Open64 on MIPS: porting and enhancing Open64 for Loongson II

Zhou Shuchang, Liu Ying, Lu Fang, Yin Le, Huang Lei, Li Shuai, Ma Chunhui, Gao Zhitao, Lian Ruiqi
Key Laboratory of Computer System and Architecture
Institute of Computing Technology, Chinese Academy of Sciences
Beijing, China
{*zhoushuchang,liuying2007,flv,yinle,leihuang,lishuai,machunhui,gaozhitao,lianruiqi*}*@ict.ac.cn*

***Abstract*—Loongson II is a MIPS III-compatible platform with various modern features, including a state-of-art memory subsystem. In the process of porting and enhancing Open64 on Loongson II, we observe that some optimizations lacking or being rudimentary in Open64, like edge profiling and array contraction, prove crucial for performance. We also develop LIDO, Locality Inspired Data Optimizer, to facilitate the optimization of data structure by source-to-source transformation. In orchestra of the powerful optimization framework of Open64, our work results in 28.5%/78.4% performance improvement in SPEC2000INT/FP over latest GCC on Loongson 2F.**

***Keywords*-Open64, Loongcc, Pathscale, ORC, MIPS, Loongson II, LIDO, CIL, source-to-source transformation**

I. Introduction

MIPSPro was made open source except its high performance MIPS backend under the name of Pro64 in 2000. In later developments, it evolved into Open64 and has been extended to various platforms, ranging from GPU to embedded devices. However, an open source port on MIPS is apparently missing from the picture. In addition, several trends in hardware development, including the widening gap between processor computation speed and memory system bandwidth and latency, greatly impact compiler optimization techniques. Consequently, a back-porting to MIPS is more challenging than it seems. These two factors influence our development of Loongcc, an Open64 branch for Loongson II.

From our experience of adapting Open64 to Loongson II, we observe that the memory system remains to be the bottleneck, especially for SPEC2000FP. In particular, the optimizations on spatial locality and array contraction prove very effective. In comparison, the source of speedup for SPEC2000INT is more diversified, consisting of contributions from feedback-directed optimizations, improved delay-slot filling, extended instruction set, and enhanced instruction selection and scheduling. We give an analysis of performance in Section III-B.

We build Loongcc by merging front-end and middle-end of Pathscale compiler with the ORC-based backend for Loongson II developed by our team[1]. To perform various data structure optimizations, we develop LIDO, a source-to-source transformation tool. An overview of our work is in Figure 1.

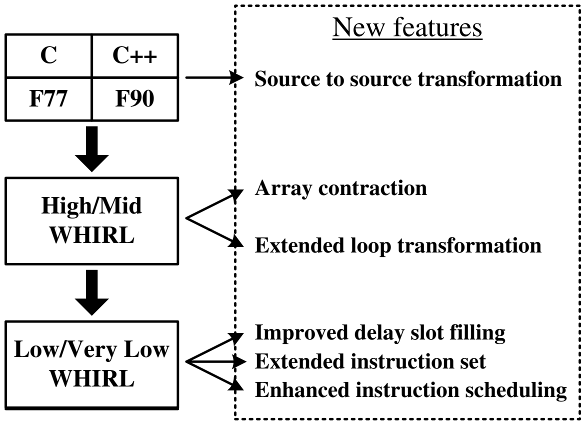

C
C++
F77
F90
High/Mid WHIRL
Low/Very Low WHIRL
New features
Source to source transformation
Array contraction
Extended loop transformation
Improved delay slot filling
Extended instruction set
Enhanced instruction scheduling

Figure 1: Overview of features of Loongcc

The rest of paper is organized as follows. Section II gives some background on the target platform we used. Section III describes our porting process and enhancement of Open64. Section IV describes LIDO. Section V overviews related work.

II. Background

The Loongson 2F processor we use is a 64-bit, Out-Of-Order, 4-issue, MIPS III-compatible platform[4]. It has a memory managing unit supporting DDR2 memory, 64K/64K Data/Instruction L1 caches, and a 512K L2 cache, all on-chip. The memory system of Loongson 2F supports load speculation, store fill buffers and non-blocking cache access to improve the bandwidth and latency. Besides support of full MIPS-III instruction set, Loongson 2F has a few extension instructions aiming to improve the efficiency of pipelines, including three-register-format multiplication, division and modulo operation. We perform our tests on a 800MHz Loongson 2F with 512MB DDR2 operating at 533Mhz.

III. Porting and Enhancing Open64

A. Porting

We choose Pathscale compiler[2], the branch of Open64 targeting X86-64, as our basis of development for front-end and middle-end, also incorporating modules from the sibling branch ORC [3] and Open64 itself. We then merge the modules from Pathscale with the backend developed for Loongson 2E here at ICT as starting point. Thanks to the clean hierarchy of WHIRL intermediate language and modularity of Open64, the porting process mainly involves replacing modules of CG phase in Pathscale compiler with ORC-based Loongson modules, besides fixing a few glitches scattered around. Table I gives an outline of our porting process.

Module	Changes
Machine model files	A model for MIPS is constructed from scratch as ORC has an incompatible scheme.
Front-end	Support for MIPS N32/64 ABI is included.
Middle-end	A few machine dependent parts in WOPT and LNO are fixed.
Backend	For simplicity, we switch to IGLS scheduling and drop region based compilation framework.
Profiling	Whirl profiling from Pathscale is preserved. Edge profiling from ORC-based backend is incorporated.

Table I: Summary of porting process of Loongcc

Throughout the porting process, we perform regression tests to ensure that the augmented parts do not impair the initial support for X86-64. All new modifications are wrapped up in proper macros.

B. Performance

1) Comparison with GCC: Figure 2 gives a comparison of performance between GCC and Loongcc. The benchmark we use is SPEC2000 as support for SPEC2006 is still under work. GCC *base* refers to a GCC branch released by STMicroelectronics that is developed specifically for Loongson 2E/2F. GFortran of the corresponding version is used to process the Fortran examples. The flag we use is ”-O3 -march=loongson2f -mtune=loongson2f’, the highest optimization level with tuned parameters for Loongson 2F. GCC *peak* refers to the highest score selected from the afore mentioned GCC branch, GCC 4.3 and GCC trunk 4.4. The flags used for GCC *peak* allow feedback-directed optimizations and turn on special optimizations like matrix reorganization. For Loongcc *base* we use ”-O3 -ipa” flag, the highest optimization level with inter-procedural optimization, but without profiling. Loongcc *peak* is Loongcc with heavily tuned flags. In all, Loongcc *base* outperforms GCC *base* by 13%/35%; Loongcc *peak* outperforms GCC *peak* by 28%/78%; Loongcc *peak* outperforms Loongcc base by 26% and 52%, in SPEC2000INT/FP respectively. We will analyze the performance gains below.

2) Performance analysis of SPEC2000INT/FP: The performance gain for SPEC2000INT can be classified into several categories, as shown in Table II. Since WHIRL profiling and edge profiling affect each other, only the sum of gain is given here. Figure 3 illustrates the composition. As interference between different optimizations can not be completely eliminated, we approximate the effect of an optimization by measuring the amount of degrade of performance when that optimization is disabled.

Optimization	Effect on the generated code
Feedback-directed optimizations by WHIRL and edge profiling	Frequency information facilitate various optimization heuristics.
Tweaking optimization flags and parameters	Parameters controlling optimizations are adapted for Loongson.
Improved delay-slot filling	The delay-slot module is enhanced to allow selection of instruction from consequent basic blocks in speculative manner. The main work is in fixing bugs and enriching the CG dependency graph to support such speculations.
Enhanced instruction scheduling	Memory access instructions are reordered to reduce stalls. Switching from backward-direction scheduling to forward-direction scheduling yields 8% performance improvement for *gap*.
Stride prefetch based on profiling	The pseudo-prefetch instruction in Loongson II is exploited to reduce latency of memory accesses. Test cases abundant in regular-stride memory accesses like *mcf* benefit most from this optimization, with its ratio increases by about 27%. Applying prefetching to *gap* and *parser* improves them by 4%/6.3% respectively.
Other optimizations	Optimization of global data access[5], use of conditional move instruction, and peephole optimizations also improve performance.

Table II: categories of performance gain in Loongcc peak over Loongcc base in SPEC2000INT

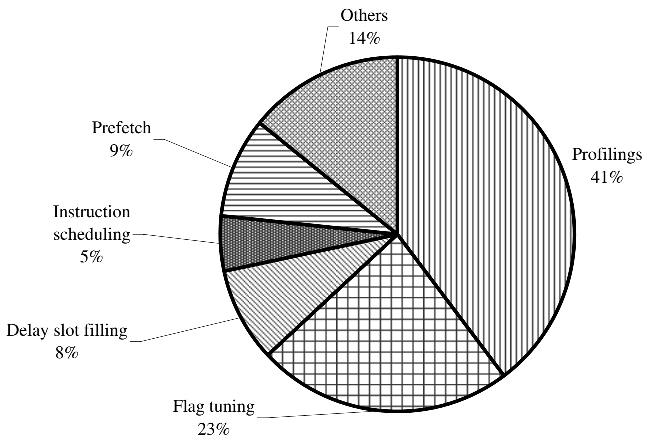


Figure 3: Composition of performance gain in Loongcc peak over Loongcc base in SPEC2000INT

Profiling proves to be the most effective optimization for SPEC2000INT. Both WHIRL profiling and edge profiling

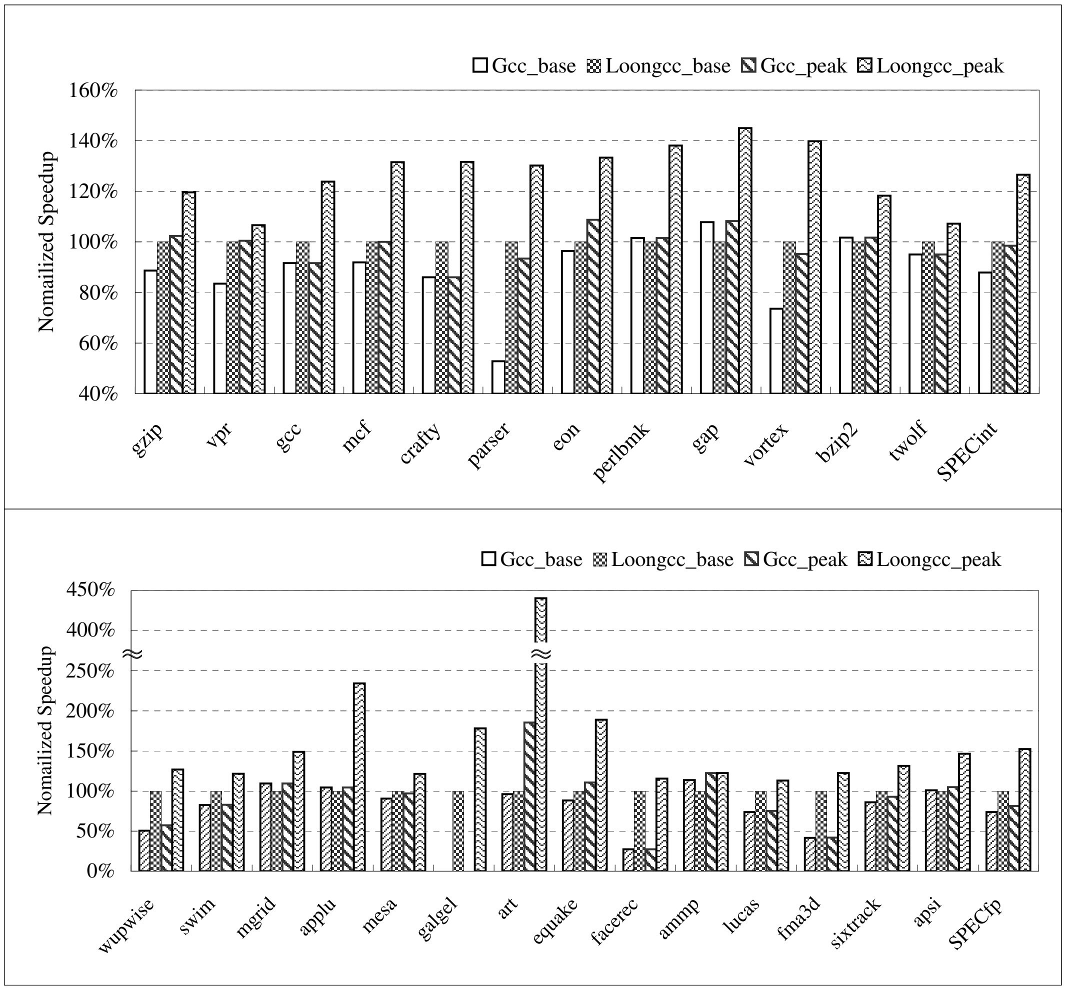


Figure 2: Comparison of performance of Loongcc and GCC on Loongson 2F. No GCC score for *galgel* can be produced here as we fail to produce correct code on Loongson 2F with GCC.

collects the execution counts by instrumentation. However, accuracy of frequency information collected by WHIRL profiling is affected by transformations of WHIRL tree in consequent phases. Edge profiling aims to improve the accuracy by performing instrumentation as late as possible, namely in code generation phase.

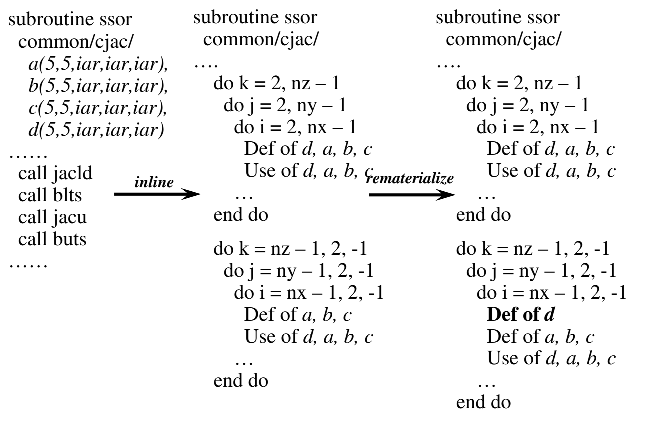
In comparison to SPEC2000INT, the performance gain for SPEC2000FP is concentrated in optimizations for memory accesses. For example, array contraction effectively improves *applu* and *galgel*, and locality optimizations improves *art* and *equake*. Probably due to the restrictiveness of the pseudo-prefetch instruction, we have not be able to improve SPEC2000FP ratio by stride prefetch yet. Performance of *facerec* and *fma3d* are more than doubled after switching to flush-to-zero mode. Below we describe array contraction module in Loongcc.

C. Array contraction in Loongcc

Besides developing LIDO, which we will cover in Section IV, we enhance several modules of Open64. In particular, array contraction improves *applu* by 29.7% and *galgel* by 35.4%.

Array contraction is a form of array scalarization that replaces the use of arrays with a few scalars or arrays of less number of dimensions[6][7]. This transformation both directly cuts the amount of memory accesses, and reduces the cache footprint so that data are more likely to be held in cache. We develop an array contraction module for Open64 which uncovers more optimization opportunities by aggressively applying loop transformations and performing ”rematerialization of array” when necessary[8]. Figure 4 gives an example of array contraction in *applu*.

After inlining the four subroutines and applying loop fusion, two loop nests are constructed. It is obvious that *a*, *b*, *c* and *d* are potential candidates for contraction. However, *d* prevents direct contraction because of the data dependency it introduces between the two loops. We apply rematerialization to address this problem. By recalculating the value of *d* at the second loop, we break the dependency and allow the four arrays to be contracted.



```
subroutine ssor
  common/cjac/
    a(5,5,iar,iar,iar),
    b(5,5,iar,iar,iar),
    c(5,5,iar,iar,iar),
    d(5,5,iar,iar,iar)
......
  call jacld
  call blts
  call jacu
  call buts
......
```

```
subroutine ssor
  common/cjac/
....
  do k = 2, nz – 1
    do j = 2, ny – 1
      do i = 2, nx – 1
        Def of d, a, b, c
        Use of d, a, b, c
        …
  end do
  do k = nz – 1, 2, -1
    do j = ny – 1, 2, -1
      do i = nx – 1, 2, -1
        Def of a, b, c
        Use of d, a, b, c
        …
  end do
```

```
subroutine ssor
  common/cjac/
....
  do k = 2, nz – 1
    do j = 2, ny – 1
      do i = 2, nx – 1
        Def of d, a, b, c
        Use of d, a, b, c
        …
  end do
  do k = nz – 1, 2, -1
    do j = ny – 1, 2, -1
      do i = nx – 1, 2, -1
        Def of d
        Def of a, b, c
        Use of d, a, b, c
        …
  end do
```

Figure 4: Array contraction in *applu* by rematerialization. After rematerialization, data dependency between the first and the second loop is broken, allowing all four arrays to be contracted into four much smaller two-dimension arrays.

IV. A Source-to-source Transformation Tool

A. Source-to-source transformation

Data locality refers to the phenomenon that data accesses are often performed at the same or related addresses. By exploiting data locality we can effectively reduce memory access latency and improve bandwidth[9][10]. In the effort to improve the data locality of program by transforming data structure, a family of transformations has been proposed, including structure field reordering, structure splitting, structure peeling, array flattening, and array transposition[11] [12][13][14]. These transformations in Open64 context are often implemented as a rewriting of WHIRL at some level. For example, [15] performed the structure peeling and splitting in IPA. However, performing the transformation at intermediate level has a few limitations due to the information loss in front-end's conversion to WHIRL. For example, when structure size changes, all places with *sizeof* expression of the structure need to be changed. But during conversion to WHIRL, the front-end would have already turned the *sizeof* expression into a constant number. [15] suggests looking at the receiver pointer type at allocation sites to identify the places to change. However, it is hard to trace the receiving pointer when the calls to allocation routines are wrapped in custom wrappers, as in the case of *parser* and *twolf*.

To avoid this information loss, we develop LIDO, Locality Inspired Data Optimizer, to perform these transformations at the source level. The effect of performing the transformation at source level has its strength and weakness. On one hand, transformations on data structures are naturally expressible at the source level. As these transformation are rather high-level, describing the transformation rules in lower levels would be much more verbose. In addition, the source level transformation allows easy debugging as all changes are completely reflected in the human-readable transformed source files. On the other hand, special care need be taken to avoid generating code patterns that confuses later optimizations. These factors motivate us to develop LIDO.

B. Background on CIL

LIDO is based on CIL[16], a source-to-source transformation framework previously used for verification of C code. Unlike some source transformation tools like TXL[17] that works by direct pattern matching, CIL first transforms the source file to a standard form and then provides interfaces for tree rewriting rules based on type information. We observe that this *canonicalizing* process of the source considerably improves robustness of consequent analysis and transformations. For example, after transformation, arguments of functions are guaranteed to be side-effect-free, and different forms of loops are all converted into a *while(1)* loop, with exit statement placed at the head of loop body. Nevertheless, as mentioned above, this transformation may confuse later phases of compiler transformation. CIL provides an option to mildly transform the code to more compiler-friendly form. An example from *art* is shown in Figure 5. Although it can be observed that the three forms are equivalent, Open64 produces much inferior code for the second format in our experiments. Inspecting the intermediate WHIRL suggests that it is due to failure of Open64 to recover the loop index variable from the code in AS-IS format.

With the built-in support for dataflow analysis in CIL, pointer analysis etc., the legality of transformations can be established through analysis.

C. Structure of LIDO

LIDO is implemented as a plug-in to CIL. It performs both analysis and transformations and includes modules to analyze *DEF-USE* of arrays, uncover constant structures, and recognize loop invariants. These analysis are light-weight and not meant to replace corresponding analysis in later phases of compiler. Instead, they are performed on demand of establishing legality of a potential transformation. At this stage of development, we only employ minimal alias analysis. LIDO supports various transformations of structure layout and array layout, and some pattern matching rules to optimize special forms of program construct. Figure 6 illustractes structure of LIDO. LIDO works between C preprocessor and Loongcc. Both its input and output are in CIL format, which are also plain C sources. As analysis of LIDO requires frequency information corresponding directly to statements in the plain C source, we cannot use data

```
Original:
   for (j=0;j<numf1s;j++)
      norm += f1_layer[j].R
        * f1_layer[j].R;

CIL AS-IS format:
  j____0 = 0;
  while (1) {
    if (j____0 < numf1s) {

    } else {
      break;
    }
    norm += (f1_layer + j____0)->R
        * (f1_layer + j____0)->R;
    j____0 += 1;
  }

Simplified:
  j____0 = 0;
  while (j____0 < numf1s) {
    norm += (f1_layer + j____0)->R
        * (f1_layer + j____0)->R;
    j____0 ++;
  }
```

Figure 5: Effect of the canonicalizing of source file to CIL and later simplification. *j* is renamed to reflect its scope of definition so as to avoid name conflicts.

collected by WHIRL/edge profiling of Open64 and have to develop an instrumentation framework that operates at the source level, which works as a standalone phase of Loongcc.

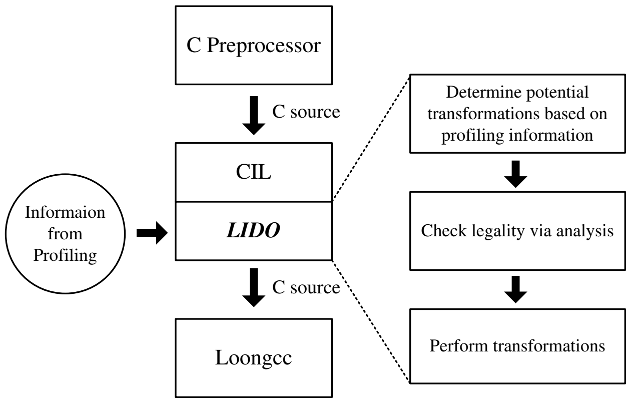

C Preprocessor
C source
CIL
LIDO
C source
Loongcc
Informaion from Profiling
Determine potential transformations based on profiling information
Check legality via analysis
Perform transformations

Figure 6: Structure of LIDO

D. Example Transformations

In this subsection we will cover LIDO's transformations of *art* and *equake* in detail.

```
Original:
typedef struct {
    double *I;
    double W;
    double X;
    double V;
    double U;
    double P;
    double Q;
    double R;
        } f1_neuron;

Transformed:
    double ** f1_layer_I;
    double * f1_layer_W;
    ...
```

Figure 7: *f1_layer* structure in *art*

As the first step, either by extracting frequency information collected by Loongcc or by heuristics, we can decide that the global array *f1_layer* is the key data structure. By inspecting its *USE* sites, we observe that typically only part of fields in the structure is accessed in loops. Intending to optimize the spatial locality, we perform structure peeling for the *f1_neuron* array to turn it into a couple of smaller arrays, named as *f1_layer_I f1_layer_W* etc. It is performed as in [15], but now at source level. The transformation is shown in Figure 7. The new array names are constructed to correspond to the field names.

After structure peeling, LIDO performs a custom dataflow analysis inspecting *DEF-USE* chains of the newly created arrays, as they are now the key data structure in the program. The analysis reveals that *f1_layer_P* is always equal to *f1_layer_Q*. Hence *USE* of *f1_layer_Q* can be replaced by that of *f1_layer_P*. However, we cannot replace in reverse order as *f1_layer_P* escapes the procedure. This requires another round of dataflow analysis, which is a weak form of shape analysis[22]. The analysis is performed only when required to establish the legality by some transformations. The results of analysis is saved for potential reuse.

The dataflow analysis of arrays also reveals that *f1_layer_W/f1_layer_V/f1_layer_X* and *f1_layer_U* have a special usage pattern. These arrays are typically used as temporary storage places for intermediate calculation results and are visited in circular style in a hot loop. As they are stream processed and not reused, bringing them into the cache only pollute the cache. Observing that there is no real overlap of lifetime of these arrays, we can overlay these arrays to overlap the region of cache that these arrays may pollute, consequently increase the chance for cache reuse of other data. In addition, the write-allocation policy of cache management requires data to be first loaded into cache before further operations. In this case, overlaying of temporary arrays reduces the number of write-misses. Last but not least, overlaying exploits the fact that there is no need to write these temporary values into main memory, as long as they are present in cache. The effect of overlay can be verified by statistics in Table III. We can observe that array overlaying significantly reduces the amount of traffic among L1 cache, L2 cache and main memory. Array contraction does not apply here due to data dependency. We perform the overlay by straightforward replacement of references.

Event	Normalized number of event after overlay
CPU_CLK	0.698
DCACHE_MISSES	0.696
MEM_READ	0.683
MEM_WRITE	0.317

Table III: Overlay of arrays in *art* causes number of many performance-critical events to be reduced. CPU_CLK measures the number of clock cycles. DCACHE_MISSES counts L1 data cache misses. MEM_READ and MEM_WRITE measures read/write traffic between L2 cache and main memory. All statistics are collected by Oprofile[**?**] by checking performance counters of Loongson 2F.

Also by inspecting the hot loops identified from profiling information, we observe that *bus* array and *tds* array are multi-dimensional arrays accessed in non-row-major order. LIDO flattens these arrays to one-dimension after proper array transposition. In addition, as structure peeling produces *f1_layer_I* as an two-dimensional array as in Figure 7, we can also flatten it to further improve performance. The fact that input and output of transformations are all valid C code in CIL format allows direct chaining of transformations. The multi-dimensional arrays are all dynamically allocated arrays, which is a common program idiom as C language prior to C99 does not allow variable-length arrays. The flow of the array flattening and transposition in LIDO is performed in three passes over the whole source. First, all allocation sites of the target array are scanned to gather span of each dimension by inspecting the arguments passed to the allocator. In the second pass, the allocation sites are transformed. The size argument of allocation calls for the base dimension pointer are increased to hold the whole array. Allocation calls for higher dimensions are deleted. Finally, LIDO scans all the *USE* sites of the array and transforms the index expression based on information collected in the first pass. We also map the *USE* of pointers of higher dimension to equivalent addresses in the new array. Transposition of array can be performed in this pass by simply swapping items in the index expression. An example is given in Figure 8.

The transformations will be aborted whenever legality cannot be established, like when programmers hard-code the size of structure as ”magic numbers” and the receiving pointer does not give type information, which is a case of bad programming style hindering optimization.

LIDO also includes some miscellaneous transformations.

```
Original:

   while (ti < numf1s) {
     (f1_layer + ti)->W =
         *((f1_layer + ti)->I + cp)
             + a * (f1_layer + ti)->U;
     tnorm += (f1_layer + ti)->W
         * (f1_layer + ti)->W;
     ti ++;
   }

Transformed:
   while (ti < numf1s) {
     *(f1_layer_U + ti) =
        *(f1_layer_I +
         (cp *
          (unsigned int )numf1s + ti))
        + a * *(f1_layer_U + ti);
     tnorm += *(f1_layer_U + ti)
         * *(f1_layer_U + ti);
     ti ++;
   }
```

Figure 8: Flattening *f1_layer_I* in *art*. *numf1s* is the span of inner-most dimension of *f1_layer_I*. Also note that accesses to W field are first transformed to accesses to *f1_layer_W*, and then replaced by accesses to *f1_layer_U* due to overlay.

```
Original:

    for (tj=0;tj<numf2s;tj++)
    {
       if ((tj == winner)&&(Y[tj].y > 0))
             tsum += tds[ti][tj] * d;
    }

Transformed:

    if(winner >=0 && winner <numf2s)
    {
       tj = winner;
       if (Y[tj].y > 0)
             tsum += tds[ti][tj] * d;
    }
```

Figure 9: An inefficient loop in *art*

For example, we observe that a hot loop can be reduced as in Figure 9, which we implement as a tree rewriting rule as follows: LIDO scans for all loops enclosing a conditional branch involving an equality test of index variable. The loop found is reduced to a conditional after transformation and the equality test in the loop body is deleted.

Transformations applied for *equake* are array flattening,

```
for (i = 0; i < ARCHnodes; i++)
  for (j = 0; j < 3; j++)
    disp[disptplus][i][j] *=
         - Exc.dt * Exc.dt;

for (i = 0; i < ARCHnodes; i++)
  for (j = 0; j < 3; j++)
    disp[disptplus][i][j] +=
    2.0 * M[i][j] * disp[dispt][i][j] -
   (M[i][j] - Exc.dt / 2.0 * C[i][j]) *
    disp[disptminus][i][j] -
    Exc.dt * Exc.dt *
    (M23[i][j] * phi2(time) / 2.0 +
        C23[i][j] * phi1(time) / 2.0 +
        V23[i][j] * phi0(time) / 2.0);

for (i = 0; i < ARCHnodes; i++)
  for (j = 0; j < 3; j++)
    disp[disptplus][i][j] =
     disp[disptplus][i][j] /
       (M[i][j] + Exc.dt / 2.0 * C[i][j]);
```

Figure 10: Hot loops in *equake*

lifting of loop invariant codes and aggressive loop fusion. One of the kernel loops of *equake* is shown in Figure 10.

The *phi0/phi1/phi2* functions are in fact pure, but compiler fail to recognize their purity because the functions read some global variables. We perform a simple analysis to discover constants of structure type and then replace their *USE* by constants, hence establishing purity of the *phi0/phi1/phi2* functions. LIDO also observe that fusing the three kernel loops may be beneficial. However, data dependency may exist if any two of *disptplus*, *dispt* and *disptminus* are equal. To resolve this dependency, an analysis is performed by a scan of *DEF* of the *point-to* sets [23] of these variables.

As the scanning in LIDO is performed in top-down tree traversal manner, the algorithm complexity is *O(kn)* where *n* is the length of the program and *k* is the number of passes. Therefore, although every transformation performed by LIDO requires a few scanning passes of the whole syntax tree, the time of analysis and transformations is barely noticeable in our experiments.

The applicability of LIDO is limited only by that of CIL, which can parse the whole LINUX kernel. However, due to front-end restrictions, LIDO can only perform transformations on C sources. We hope to remove this restriction in later development.

E. Effect of applying LIDO

As source-to-source transformation is compiler independent, we measure the effect of LIDO for *art* and *equake* on three platforms. Table IV gives the performance gain achieved by LIDO on an Intel Xeon 2.33GHz and an AMD Opteron 1GHz. For Xeon we use Intel C Compiler 10.1, for Opteron we use Pathscale 3.1, with flags fetched from SPEC CPU2000 peak configurations. No parallelization is involved here.

Benchmarks	Xeon	Opteron	Loongson 2F
art	+344%	+106%	+349%
art without array overlay	+189%	+64%	+212%
equake	+53%	+30%	+44%

Table IV: Performance for *art* and *equake* on three platforms.

Structure peeling and matrix reorganization on *art* has been implemented in GCC under the ”-fipa-struct-reorg” and ”-fipa-matrix-reorg” flags. GCC with these two flags yield 93% improvement on Loongson 2F. However, to our knowledge, the overlaying of arrays of splitted structure fields in LIDO is unique. It can be seen from the table that the overlay contributes significantly to the gross speedup. By composing all transformations mentioned above, LIDO achieves better overall performance gain for *art*.

As main strength of LIDO is in improving effective memory bandwidth, applying LIDO to other C test cases in SPEC2000 does not yield significant performance gain and is not shown here.

V. Related work

Our work is part of continuous effort at ICT to develop custom compilers designed for Loongson family. The back end of our compiler is heavily influenced by ORC. In parallel to our work, Cui Huimin independently developed a backend for Loongson II based on an Open64 branch. What distinguishes our work is the maturity and the emphasis on performance.

[24] describes porting Open64 to Power-PC platform. They develop a method to partially automate the creation of code generators.

Cetus[21] is another open source source-to-source transformation tool that targets C sources. To our knowledge, it does not yet include structure and array transformation modules.

VI. Conclusion

We develop Loongcc, a high performance Open64 branch for Loongson II, which is a MIPS III-compatible platform. We hope to make Loongcc open source in this year, providing another option for Open64 target platforms. Due to the RISC nature of Loongson II, we hope that our branch would serve as a basis for porting Open64 to other platforms in RISC family, while preserving most optimizations available in Loongcc.

We also develop LIDO to perform various optimizations of data structure at the source level. The transformations that LIDO performs include array overlaying, array flattening, array transposition, structure field reordering, structure peeling, and structure splitting, and optimization of specific program constructs. To establish the legality of these transformations, LIDO performs a variety of light-weight analysis. Due to frontend limitations, currently LIDO can only process C source. We hope to eliminate this restriction and extends it to work on the whole SPEC2000FP in later versions.

Acknowledgment

We are grateful to developers of MIPSPro, Open64, Pathscale and ORC for their contributions to the open source society. Mi Wei developed the first version of array contraction model. Shi Hui and Xiang Xiaoya initiated the work for locality optimization on Loongson platform. We would also like to thank the anonymous reviewers for their precious remarks which help improve this paper.

References

[1] Hu, W. W., and Wang J. Making effective decisions in computer architects' real-world: Lessons and experiences with Godson-2 processor designs. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 23(4):620-632 July 2008

[2] Pathscale compiler. http://www.pathscale.com/

[3] Open Research Compiler. http://ipf-orc.sourceforge.net

[4] Hu, W. W., Zhao, J. Y., Zhong S. Q., Yang X., Guidetti, E., and Wu C. Implementing a 1GHz Four-Issue Out-of-Order Execution Microprocessor in a Standard Cell ASIC Methodology. In Journal of Computer Science and Technology, Vol. 22, No. 1, pp.1-14, 2007.

[5] Haber, G., Klausner, M., Eisenberg, V., Mendelson, B., and Gurevich, M. 2003. Optimization opportunities created by global data reordering. In Proceedings of the international Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization (San Francisco, California, March 23 - 26, 2003). ACM International Conference Proceeding Series, vol. 37. IEEE Computer Society, Washington, DC, 228-237.

[6] Gao, G. R., Olsen, R., Sarkar, V., and Thekkath, R. 1993. Collective Loop Fusion for Array Contraction. In Proceedings of the 5th international Workshop on Languages and Compilers For Parallel Computing (August 03 - 05, 1992). U. Banerjee, D. Gelernter, A. Nicolau, and D. A. Padua, Eds. Lecture Notes In Computer Science, vol. 757. Springer-Verlag, London, 281-295.

[7] Song, Y., Xu, R., and Wang, C. 2004. Improving Data Locality by Array Contraction. IEEE Trans. Comput. 53, 9 (Sep. 2004), 1073-1084.

[8] Briggs P., Cooper K. D., and Torczon L. Rematerialization. Proceedings of the SIGPLAN 92 Conference on Programming Language Design and Implementation, SIGPLAN Notices 27(7), p.311-321. July 1992.

[9] Ding, C. and Kennedy, K. 2004. Improving effective bandwidth through compiler enhancement of global cache reuse. J. Parallel Distrib. Comput. 64, 1 (Jan. 2004), 108-134.

[10] D'Hollander, E. Discovery of Locality-Improving Refactorings by Reuse Path Analysis. Proceedings of the 2nd International Conference on High Performance Computing and Communications (HPCC). Springer. Lecture Notes in Computer Science. Vol. 4208. 2006. pp. 220–229

[11] Zhong, Y., Orlovich, M., Shen, X., and Ding, C. 2004. Array regrouping and structure splitting using whole-program reference affinity. SIGPLAN Not. 39, 6 (Jun. 2004), 255-266.

[12] Ding, C. and Kennedy, K. 2001. Improving Effective Bandwidth through Compiler Enhancement of Global Cache Reuse. In Proceedings of the 15th international Parallel and Distributed Processing Symposium

[13] Hundt, R., Mannarswamy, S., and Chakrabarti, D. 2006. Practical Structure Layout Optimization and Advice. In Proceedings of the international Symposium on Code Generation and Optimization (March 26 - 29, 2006). Code Generation and Optimization. IEEE Computer Society, Washington, DC, 233-244.

[14] Chilimbi, T. M., Hill, M. D., and Larus, J. R. 1999. Cache-conscious structure layout. In Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation (Atlanta, Georgia, United States, May 01 - 04, 1999). PLDI '99. ACM, New York, NY, 1-12.

[15] Gautam C., Fred C., Structure Layout Optimizations in the Open64 Compiler: Design, Implementation and Mesurements. In Open64 Workshop at CGO 2008, Boston, Massachusetts.

[16] Necula, G. C., McPeak, S., Rahul, S. P., and Weimer, W. 2002. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In Proceedings of the 11th international Conference on Compiler Construction (April 08 - 12, 2002). R. N. Horspool, Ed. Lecture Notes In Computer Science, vol. 2304. Springer-Verlag, London, 213-228.

[17] Thurston A. and Cordy J.R. , Evolving TXL, Proc. SCAM 2006, IEEE 6th International Workshop on Source Code Analysis and Manipulation, Philadelphia, September 2006, pp. 117-126.

[18] GNU Compiler Collections. http://gcc.gnu.org/

[19] SPEC. Standard performance evaluation corporation. http://www.spec.org

[20] Mostafa H. and Caroline T.. Cache aware data layout reorganization optimization in GCC . In Proceedings of the GCC Developers' Summit , pages 69–92, 2005.

[21] Lee S. I., Johnson T. A., and Eigenmann R.. Cetus - An Extensible Compiler Infrastructure for Source-to-Source Transformation. In Proc. of the Workshop on Languages and Compilers for Parallel Computing(LCPC'03), pages 539–553. (Springer-Verlag Lecture Notes in Computer Science), Oct. 2003. (Ipdps'01) - Volume 1 (April 23 - 27, 2001). IPDPS. IEEE Computer Society, Washington, DC, 10038.2.

[22] Mooly S., Thomas R., Reinhard W. May 2002. ”Parametric shape analysis via 3-valued logic”. ACM Transactions on Programming Languages and Systems (TOPLAS) (ACM) 24 (3): 217C298. Beyls, K.;

[23] OProfile system-wide profiler. http://oprofile.sourceforge.net

[24] Steensgaard, B. 1996. Points-to analysis in almost linear time. In Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (St. Petersburg Beach, Florida, United States, January 21 - 24, 1996). POPL ’96. ACM, New York, NY, 32-41.

[25] Lin M., Yu Z., Zhang D., Zhu Y., Wang S., Dong Y., Retargeting the Open64 Compiler to PowerPC Processor, icesssymposia,pp.152-157, 2008 International Conference on Embedded Software and Systems Symposia, 2008