiler heuristics with machine learning. In *PLDI*, June 2003.
27. S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. I. August. Compiler optimization-space exploration. In *CGO*, pages 204–215, Mar. 2003.
28. R. Vuduc, J. Demmel, and J. Bilmes. Statistical models for automatic performance tuning. *IJHPCA*, 18(1):65–94, 2004.
29. P. R. Woodward and S. E. Andersen. Portable petaflop/s programming: Applying distributed computing methodology to the grid within a single machine room. In *Proceedings of the 8th IEEE International Conference on High Performance Distributed Computing*, Redondo Beach, CA, USA, Aug. 1999.
30. Q. Yi and K. Kennedy. Improving memory hierarchy performance through combined loop interchange and multi-level fusion. In *LACSI Symposium*, Santa Fe, NM, Oct 2002.
31. K. Yotov, X. Li, G. Ren, M. Garzaran, D. Padua, K. Pingali, and P. Stodghill. Is search really necessary to generate high-performance BLAS? *Proc. IEEE*, 93(2), 2005.