A Tuning Framework for Software-Managed Memory Hierarchies

Manman Ren
Stanford University
mmren@stanford.edu

Ji Young Park
Stanford University
jypark76@stanford.edu

Mike Houston
Stanford University
mhouston@graphics.stanford.edu

Alex Aiken
Stanford University
aiken@cs.stanford.edu

William J. Dally
Stanford University
dally@stanford.edu

ABSTRACT

Achieving good performance on a modern machine with a multi-level memory hierarchy, and in particular on a machine with software-managed memories, requires precise tuning of programs to the machine's particular characteristics. A large program on a multi-level machine can easily expose tens or hundreds of inter-dependent parameters which require tuning, and manually searching the resultant large, non-linear space of program parameters is a tedious process of trial-and-error. In this paper we present a general framework for automatically tuning general applications to machines with software-managed memory hierarchies. We evaluate our framework by measuring the performance of benchmarks that are tuned for a range of machines with different memory hierarchy configurations: a cluster of Intel P4 Xeon processors, a single Cell processor, and a cluster of Sony Playstation3's.

Categories and Subject Descriptors

C.4 [**Performance Of Systems**]: Measurement techniques, Modeling techniques; D.3.4 [**Programming Languages**]: Processors—*Compilers, Optimization*

General Terms

Algorithms, Design, Performance

1. INTRODUCTION

Program performance is often limited by data movement, not arithmetic. An emerging class of high performance architectures, including *stream* processors such as Stanford's Merrimac [9] and Imagine [21] and the Sony/Toshiba/IBM Cell Broadband Engine Processor [26] (Cell), use software-managed memory hierarchies to bridge the gap between memory bandwidth and arithmetic throughput. Unlike cache-based machines, machines of this type require applications to explicitly orchestrate all data transfers between on-chip and off-chip memories, and further, to explicitly manage data allocation in the on-chip local memories. Due to this fundamentally different programming requirement, a number of programming systems have emerged to simplify the task of programming machines with software-managed memory hierarchies [24, 5, 14, 25, 3, 13, 12]. In this paper, our applications are coded in *Sequoia* [14], a language whose goal is to provide portable performance across machines with varying explicitly-managed memory hierarchies.

Sequoia focuses on the decomposition and communication aspects of a problem so that algorithms can be structured to be bandwidth-efficient. Sequoia achieves portability through parameterized application decomposition. We call the machine-dependent parameters *tunables* and the ratios between values of tunables *the shape of the tunables*. The performance of the application depends heavily on the values of the tunables. With large programs exposing non-linear, multi-dimensional parameter spaces, it is natural that programmers are increasingly looking to automated tuning approaches to avoid the tedious, error-prone process of manually tuning applications. As architectures and applications increase in complexity, statically predicting the performance of an application becomes an intractable problem outside of certain regular application domains, and thus a recent trend [23, 6, 31, 32] is to combine empirical tuning with static modeling to tune a program for a particular machine.

Given that memory bandwidth is scarce, compilers should ensure that data elements are reused as often as possible at each level of the memory hierarchy. Loop fusion is a well-known compilation technique that enhances locality by merging loop nests that access similar sets of data. When two loop nests are fused, the tunables of the two loop nests are combined, which means compromising on the best values of each individual loop nest. Because of capacity constraints, the tunables after fusion are usually smaller than prior to fusion. Also the tunable shape of one loop nest can be very different from the shape of another, and the combination of two different shapes can cause serious performance degradation. As an example, for two loop nests of our `FFT3D` benchmark, the performance after fusing two loop nests is 5 times worse than without fusion on the Cell processor.

The work presented in this paper extends existing tuning and compilation techniques to handle machines with software-managed memory hierarchies. Our contributions

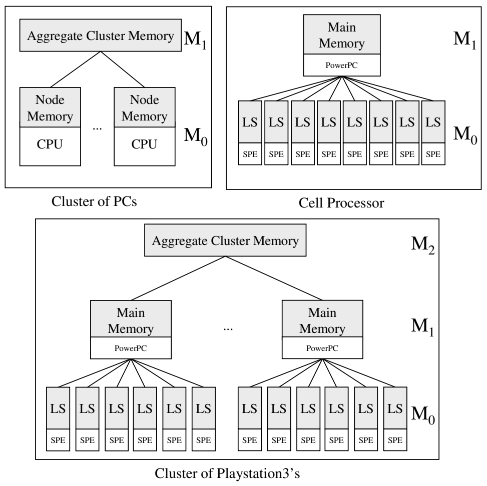

Figure 1: Machine models for our system configurations

are as follows:

- We characterize the search space of tunables for machines with software-managed memory hierarchies and present several methods to search the space. Only 20 evaluations are required to achieve 90% performance for all three platforms.
- We present a loop fusion algorithm targeting software-managed memory hierarchies, which considers mismatch of the tunables.
- Our tuning framework is evaluated by running benchmarks on a cluster of Intel P4 Xeon processors, on a single Cell processor and on a cluster of Sony PS3s, in each case comparing the performance obtained by our automatically tuned program against the best-available hand-tuned version coded in Sequoia ([20]); in all cases, our automated framework achieved similar or better performance.

The rest of the paper is organized as follows. We describe the machine model and the programming model in Section 2 and give an overview of the tuning framework in Section 3. In Section 4, we characterize the search space of the tunables and present algorithms to search the tunable space. We present the loop fusion algorithm in Section 5. Section 6 evaluates the framework, Section 7 discusses related work and Section 8 concludes.

2. BACKGROUND

2.1 Abstract Machine Model

We represent machine hierarchies as *trees of nodes.* Each node of the machine hierarchy has storage (memory) and may have the ability to perform computation. This simple model captures the important features of modern machines with multi-level (and software-managed) memory hierarchies. In this paper, we target 3 machines: a Cell processor, a cluster of PCs and a cluster of PS3s; the abstract machine models are shown in Figure 1. *Virtual levels* [14] are used to model inter-node communication in a cluster: transferring data from the aggregate cluster memory to the memory in a single node may result in communication among the cluster nodes.

2.2 The Sequoia Programming Language

In brief, the input language (Sequoia) [14] has the following properties:

- **Hierarchical, bulk decompositions:** Sequoia is designed to permit the expression of program decompositions: computations on large datasets are split into sub-computations on sub-datasets. All data transfers and computations are expressed in bulk.
- **Isolated tasks:** Sequoia's core construct is the *task*: a side-effect free function that executes on private bulk data. Sequoia allows the programmer to provide several implementations, or *variants*, of a task. At any given task call site, there may be multiple choices for which variant of the subtask to call.
- **Tunables:** task parameters or *tunables* can be set to different values for different machines.

Sequoia focuses on the decomposition and communication aspects of a problem, and how to intelligently structure algorithms to be bandwidth-efficient. Thus high-performance leaf tasks are generally written in a platform-specific language. We impose a phase order between tuning the Sequoia program and tuning the leaf tasks. First the leaf tasks are tuned with the problem size chosen to assure all data is accessed from the lowest memory level. This should simplify the problem of scheduling the contents of the leaf tasks because the uncertainty of a memory operation is reduced. Next our tuning framework is invoked to maximize the performance of the whole application. A user can use the findings of the tuning framework to determine the bottleneck of the application, and to decide whether it is necessary to improve the implementation of a certain leaf task.

A Sequoia implementation of 2D convolution is given in Figure 2. The `conv2d` task convolves a 2D array ((N_y+U-1) x (N_x+V-1)) with a 2D filter (U x V) to generate a 2D array (N_y x N_x). The `inner` variant partitions the input array into blocks and iterates over the small-sized convolution performed on these blocks.

```
void task conv2d::inner(in d1[Ny+U-1][Nx+V-1],
                        out d2[Ny][Nx], in f[U][V])
{
  tunable YBLK, XBLK;
  mappar (int j=0 to Ny/YBLK;
          int i=0 to Nx/XBLK) {
    conv2d(d1[j*YBLK;YBLK+U-1][i*XBLK;XBLK+V-1],
           d2[j*YBLK;YBLK][i*XBLK;XBLK], f);
  }
}
void task conv2d::leaf_ext(...); //external leaf task
```

Figure 2: 2D convolution in Sequoia

2.3 Data Reuse

Since Sequoia deals with bulk operations and each array reference refers to a range of an array, we extend the standard reuse analysis for Sequoia. Define $footprint(R, L, \vec{v})$ as a region of data accessed by reference R at iteration vector $\vec{v}$ of loop nest L. If $footprint(R_s, L_s, \vec{vs})$ overlaps with $footprint(R_d, L_d, \vec{vd})$, we say there exists

- *loop-carried reuse* when $L_s = L_d$, $\vec{vs} \neq \vec{vd}$.
- *loop-independent reuse* when $L_s = L_d$, $\vec{vs} = \vec{vd}$.
- *inter-loop reuse* when $L_s \neq L_d$.

We say *reuse is exploited* if the reuse leads to saved memory accesses.

3. TUNING FRAMEWORK DESIGN

The tuning framework presented in this paper first maps the Sequoia program to the target machine. Next, a bottom-up pass empirically explores the tunable space of each memory level, beginning with the lowest memory level (Section 4). Finally the integrated loop fusion algorithm (Section 5) is invoked to select a loop order for each loop nest and a fusion configuration to maximize the profitability of loop fusion.

On cache-based architectures, [36] makes tiling decisions after loop fusion and loop distribution. But for software-managed memory hierarchies, we believe it is better to perform loop fusion after exploring the tunable space. The detailed reasons are given in Section 5.

3.1 Mapping Programs to Target Machines

The tuning framework maps a Sequoia program to the target machine by matching the decomposition hierarchy with the machine's memory hierarchy, placing data into a memory level and annotating control statement with the level of a machine at which it will execute. The tuning framework uses a top-down algorithm (starting from each entry task) to generate multiple such *mapped* versions of the program, which are consumed by subsequent stages; ultimately the fastest version is selected.

When setting the level of each data object and the execution level of each statement, the following constraints apply:

- All arguments of a task are located within a single level of the memory hierarchy and are resident at the memory level `L` the entire time the task is in progress. If an element of an argument is reused inside the task, the reuse is exploited at level `L`.
- A control statement at level `L` can only access data objects (scalar variables or array blocks) that are placed in the same level `L`.

A copy operation is inserted by the framework if a data object resides in one level and it (or part of it) is needed in another level.

Consider the `conv2d` example, the call graph of the original program is displayed in the left column of Figure 3. Only the name of the callee task is specified for the call site in `conv2d::inner`, so it can call either `conv2d::inner` or `conv2d::leaf`. The expanded call graph targeting a cluster of PS3s is shown in the right side of Figure 3. Copy operations that transfer data between M_2 and M_1 are inserted since the call site at M_1 accesses subblocks of arrays at M_2. Similarly copy operations which move data between M_1 and M_0 are inserted.

3.2 Performance Measurement

For each *control level* (machine level that can perform computation), profiling code measuring the performance of each loop nest and each bulk operation (data transfer or task call) is inserted during code generation. This level-aware profiling facilitates level-by-level tuning. Consider the `conv2d` example targeting a cluster of PS3s. For the loop nest at M_1, the profiling system will collect the run time of the loop nest, the run time of each copy operation that moves data between M_2 and M_1, and the run time of the task call that is invoked by the call site at M_1. The profiling system collects the same data for the loop nest at M_0.

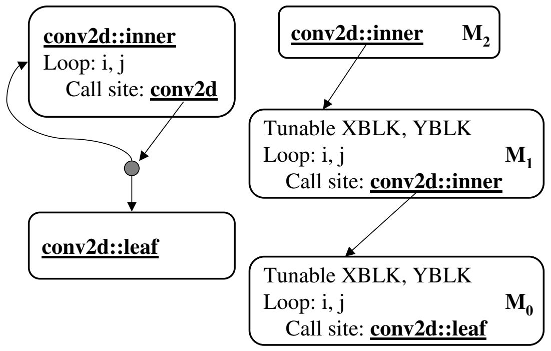

Figure 3: Call graph of the original program (left) and the version mapped to a cluster of PS3s (right)

In our experience, the profiling system gives the user valuable feedback. For each loop nest at each control level, is it communication-bound or computation-bound? For each control level, which loop nest (which bulk operation) is the most time-consuming?

4. SEARCHING THE TUNABLE SPACE

We first discuss how to reduce the size of the tunable search space. Next we characterize the search space of tunables on software-managed memory hierarchies. Finally we give methods for empirically searching the space.

4.1 Pruning the Search Space

A *point* is an assignment of values to the tunables. Since data is already allocated to a specific memory level in Section 3.1, a point is *infeasible* if data no longer fits in the memory level with the assignment of tunables. This capacity constraint is used to prune the search space. In addition, several types of user-provided constraints are supported in the system: a tunable can be constrained to be bound below, bound above, a multiple of, or a factor of an integer. These constraints allow a user to express correctness conditions for external leaf tasks (e.g. alignment restrictions) and to manually prune the search space.

4.2 Reducing the Dimensionality of the Search Space

The dimensionality of the search space is equal to the number of tunables and the number of tunables increases with the depth of the memory hierarchy. The largest application we evaluated has 78 tunables on a cluster of PS3s, and searching the resultant high-dimensional tunable space is very time-consuming. Thus we reduce the dimensionality by grouping tunables and searching each group separately.

The *level* of a tunable is L if it affects the decomposition of a problem at level $L+1$ to a set of sub-problems at level L. When exploring the search space of tunables at level L, a problem size that fits in level $L+1$ is chosen, and profiling results at level L are collected to guide the search. In order to collect profiling results, tunables at lower levels should have been set. Thus we use a bottom-up approach, but re-evaluate the decisions made at lower levels when necessary.

Consider targeting a cluster of PS3s, which has three levels M_2, M_1 and M_0. We first search the space of tunables at M_0 with a problem size P_1 chosen to fit M_1. Next tunables at M_0 are set to the best point found at the previous step, and the space of tunables at M_1 is searched. The problem size determined by the best values of tunables at M_1 is P_2. If P_2 is different from the assumed size P_1, to get the optimal performance, the tunable space at M_0 should be explored again. Unlike cache-based architectures, where conflict misses vary with the problem size, for software-managed memory hierarchies, we observe much less correlation between the problem size and the best values of tunables. If the run time of the problem at M_2 is dominated by communication operations, or if both problem sizes P_1 and P_2 at M_1 are much larger than the best problem size at M_0, there is little or no benefit gained from re-exploring the space at M_0. In fact, for all our benchmarks, at least one of the two conditions are satisfied. Therefore, a single bottom-up pass suffices.

Changing the values of the tunables involved in one loop nest, in many cases has little or no effect on the performance of other loop nests. To exploit this independence, we further divide tunables in the same level into groups. If two tunables are not involved in the same loop nest, we say they are independent and belong to different groups. Separate instances of the search algorithm on individual groups are initiated with loop level profiling results to guide the search. And those instances run simultaneously to reduce the tuning time.

4.3 Characteristics of the Search Space

We compare the search space of tile sizes on cache-based machines with the search space of the tunables on software-managed memory hierarchies. Since conflict misses play a significant part in the cache behavior of blocked algorithms, the repetitive characteristic of conflict misses causes the search space of tile sizes to be periodic with high frequency oscillations. Studies [22][17] have shown that indeed the search space is neither smooth nor continuous. A small deviation from "good" tile sizes can cause a huge increase in execution time. Due to conflict misses, "good" tile sizes usually utilize only a fraction of the cache's capacity, and square tile sizes usually work well.

Consider how the performance of an application changes with the tunables on software-managed memory hierarchies. First, the amount of reuse that is exploited changes when the tunables are varied. We can estimate the exploited reuse by the number of memory transfers: more exploited reuse means fewer bytes transferred. For our `conv2d` example, the number of elements transferred scales with $1 + \frac{U-1}{YBLK}$ and $1 + \frac{V-1}{XBLK}$. For IJK version of matrix multiplication with NxN problem size, the amount of transfer scales with $N/JBLK$ and $N/IBLK$. For different applications, the exploited reuse varies with the tunables in different ways. Second, transfer sizes of communication operations vary as the tunables. We achieve higher bandwidth for larger transfers, particularly for MPI communication operations between nodes and DMA operations across levels. Third, the values of tunables can impact the number of TLB misses, because they change the way arrays are traversed. Finally, alignment of transfer operations and SIMD operations in the leaf tasks affect performance.

We study the search space of tunables by evaluating all the feasible points on a coarse grid (some tunables are multiples of 8). For `conv2d` on Cell, the search space is shown in Figure 4. We notice that the space is smooth and the high frequency components due to alignment issues cause variations of no more than 20 percent. If we downsample the space by collecting the points that are multiples of 32, most of the high frequency components are gone (i.e. the data is properly aligned). We observe similar characteristics on our other benchmarks running on Cell. A rougher surface is observed for our benchmarks on a cluster of PCs because each node is a cache-based machine. And we notice that the best values are often close to the boundary created by the capacity constraints.

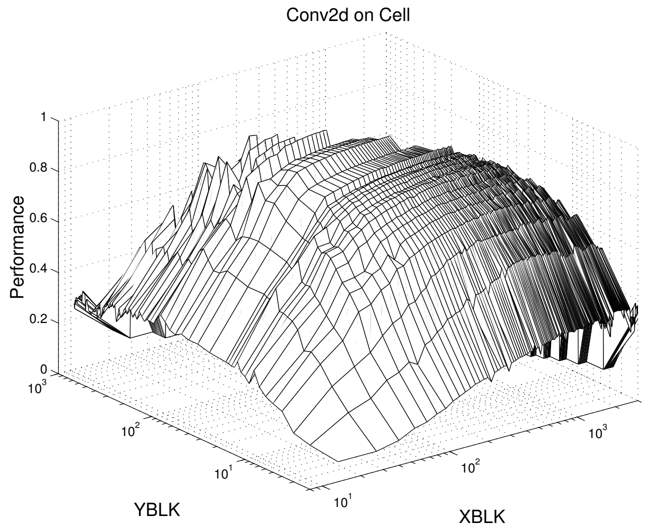

Figure 4: Search space of conv2d on Cell

We notice that *square tunables* (i.e. the same value is used for multiple tunables) do not work well for several tunable groups of SUmb (Stanford University MultiBlock, see Section 6). On Cell, the best tunable values for two loop nests of `SUmb` are (128,1,4) and (128,4,1) respectively, far from the square shape. With the best square tunables, the performance of the two loop nests degrades 6.5x and 5.4x, due to small transfer sizes.

In summary, the tunable search space on software-managed memory hierarchies displays different characteristics from the search space on cache-based architectures:

- Smoothness: The search space is rough for cache-based machines due to the repetitive characteristic of conflict misses. If a subblock is copied to a contiguous region, the search space becomes much smoother due to reduced self-interference misses. For software-managed memory hierarchies, the search space is smooth with high-frequency components due to alignment issues.
- Sensitivity to the tunable shape: Memory bandwidth saturates at the cache line size for cache-based architectures, but on software-managed memory hierarchies, the achieved bandwidth of bulk transfers still scales up at 1K bytes. This often requires tunables affecting the transfer size to be larger than other tunables on software-managed memory hierarchies. We say the performance is more sensitive to the tunable shape for software-managed memory hierarchies than for cache-based architectures.
- Closeness to the search boundary: Due to conflict misses on cache-based architectures, only a small portion of the cache capacity is utilized when achieving the best performance. However for machines with software-managed memory hierarchies, the best tunable values are often close to the capacity boundary.

- Sensitivity to the problem size: Performance of an application is sensitive to the problem size on cache-based architectures due to the correlation between self-interference misses and the problem size.

4.4 The Search Algorithm

We employ a pyramid search that starts with a coarse grid, and refines the grid when no further progress can be made. At each grid level, we are looking at the downsampled space, thus ignoring the local oscillations of the search space. Since the search space for software-managed memory hierarchies is relatively smooth compared to the search space of cache-based machines, greedy search algorithms that rely solely on profiling can achieve good performance quickly. To avoid local minima, once we are done with one pass of the search from the coarsest grid to the finest grid, we restart the pass again with a base point at each grid level chosen to be the point with the best performance among the evaluated points that have not been used as base at this grid level. The search stops when no progress is made for the last restarting or when the maximal number of evaluations is reached.

4.4.1 Setup of the Search Algorithm

We associate a weight with each tunable. Initially the weight of any tunable (w_t) is set to zero. If the transfer size of an operation gets larger when the tunable is increased, we update its weight to $max(w_t, w_1)$, w_1 is a machine-specific constant. If the exploited reuse increases as the tunable, we set its weight to $max(w_t, w_2)$, w_2 is application-specific depending on how the exploited reuse scales with the tunable. Currently, we set both w_1 and w_2 to 1 for any application.

To get the initial point, we set the tunables with weight zero to minimum possible values, and set other tunables to the maximum feasible values, with the ratio between tunables equal to the relative weight. As shown in the evaluation section, *square grid* (i.e. same grid spacing in each tunable dimension) does not work well for some of our benchmarks. In fact, we update the ratios between grid spacing in each tunable dimension to be the ratios between tunable values of the initial point.

4.4.2 The Algorithm at Each Grid Level

Define vector s_t as one step along the direction of tunable t, and vector v as the base point. The algorithm is described in Figure 5. If the current best point is on the boundary, it is very likely that the algorithm will become stuck, since making any tunable larger will make the new point infeasible. $step(v, s_t)$ handles the boundary case by first checking whether $v + s_t$ is feasible. If it is infeasible, it keeps the value of tunable t, and reduce the values of other tunables until a feasible point is found.

5. INTEGRATED LOOP FUSION

Loop fusion is a well-known compilation technique to reduce the distance of inter-loop reuse by merging pairs of loop nests. However loop fusion can increase the distance of the original loop-carried reuse since more data is touched in the fused loop. Traditional loop fusion algorithms often use reuse distances to estimate profitability. However, reduced reuse distances do not guarantee performance improvement. Consider the last two loop nests from our `FFT3D` benchmark (shown in Figure 6). There exists a single inter-loop reuse pair between reference to $d2$ in the forth loop nest and ref-

```
procedure search(v)
  while progress is made
    choose $x \in \{+s_t, -s_t \mid \forall$ tunable $t\}$
      s.t. $f'$ is maximized, where $\{v', f'\} \leftarrow step(v, x)$
    if no better point is found, break
    $v \leftarrow$ the best point
    do
      $\{v', f'\} \leftarrow step(v, x)$
      if $v'$ is better than $v$, $v \leftarrow v'$
      else break
    while true
procedure step(v, x)
  if $x$ is a backward step
    $f' \leftarrow$ evaluate the performance at $v + x$
    return $\{v + x, f'\}$
  $v' \leftarrow v + x$
  while $v'$ is infeasible
    $v' \leftarrow$ reduce values of tunables other than $t$, where $x = s_t$
    if tunables cannot be reduced further, return $\{v, 0\}$
  $f' \leftarrow$ evaluate the performance at $v'$
  return $\{v', f'\}$
```

Figure 5: The search algorithm

```
tunable Kz_0, Kx_0;
mappar (int k0 ..., int i0 ...) {
  fft1D_Y(d0[k0*kz_0;Kz_0][0:Ny][i0*Kx_0;Kx_0], coef);
}
tunable Kz_1, Ky_1, Kx_1;
mappar (int k1 ..., int j1 ..., int i1 ...) {
  transpose(d0[k1*Kz_1;Kz_1][i1*Kx_1;Kx_1][j1*Ky_1;Ky_1],
            d1[k1*Kz_1;Kz_1][j1*Ky_1;Ky_1][i1*Kx_1;Kx_1]);
}
tunable Kz_2, Kx_2;
mappar (int k2 ..., int i2 ...) {
  fft1D_Y(d1[k2*kz_2;Kz_2][0:Nx][i2*Kx_2;Kx_2], coef);
}
tunable Kz_3, Ky_3, Kx_3;
mappar (int k3 ..., int j3 ..., int i3 ...) {
  transpose(d1[k3*Kz_3;Kz_3][i3*Kx_3;Kx_3][j3*Ky_3;Ky_3],
            d2[k3*Kz_3;Kz_3][j3*Ky_3;Ky_3][i3*Kx_3;Kx_3]);
}
tunable Ky_4, Kx_4;
mappar (int j4 ..., int i4 ...) {
  fft1D_Z(d2[0:Nz][j4*Ky_4;Ky_4][i4*Kx_4;Kx_4], coef);
}
```

Figure 6: Loop nests from FFT3D

erence to $d2$ in the last loop nest, so loop fusion cannot increase the distance of any reuse. However, on Cell the performance is 5 times worse after fusion compared to no fusion. The performance degradation is caused by *the tunable mismatch penalty*, an important factor not captured by reuse distances.

Suppose for loop nests $\mathcal{L}_1$ and $\mathcal{L}_2$, the best tunable values are $\mathcal{V}_1, \mathcal{V}_2$ prior to fusion and $\mathcal{V}_1', \mathcal{V}_2'$ after fusion. The performance degradation of $runtime(\mathcal{L}_1, \mathcal{V}_1') + runtime(\mathcal{L}_2, \mathcal{V}_2') - (runtime(\mathcal{L}_1, \mathcal{V}_1) + runtime(\mathcal{L}_2, \mathcal{V}_2))$ is called *the tunable mismatch penalty*, where $runtime(\mathcal{L}, \mathcal{V})$ is the execution time of loop nest $\mathcal{L}$ with tunable values $\mathcal{V}$. The reason that $\mathcal{V}_1 \neq \mathcal{V}_1'$ $\mathcal{V}_2 \neq \mathcal{V}_2'$ is two-fold:

- Due to capacity constraints, the tunable values after fusion are reduced.
- The best *tunable shape* of one loop nest can be very different from the best tunable shape of another. When two loop nests are fused, two different shapes are combined, which means compromising on the best tunable shape of each individual loop nest.

The tunable mismatch penalty should be greater on software-managed memory hierarchies than on cache-based architectures because performance is less sensitive to tunable shape on cache-based architectures and because the best tunable values prior to fusion are usually close to the search boundary on software-managed memory hierarchies.

In most cases, we want tunables to be large to have more exploited reuse and to have larger transfer sizes. On a memory level with smaller capacity, the tighter capacity constraints will drive the tunables to have smaller values. Usually performance varies more rapidly with the tunables when the tunables are smaller. Thus the tunable mismatch penalty is greater on levels with smaller capacity.

We propose a fusion algorithm for software-managed memory hierarchies that is different from the traditional fusion algorithms targeting cache-based architectures:

- Without exploring the tunable space first, we can't measure the amount of degradation caused by tunable mismatch. Thus we perform loop fusion after exploring the tunable space. In fact, the knowledge gained when searching the tunable space is used to guide the selection of a fusion configuration.
- Since it is important to consider tunable mismatch, we can no longer make fusion decisions by looking only at the outermost loop level. Our algorithm considers multiple outermost loop levels in a single step.

5.1 The Fusion Algorithm

The term *fusion depth* is used throughout this section. At each algorithmic step, if only the outermost loop level is considered, the fusion depth is 1, if we consider two outermost loop levels, the fusion depth is 2, and so on. Profitability is not estimated from a static model of the targeted architecture, instead it is constructed from the profiling information collected when searching the tunable space. We focus on the problem of selecting *a loop order* for each loop nest and *a fusion configuration* to maximize the profitability of loop fusion.

A reuse pair is defined as a pair of references that touch overlapping memory regions.In this section, we consider inter-loop reuse pairs only. Our framework solves the fusion problem top-down, beginning with the top memory level, because fusion at high memory levels can create more fusion possibilities at low levels and bandwidth is scarcer at higher memory levels. The algorithm for a memory level is described below. First, for each reuse pair, the possibility of exploiting the reuse is checked by applying fusion multiple times, considering only the loops the pair of data references are in (Section 5.1.2). This step annotates each reuse pair with information that will help the later steps make their decisions. The fusions applied are reverted after the reuse pair is analyzed. The algorithm then generates a loop order for each loop nest (Section 5.1.4). Finally a weighted pair-wise fusion algorithm is applied (Section 5.1.5).

5.1.1 Multi-dimensional Loop Alignment

Loop alignment [1] is an iteration space transformation technique that aligns the iterations of two loops to remove backward true dependencies which make loop fusion illegal or to bring data reuse closer. Here, we are aligning multiple outermost loop levels of a pair of loop nests. *Tunable relations* (relationships between tunables of the two loop nests), and *iteration relations* are generated as a result of the loop alignment.

A basic iteration relation is written as:

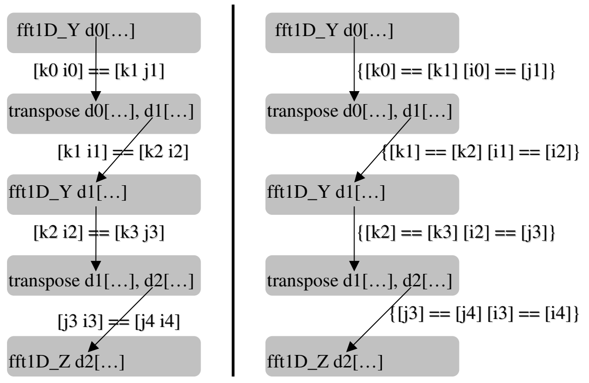

Figure 7: Annotated reuse pairs for FFT3D. Left: Cell, Right: a cluster of PCs. Edges represent reuse pairs and are annotated with the iteration relations.

$[i_1\_l_1\ i_2\_l_1\ \ldots\ i_N\_l_1]$==$[i_1\_l_2\ i_2\_l_2\ \ldots\ i_N\_l_2]$, where N is the fusion depth and $i_k\_l_j$ represents a loop level of loop nest j with loop variable i_k. The basic iteration relation says that loop levels $i_1\_l_j$ to $i_N\_l_j$ are the N outermost loop levels, loop $i_k\_l_1$ must be at the same loop position as $i_k\_l_2$, and loop order is not specified. *An iteration relation* is a set of basic iteration relations, which means any basic iteration relation in the set will work.

5.1.2 Annotating Reuse Pairs

For each reuse pair, we test whether it is possible to reduce the reuse distance such that the reuse can be exploited at the targeted memory level. Considering only the loops the reuse pair resides in, a series of fusions is performed until the reuse distance is smaller than the memory capacity or further fusion is illegal. The series of fusions, together with alignment information for each fusion, are annotated with the reuse pair and the fusion depth of each fusion is determined as follows:

- $k_1 = \max_k$ k-deep loop nest is fully permutable and k-deep fusion is legal.
- $k_2 = \min_{k \leq k_1}$ data touched by a single iteration of fused k-deep loop nest fits in the targeted memory level
- If such k_2 exists, use k_2, otherwise, use k_1 as fusion depth.

We use `FFT3D` (Figure 6) as an example and show how our fusion algorithm performs differently on Cell vs. on a cluster of PCs. k_1 is machine-independent and k_2 is specific to a machine. Since the capacity of a cluster node is much bigger than that of local store, k_2 is determined to be 2 for Cell and 1 for a cluster of PCs. The fusion depth is equal to k_2. In Figure 7, a reuse pair is shown as an edge between two references and the annotations are displayed along the edge. Only a single pair-wise fusion is required to exploit each reuse pair in our example.

5.1.3 Profitability

We consider penalties due to tunable mismatch and benefits gained by exploiting the inter-loop reuse, when calculating the profitability of fusing a pair of loop nests. We first determine the tunable values of the fused loop nest, and then calculate *MismatchPenalty* (Table 1). Data transfer operations can be (partially) removed as a result of fusing the pair. A penalty term is applied for transfer operations that are eliminated from *misaligned reuse* (see Section 5.1.6).

IN:	(L_i, T_i)	=	An original loop nest before any loop fusion is applied and its tunables
	P_i	=	Set of data points evaluated when exploring the tunable space of L_i
	$(\mathcal{L}_j, \mathcal{T}_j)$	=	A loop nest at the current state and its tunables
	$\mathcal{S}_j$	=	$\{(L_{jk}, R_{jk}), k \in [1, N_j]\}$
			L_{jk}: The set of original loop nests that $\mathcal{L}_j$ is generated from
			R_{jk}: Tunable relation that maps $\mathcal{T}_j$ to T_{jk}
	Candidate Pair	=	$(\mathcal{L}_1, \mathcal{T}_1)$ $(\mathcal{L}_2, \mathcal{T}_2)$
	$\mathcal{R}(\mathcal{T}_1 \rightarrow \mathcal{T}_2)$	=	Tunable relation given by alignment of $\mathcal{L}_1$ with $\mathcal{L}_2$
OUT:	$MismatchPenalty$		
ALG.:	$(\mathcal{V}_1, \mathcal{V}_2)$	=	The best values of $\mathcal{T}_1$ and $\mathcal{T}_2$
	$RunTime(L_i, V_i)$	=	Neighborhood interpolation of data points in P_i
	$RunTime(\mathcal{L}_j, \mathcal{V}_j)$	=	$\sum_{(L_{jk}, R_{jk}) \in \mathcal{S}_j} RunTime(L_{jk}, R_{jk}(\mathcal{V}_j))$
	$(\mathcal{V}_1{}', \mathcal{V}_2{}')$	=	Values of $\mathcal{T}_1$ and $\mathcal{T}_2$ minimizing $RunTime(\mathcal{L}_1, \mathcal{T}_1) + RunTime(\mathcal{L}_2, \mathcal{T}_2)$
			$\forall(\mathcal{T}_1, \mathcal{T}_2) \in \mathcal{R}$ and $(\mathcal{T}_1, \mathcal{T}_2)$ is feasible for the fused loop nest
	$MismatchPenalty$	=	$RunTime(\mathcal{L}_1, \mathcal{V}_1{}') + RunTime(\mathcal{L}_2, \mathcal{V}_2{}') - (RunTime(\mathcal{L}_1, \mathcal{V}_1) + RunTime(\mathcal{L}_2, \mathcal{V}_2))$

Table 1: Algorithm to calculate the tunable mismatch penalty

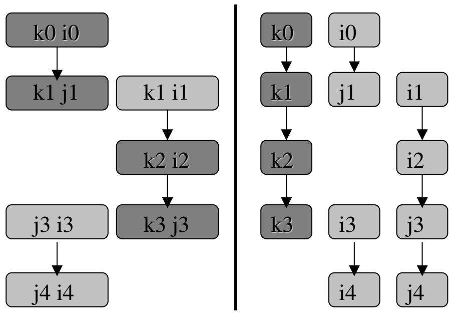

Figure 8: Loop order graph for FFT3D. Left: Cell, Right: a cluster of PCs. Edges mean alignment constraints between the node pair in order to exploit reuse.

Finally the profitability is updated to be the transfer time reduced minus $MismatchPenalty$. Due to the smoothness of tunable search space on software-managed memory hierarchies, neighborhood interpolation is used to estimate performance at un-evaluated points.

5.1.4 Selecting Loop Order

We create *a loop order graph* from the annotations generated at Section 5.1.2. Nodes are created from the iteration relations associated with each reuse pair. For example, from the iteration relation $[k_0\ i_0]$==$[k_1\ j_1]$, two nodes $[k_0\ i_0]$ and $[k_1\ j_1]$ are constructed. An edge is added between two nodes if there exists a reuse pair that requires the alignment of the two nodes and all such reuse pairs are associated with the edge. Edge weight is defined as the profitability of fusing the pair with the alignment defined by the edge. The loop order graphs of FFT3D are shown in Figure 8. Nodes belonging to the same loop nest are displayed at the same row.

From the loop order graph, we select a single node for each loop nest to maximize the potential profitability of loop fusion. And the profitability of loop fusion is measured as sum of weight on edges that can be realized with the set of selected nodes. For FFT3D, the set of selected nodes are highlighted by darker colors in Figure 8, and no node is chosen for the last loop nest because its loop order does not affect the profitability of loop fusion.

5.1.5 Performing Fusion

After a loop order is selected for each loop nest, a pairwise fusion algorithm is used. At each step, the algorithm greedily picks the candidate pair with the biggest profitability to fuse. If the profitability is calculated as described in Section 5.1.3, we name the approach *model-based fusion*, since no further program evaluations are required when applying the fusion algorithm. If for each candidate pair, we empirically determine the tunables of the fused loop nest using our search algorithm and update the profitability with the actual performance gain, this approach is called *search-based fusion*. We compare the performances of these two approaches in the evaluation section.

5.1.6 Misaligned Reuse

Given a reuse pair <(R_s, L_s) (R_d, L_d)>, where $\vec{vs}$ of L_s is aligned with $\vec{vd}$ of L_d, if $footprint(R_s, L_s, \vec{vs})$ overlaps with but is not the same as $footprint(R_d, L_d, \vec{vd})$, we say the reuse pair is *misaligned*. On cache-based architectures, the compiler is responsible for reducing the reuse distance and reuse is automatically exploited by the cache once the distances are short enough. For software-managed memory hierarchies, there is a separate namespace for each memory level, and the compiler is responsible for renaming the data such that misaligned reuse can be explicitly exploited.

For misaligned *read-after-read* reuse, a data object covering union of $footprint(R_s, L_s, \vec{vs})$ and $footprint(R_d, L_d, \vec{vd})$, is created in the lower memory level. Then R_s and R_d are updated to be relative to the newly-created data.

The handling of misaligned *read-after-write* (RAW) reuse is more complicated. We use two loop nests from the SUmb benchmark as an example, shown in Figure 9, where the RAW reuse of array r is misaligned at data dimension y. The left column of figure 10 shows the access patterns of the two references to array r prior to fusion. To make fusion legal, we can either *duplicate computation* at R_s to generate aligned reuse or shift the region produced by R_s. The right column of Figure 10 shows the access patterns after duplicating computation along y dimension. The case where the first loop nest is shifted is displayed in the middle column. The dark rectangles are regions of array r that are needed at R_d, but are generated at previous iterations. That means some of the inter-loop reuse is converted to loop-carried reuse.

Reuse can be exploited if the overlapping region stays at the same physical location and is not evicted. We use *a circular layout* to exploit reuse carried by the innermost loop if possible. *A circular layout* is the mapping from logical data to physical location that wraps around at boundary of the physical region. For data used by external leaf tasks, its lay-

```
tunable Kz_0, Ky_0, Kx_0;
mappar (int k0 ..., int j0 ..., int i0 ...) {
   inviscidEta(r[k0*Kz_0;Kz_0][j0*Ky_0;Ky_0][i0*Kx_0;Kx_0],
               ...); //update r
}
//different blocking for boundary sub-blocks
tunable Kx_1, Ky_1, Kx_1;
mappar (int k1 ..., int j1 ..., int i1 ...) {
   diffEta(r[k1*Kz_1;Kz_1]
            [max(0,j1*Ky_1-2):min((j1+1)*Ky_1+2, Ny);Ky_1+4],
            [i1*Kx_1;Kx_1], ... );
}
```

Figure 9: Loop nests from SUmb with misaligned reuse.

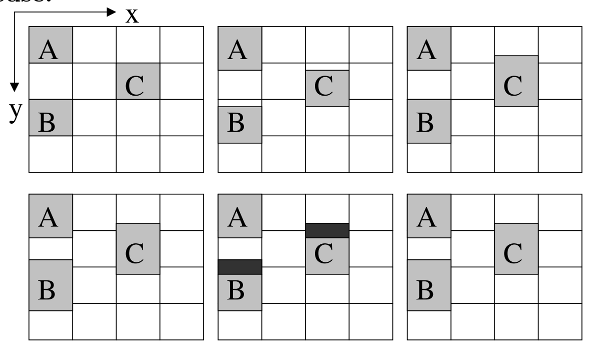

Figure 10: Example of misaligned reuse. Top(Bottom) row is the data accessed by reference r of the first(second) loop nest at iterations A B and C. Left: prior to fusion; Middle: shift the first loop nest to make fusion legal; Right: duplicate computation along y dimension.

out at *the lowest dimension* should be contiguous to enable vector operations, thus a circular layout is not supported for the lowest dimension due to violation of contiguity.

We can discard the loop-carried reuse by always loading from the higher memory level, which means reference R_s stores part of its footprint back to the higher memory level, then reference R_d loads from the higher memory level. And we call it *loading method*. Since loading a small section along the lowest data dimension requires transfers of small sizes, and the achieved bandwidth is low for small transfers, we do not recommend loading method for the lowest dimension.

For each misaligned data dimension of RAW reuse, we can choose one method out of the three methods described above (i.e. loading method, a circular layout and computation duplication). Since a circular layout has the least amount of overhead, we give it the highest priority. Due to the reasons described above, for the lowest dimension, we consider computation duplication before loading method. For other data dimensions, we give loading method higher priority.

6. EVALUATION

In this section, we present an evaluation of our method using some preliminary experimental results. We implemented several benchmarks (described in Table 2) in Sequoia. The benchmarks were executed on three different platforms:

- **Cell**: a single 3.2GHz Cell processor with 8 SPEs and 1GB of XDR memory in an IBM BladeCenter. 16MB pages are used to reduce TLB misses.
- **Cluster of PCs**: a cluster of 16 nodes each with dual 2.4GHZ Intel P4 Xeon processors and 1GB of main memory. The nodes are connected with Infiniband 4X SDR PCI-X HCAs and only one processor is utilized per node.
- **Cluster of PS3s**: a cluster of 2 nodes, each is a Sony Playstation3 with 256MB of memory, which uses a Cell processor with 6 SPEs. Nodes are connected with GigE.

FFT3D	Fast Fourier transform of a complex 256^3 dataset (128^3 for Cluster of PS3s).
SGEMM	BLAS L3 sgemm, multiplying matrices of size 4096x4096.
CONV2D	Convolution of a 9x9 filter with a 8192x4096 input signal.
SUmb	Stanford University multiblock. A massively parallel flow solver which uses a multi-block structured meshing approach. It has 13 kernels and 39 tunables on a machine with a 2-level memory hierarchy.

Table 2: Benchmarks used for evaluation (single-precision)

We implemented two versions of the `FFT3D` benchmark that differ in how to decompose the problem. The example used in Section 5 is named `FFT3D V2` here, and it transposes the data twice and performs three 1D FFTs in a *(Y,X,Z)* dimension order. The other version, which is called `FFT3D V1` here, is a 3-transpose version and it performs three 1D FFTs in a *(Y,Z,X)* dimension order.

Our leaf tasks utilize the fastest implementations available. For x86, we use FFTW [15] and the Intel MKL and for Cell or PS3, we use the IBM SPE matrix library. All other leaf tasks are our own best effort implementations, hand-coded in SSE or Cell SPE intrinsics. Other than the Sequoia source codes and the codes for the leaf tasks, the user only provides necessary constraints on tunables at the leaf level to express correctness conditions for the external leaf tasks.

6.1 The Achieved Raw Performance

Table 3 compares the raw performance achieved by our tuning framework using our search algorithm that evaluates 30 data points followed by our model-based fusion algorithm, to the performance of the best-available hand-tuned version coded in Sequoia ([20]). Applying our fusion algorithm may create the possibility of fusion of leaf tasks (i.e. combining leaf tasks to improve utilization of the leaf processor). With fusion of leaf tasks, the performance achieved by our tuning framework is similar to or better than the hand-tuned version. For `FFT3D` on Cell, the best performance achieved by our tuning framework is 57 GFLOPS, which is better than the 39 GFLOPS reported for version 3.2 of FFTW [15], the 54 GFLOPS achieved by the hand-tuned version [20], and the 46.8 GFLOPS reported for IBM's large FFT implementation [7]. `FFT3D` on Cluster of PCs achieves 5.5 GFLOPS with kernel fusion, which is a little better than the 5.3 GFLOPS achieved by FFTW 3.2 alpha 2 using the same system configuration and dataset size. The Intel MKL provides support for execution of SGEMM on a cluster of workstations. Compared with the 101 GFLOPS achieved by Intel Cluster MKL on the same cluster configuration, our performance of 92.4 GFLOPS is within 9%. Since `SUmb` is too complex for hand-tuning, no performance data is available for hand-tuned version.

		CONV2D	SGEMM	FFT3D	SUmb
Cell	auto	99.6	137	42(57)	12.1
	hand	85	119	54	
Cluster of PCs	auto	26.7	92.4	4.4(5.5)	2.2
	hand	24	90	5.5	
Cluster of PS3s	auto	20.7	33.4	0.57	0.63
	hand	19	30	0.36	

Table 3: Measured raw performance of benchmarks: the tuning framework vs. hand-tuned version in GFLOPS. For FFT3D, performance with fusion of leaf tasks is displayed in parentheses.

Our approach attempts to maximize the utilization of communication bandwidth by intelligently setting the values of tunables and our fusion algorithm aims to reduce the frequency of memory accesses and communication. Figure 11 shows the breakdown of execution time and the utilization of communication bandwidth for our 4 applications across 3 platforms. On Cell, the sustained bandwidth is the total execution time divided by number of bytes transferred between memory and LS. On both Cluster of PCs and Cluster of PS3s, the sustained bandwidth is calculated as the total execution time divided by number of bytes communicated inter-node.

On Cell, `CONV2D` and `SGEMM` are compute bound and spend 97% of execution time running kernels. For compute limited benchmarks, the only way to improve performance would be to further tune the kernels. `SUmb` is bandwidth limited, waiting on memory transfers 25% of the time, and it achieves 16.6GB/s, which is a high utilization of the memory bandwidth, relative to the optimal DRAM throughput of our Cell system (25.6GB/s). `FFT3D` strikes a balance between computation and memory. It spends 94% of execution time running leaf tasks and 6% of time waiting on memory transfers. The above 4 applications either fully utilize the SPEs' arithmetic resources or achieve a high sustained memory bandwidth.

On Cluster of PCs, the 4 applications spend between 10 and 42 percent of their time waiting for transfer operations to finish. `SGEMM` is sufficiently compute intensive that it only spends 10% of its time waiting on transfers, while the other 3 applications are limited by the interconnect performance. `FFT3D` achieves the highest sustained communication bandwidth (650 MB/s), while `CONV2D` has the lowest bandwidth since it reads the boundary of a region from neighboring nodes, causing remote transfers with small sizes.

On Cluster of PS3s, all 4 applications spend a significant amount of time, between 63 and 95 percent, waiting for transfers. So all applications are limited by transfer operations between $M2$ and $M1$. For each transfer operation from a virtual level, a contiguous block is usually constructed for the requested data at the destination node, with `memcpys` to move the portion owned by the node and inter-node communication to transfer data from remote nodes. Other than the temporary data blocks, memory space is also required at each node to store the owned portion of each distributed array. The speed of GigE interconnect, the limited available memory and the overhead of `memcpys` drive the transfers between $M2$ and $M1$ slow.

6.2 Evaluation of the Tunable Space Search

To evaluate the performance of our search algorithm on Cell, we use the best result from an exhaustive search on a coarse grid as the baseline. The number of program evaluations required by the exhaustive search is shown in the last column of Table 4. On Cluster of PCs and Cluster of PS3s,

	Number of Tunable Groups Per Level	Number of Tunables Per Level	Number of Search Points
FFT3D V1	6	15	361
FFT3D V2	5	12	363
SGEMM	1	3	6546
CONV2D	1	2	566
SUmb	13	39	486

Table 4: Search space properties. The last column is the maximal number of search points across all tunable groups on Cell.

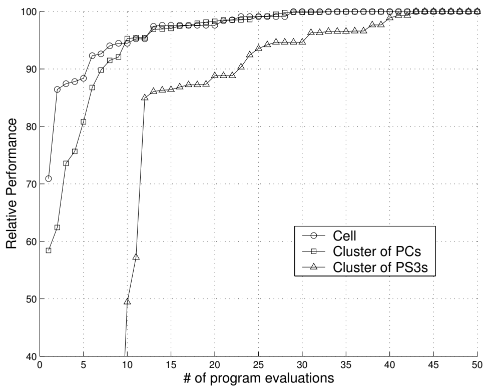

Figure 12: Convergence rate of our search algorithm on Cell, Cluster of PCs and Cluster of PS3s.

the baseline is the best performance achieved by 50 evaluations. Figure 12 shows that our search algorithm converges quickly on Cell, 90% performance achieved in 6 evaluations. On Cluster of PCs and Cluster of PS3s, we observe slower convergence compared to targeting Cell. We believe it is due to: First, the search spaces for these two targets, after pruned by the capacity constraints, are larger than the search space on Cell; Second, the search space on Cell is relatively smoother.

We also studied the performance of our search algorithm on each tunable group. Even though there are 26 tunable groups across our benchmarks, since some tunable groups are from multiple instantiations of the same loop nest, only 19 tunable groups are unique. Figure 13 shows that after x program evaluations, how many tunable groups (y) achieve 70%, 80%, 95% or 99% performance relative to the baseline. Our search algorithm works well on Cell: in 14 evaluations, all tunable groups achieve 85% performance and 16 out of 19 tunable groups achieve 99% performance. An additional 15 evaluations are needed for the other 3 tunable groups to reach 99% performance.

We achieve good performance quickly on all three platforms due to:

- the smoothness of the search space.
- the relative insensitivity to the problem size. Thus less correlation is observed between tunable values at a level and tunable values at its child level.
- the specialization of the search algorithm for software-managed memory hierarchies, such as how to select the initial point, how to set a non-square grid, and how to handle the case that the current search point is close to the boundary.

6.2.1 *Comparison of Search Algorithms on Cell*

The performance of random search is shown in Figure 14 and it makes progress at a much slower rate than our

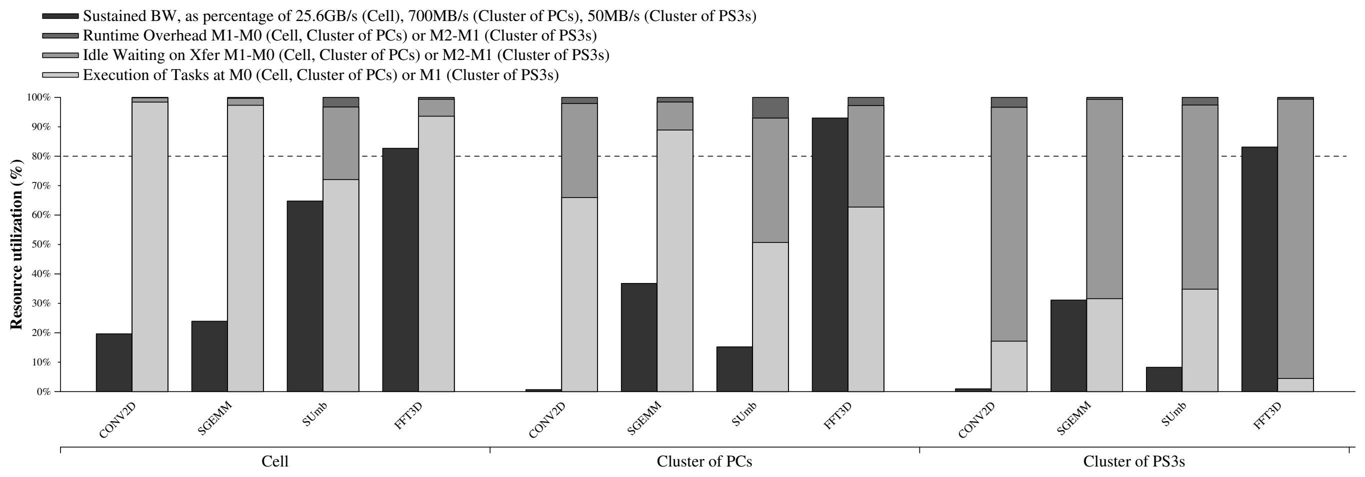

Figure 11: Utilization of communication bandwidth (left bar) and execution time breakdown (right bar).

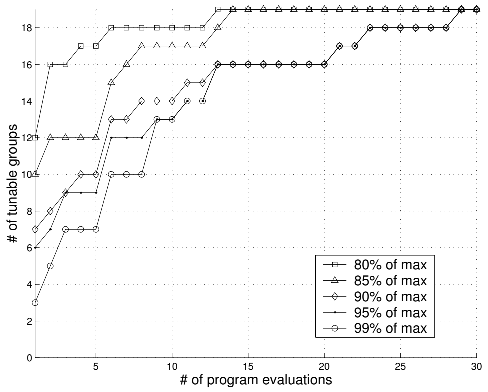

Figure 13: Performance of our search algorithm across tunable groups, on Cell.

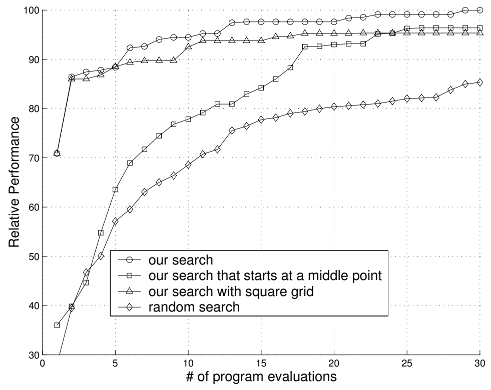

Figure 14: Comparing different search algorithms on Cell.

search algorithm. To achieve 80 percent performance, random search requires only 3 evaluations for CONV2D, but 139 evaluations for one tunable group of `SUmb`.

To evaluate sensitivity to the initial point, an alternative approach first finds the maximal square tunables, then the value of each tunable is halved to get an initial point that is in the middle of the search space. This alternative approach is often used when empirically searching tile sizes on cache-based architectures. We observe that with the alternative approach more evaluations are required to achieve a certain performance, as shown in Figure 14. Since the middle point (P_a) often performs worse than the initial point (P_b) chosen by our algorithm, it takes up to 12 evaluations to reach the performance of P_b if we start the algorithm with P_a.

If we use a square grid, the search algorithm shows a slower convergence. At coarse grid levels there are fewer points with a square grid than with a non-square grid, thus less progress is made at those levels.

6.3 Evaluation of the Integrated Fusion Algorithm

		FFT3D V1	FFT3D V2	SUmb
Cell	search-based	2.26	1.92	1.27
	model-based	2.26	1.92	1.20
Cluster of PCs	search-based	1.0	1.1	1.21
	model-based	0.9	1.1	1.16
Cluster of PS3s	search-based	1.12	1.21	2.7
	model-based	1.12	1.21	2.3

Table 5: Performance improvement with loop fusion for FFT3D V1, FFT3D V2 and SUmb, compared to without fusion.

We evaluate the model-based and search-based fusion algorithms on `SUmb`, `FFT3D V1` and `FFT3D V2`, since those are the only applications that have multiple loop nests. Table 5 shows the performance improvement gained by our integrated fusion algorithm, with 30 points evaluated during the tunable search and without fusion of leaf tasks. On Cell a 2.26x performance is achieved for `FFT3D V1`, and we see a performance improvement of 92 percent and 20 percent for `FFT3D V2` and `SUmb` respectively. The performance of model-based fusion is close to that of search-based fusion, which suggests that our profitability model is quite accurate. A greedy pair-wise fusion algorithm that fuses the pair with the biggest gain at each step does not guarantee optimal performance. Global loop fusion can be formulated as a graph-partitioning problem [18, 10], where loops are divided into a sequence of partitions. Our profitability model targeting software-managed memory hierarchies, and our extension to handle multiple outermost loop levels are applicable to other fusion methods.

The fusion algorithm on Cluster of PCs does not achieve much performance gain, in fact, the performance degrades for `FFT3D V1`. Since we do not explicitly model the cache of each node as a machine level, there are cache interactions that are not captured by our model-based fusion algorithm.

Table 6 shows the sensitivity of model-based fusion to the number of points evaluated during the tunable search. With 30 points evaluated, we achieve the highest performance of 12.1 GFLOPS. The performance degrades by 12 percent when only 10 points are evaluated. Since model-based fusion uses neighborhood interpolation to estimate the run time of a loop nest with given tunable values, the accuracy of its profitability model varies with how many points are evaluated during the tunable search.

	30	25	20	15	10
no fusion	10.1	10.0	9.73	9.77	9.7
fusion	12.1	12.1	11.3	11.4	10.6

Table 6: Performance in GFLOPS without fusion and with model-based fusion for SUmb on Cell, varying number of points evaluated during the tunable search.

7. RELATED WORK

A number of empirically-tuned libraries deliver high performance for a range of architectures, such as FFTW [15], SPIRAL [30], ATLAS [35] and PhiPAC [4]. Both FFTW and SPIRAL use empirical techniques to choose among multiple implementations of the same problem. They recursively decompose a problem into simpler sub-problems using a set of rules, derived from mathematical properties of the signal processing algorithms. SPIRAL is more general than FFTW: it generates optimized code for a large class of signal transforms, while FFTW is a library for computing discrete Fourier transforms. PhiPAC and ATLAS both generate high performance matrix-matrix multiply by empirically searching a large space of parameter values. Our approach is different in that it does not exploit properties of any specific domain.

Empirically-based tuning methods have been used in general purpose compilers. In [17], tile sizes and unroll factors of nested loops are empirically searched, and [6] considers tiling for each memory hierarchy level (from registers as the lowest level to main memory as the highest level), empirically searching tiling and prefetching parameters. [33] uses direct search to explore the space of tile sizes and unroll factors. Unlike the previous work that targets cache-based architectures, our tuning framework targets software-managed memory hierarchies; in particular, our search algorithm is tailored to the characteristics of the search space of tunables on software-managed memory hierarchies.

A fusion algorithm incorporating loop alignment, loop interchange, and a data regrouping step after fusion is presented in [11]; their method considers the outermost loop level only at each step of the algorithm, and the selection of loop order is driven by number of fusions. Our integrated fusion algorithm differs in that it looks at multiple outermost loop levels at each algorithmic step and the profitability measurement considers mismatch of the tuning parameters. Several loop transformations including loop fusion and tiling are evaluated in [36], but only static models are used to determine profitability and tiling decisions are made after loop fusion and loop distribution.

In [31], a profitability model based on reuse analysis targeting cache-based architectures is presented for loop fusion, and several architectural parameters in the profitability model are empirically searched. When those parameters are changed, a different fusion configuration can be generated from a greedy fusion algorithm. This work is extended to tiling and fusion in [32]. With this approach less speedup is observed for more complex machines, since its effectiveness depends on how well the model matches the underlying architecture, to some degree. Our tuning framework calculates the profitability of a fusion configuration with information collected when exploring the search space of tunables.

Steady progress has been made in the past decade on dependence analysis, transformations, and code generation in the polyhedral model [34, 2, 27]. The cost function used to select a transformation must be simple to be tractable mathematically. In addition to model-based approaches, semi-automatic and search-based frameworks also exist [8, 19, 16, 29, 28]. The search space of finding a set of transformation coefficients for each statement can be huge, thus heuristics are used to bound the search space and to guide the search [16, 29, 28]. Our approach depends on the programmer to decompose a program and to pick parameters as tunables, while those information is automatically extracted from the program in the polyhedral model. The polyhedral model is applicable to affine loop nests only, but our framework works for non-affine programs as well.

8. CONCLUSION

We have presented a tuning framework that allows programs written in a portable language, Sequoia, to be automatically tuned for a wide range of machines with software-managed memory hierarchies. Our framework matches the decomposition strategies to the memory hierarchies, and uses a search algorithm, specialized to software-managed hierarchies by intelligently choosing the initial search point and using a non-square grid, that achieves good performance quickly due to the smoothness of the search space. Since tunable mismatch penalties are greater on software-managed hierarchies than on cache-based architectures, we apply a novel fusion algorithm that considers multiple outermost loop levels in a single step. The knowledge learned when searching the tunable space is used to guide the selection of a fusion configuration.

Our system, on top of a portable, structured language, with the profiling system matching the machine hierarchy and profiling results for each loop nest and each bulk operation, can provide valuable and easy-to-understand feedback to the user. The user can use the findings to deduce the bottlenecks of the application and to decide whether he/she should modify the Sequoia source code or external leaf implementations.

We have demonstrated the performance of our framework on a single Cell processor, on a cluster of Intel Xeon processors, and on a cluster of PS3s. Our framework gives similar or better performance than what is achieved by the best-available hand-tuned version coded in Sequoia.

9. REFERENCES

[1] R. Allen and K. Kennedy. *Optimizing Compilers for Mordern Architectures.* 2001.

[2] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic data movement and computation mapping for multi-level parallel architectures with explicitly managed memories. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 1–10, 2008.

[3] P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta. CellSs: A programming model for the Cell BE architecture. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, 2006.

[4] J. Bilmes, K. Asanovic, C.-W. Chen, and J. Demmel. Optimizing matrix multiply using phipac: a portable high-performance ansi-c coding methodology. In *Proceedings of the 1997 ACM International Conference on Supercomputing*, 1997.

[5] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3):777–786, 2004.

[6] C. Chen, J. Chame, and M. Hall. Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy. In *CGO '05: Proceedings of the international symposium on Code generation and optimization*, pages 111–122, 2005.

[7] A. Chow, G. Fossum, and D. Brokenshire. A programming example: Large FFT on the Cell Broadband Engine, 2005.

[8] A. Cohen, M. Sigler, S. Girbal, O. Temam, D. Parello, and N. Vasilache. Facilitating the search for compositions of program transformations. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 151–160, 2005.

[9] W. J. Dally, P. Hanrahan, M. Erez, T. J. Knight, F. Labonte, J. Ahn, N. Jayasena, U. J. Kapasi, A. Das, J. Gummaraju, and I. Buck. Merrimac: Supercomputing with streams. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, page 35, 2003.

[10] C. Ding and K. Kennedy. The memory bandwidth bottleneck and its amelioration by a compiler. In *IPDPS '00: Proceedings of the 14th International Symposium on Parallel and Distributed Processing*, page 181, 2000.

[11] C. Ding and K. Kennedy. Improving effective bandwidth through compiler enhancement of global cache reuse. In *Parallel and Distributed Processing Symposium., Proceedings 15th International*, 2001.

[12] A. Eichenberger, J. O'Brien, K. O'Brien, P. Wu, T. Chen, P. Oden, D. Prener, J. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, M. Gschwind, R. Archambault, Y. Gao, and R. Koo. Using advanced compiler technology to exploit the performance of the Cell Broadband Engine architecture. *IBM System Journal*, 45(1), 2006.

[13] A. Eichenberger, K. O'Brien, K. O'Brien, P. Wu, T. Chen, P. Oden, D. Prener, J. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, and M. Gschwind. Optimizing compiler for the Cell processor. In *Proceedings of the 2005 International Conference on Parallel Architectures and Compilation Techniques*, September 2005.

[14] K. Fatahalian, T. J. Knight, M. Houston, M. Erez, D. R. Horn, L. Leem, J. Y. Park, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia: Programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, 2006.

[15] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. special issue on "Program Generation, Optimization, and Platform Adaptation".

[16] G. Fursin. A heuristic search algorithm based on unified transformation framework. In *ICPPW '05: Proceedings of the 2005 International Conference on Parallel Processing Workshops*, pages 137–144, 2005.

[17] G. Fursin, M. O'Boyle, and P. Knijnenburg. Evaluating iterative compilation. In *Proc. Languages and Compilers for Parallel Computers (LCPC)*, pages 305–315, 2002.

[18] G. R. Gao, R. Olsen, V. Sarkar, and R. Thekkath. Collective loop fusion for array contraction. In *1992 Workshop on Languages and Compilers for Parallel Computing*, number 757, pages 281–295, New Haven, Conn., 1992. Berlin: Springer Verlag.

[19] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *Int. J. Parallel Program.*, 34(3):261–317, 2006.

[20] M. Houston, J. Y. Park, M. Ren, T. J. Knight, K. Fatahalian, A. Aiken, W. J. Dally, and P. Hanrahan. A portable runtime interface for multi-level memory hierarchies. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2008.

[21] U. J. Kapasi, S. Rixner, W. J. Dally, B. Khailany, J. H. Ahn, P. Mattson, and J. D. Owens. Programmable stream processors. *IEEE Computer*, August 2003.

[22] T. Kisuki, P. M. W. Knijnenburg, and M. F. P. O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. In *IEEE PACT*, pages 237–248, 2000.

[23] P. M. W. Knijnenburg, T. Kisuki, and M. F. P. O'Boyle. Iterative compilation. pages 171–187, 2002.

[24] P. Mattson. *A Programming System for the Imagine Media Processor*. PhD thesis, Stanford University, 2002.

[25] M. D. McCool. Data-parallel programming on the Cell BE and the GPU using the RapidMind development platform. In *GSPx Multicore Applications Conference*, 2006.

[26] D. Pham, S. Asano, M. Bolliger, M. N. Day, H. P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, and K. Yazawa. The design and implementation of a first-generation CELL processor. In *IEEE International Solid-State Circuits Conference*, 2005.

[27] S. Pop, A. Cohen, C. Bastoul, S. Girbal, P. Jouvelot, G.-A. Silber, and N. Vasilache. Graphite: Loop optimizations based on the polyhedral model for gcc. In *Proceedings of the 4th GCC Developper's summit*, 2006.

[28] L.-N. Pouchet, C. Bastoul, J. Cavazos, and A. Cohen. Iterative optimization in the polyhedral model: Part ii, multidimensional time. In *PLDI '08: Proceedings of the ACM SIGPLAN 2008 conference on Programming language design and implementation*, 2008.

[29] L.-N. Pouchet, C. Bastoul, A. Cohen, and N. Vasilache. Iterative optimization in the polyhedral model: Part i, one-dimensional time. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, pages 144–156, 2007.

[30] M. PÃijschel, B. Singer, J. Xiong, J. Moura, J. Johnson, D. Padua, M. Veloso, and R. Johnson. Spiral: A generator for platform-adapted libraries of signal processing algorithms. 2004.

[31] A. Qasem and K. Kennedy. A cache-conscious profitablility model for empirical tuning of loop fusion. In *Proceedings of the 18th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2005)*, 2005.

[32] A. Qasem and K. Kennedy. Profitable loop fusion and tiling using model-driven empirical search. In *ICS '06: Proceedings of the 20th annual international conference on Supercomputing*, pages 249–258, 2006.

[33] A. Qasem, K. Kennedy, and J. Mellor-Crummey. Automatic tuning of whole applications using direct search and a performance-based transformation system. *J. Supercomput.*, 36(2):183–196, 2006.

[34] P. S. Uday Bondhugula, J. Ramanujan. A practical automatic polyhedral parallelizer and locality optimizer. In *PLDI '08: Proceedings of the ACM SIGPLAN 2008 conference on Programming language design and implementation*, 2008.

[35] R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, 1998.

[36] M. E. Wolf, D. E. Maydan, and D.-K. Chen. Combining loop transformations considering caches and scheduling. In *MICRO 29: Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 274–286, 1996.