MAO - an Extensible Micro-Architectural Optimizer

Robert Hundt, Easwaran Raman, Martin Thuresson, Neil Vachharajani
Google
1600 Amphitheatre Parkway
Mountain View, CA, 94043
{rhundt, eraman, martint}@google.com, nvachhar@gmail.com

***Abstract*—Performance matters, and so does repeatability and predictability. Today's processors' micro-architectures have become so complex as to now contain many undocumented, not understood, and even puzzling performance cliffs. Small changes in the instruction stream, such as the insertion of a single NOP instruction, can lead to significant performance deltas, with the effect of exposing compiler and performance optimization efforts to perceived unwanted randomness.**

This paper presents MAO, an extensible micro-architectural assembly to assembly optimizer, which seeks to address this problem for x86/64 processors. In essence, MAO is a thin wrapper around a common open source assembler infrastructure. It offers basic operations, such as creation or modification of instructions, simple data-flow analysis, and advanced infra-structure, such as loop recognition, and a repeated relaxation algorithm to compute instruction addresses and lengths. This infrastructure enables a plethora of passes for pattern matching, alignment specific optimizations, peep-holes, experiments (such as random insertion of NOPs), and fast prototyping of more sophisticated optimizations. MAO can be integrated into any compiler that emits assembly code, or can be used standalone. MAO can be used to discover micro-architectural details semi-automatically. Initial performance results are encouraging.

I. Introduction

Several real world examples from our daily development experience helped motivate the development of MAO. A programmer cleans up her code. She doesn't change the code itself, but only adds comments and renames a few C++ classes, functions, and variable names. After this innocuous change, a performance benchmark extracted from this code shows a 35% degradation. There is no obvious explanation for the degradation. Profiling reveals that the degradation comes from a single function. However, this function has an instruction stream identical to the version before the change and even the function's starting address has the same alignment. After a long and painful analysis using the available hardware performance counters, the branch predictor is identified as the source of the problem. Changes to the relative function layout resulted in a pathological, cross-function branch-predictor aliasing problem. None of the available documentation on the inner workings of the branch predictor would explain this issue in full.

For a concrete example of the scenario described above, consider the short code snippet in Figure 1 from a hot loop unrolled twice from the SPEC2000 181.mcf benchmark. Merely inserting the `nop` instruction right before label `.L5` results in a 5% performance speed-up for this loop on a common Intel Core-2 platform. Our analysis of several hardware performance counters revealed that another branch predictor problem was the likely root cause of the problem.

```
.L3      movsbl  1(%rdi,%r8,4),%edx
         movsbl  (%rdi,%r8,4),%eax
            # ... 6 instructions
         movl    %edx, (%rsi,%r8,4)
         addq    $1, %r8

         nop     # this instruction speeds up
                 # the loop by 5%

.L5:     movsbl  1(%rdi,%r8,4),%edx
         movsbl  (%rdi,%r8,4),%eax
            # ... identical code sequence
         movl    %edx, (%rsi,%r8,4)
         addq    $1, %r8
         cmpl    %r8d, %r9d
         jg      .L3
```

Fig. 1. Code snippet with high impact NOP instruction

For writers of compiler optimizations and code performance tuning experts, such pathological effects pose a significant challenge. Performance effects can no longer be attributed to a particular transformation. Unknown or undocumented micro-architectural effects might be the real cause of performance gains or losses.

Typically, compilers seek to address these issues in a low level optimizer (LLO), which has intimate knowledge about a particular micro-architecture. LLO combines the key passes of register allocation, instruction selection, and scheduling, with optimization passes such as loop unrolling, prefetching, peep-hole optimizations, and alignment specific optimizations. However, because of phase ordering issues in the compiler, lack of knowledge, or go-to-market pressure, compilers typically are not good at handling such micro-architectural effects.

Furthermore, several popular compilers have architectural limitations that prevent them from even addressing micro-architectural optimizations. For example, the GNU Compiler Collection (GCC) [7] follows a structure as in Figure 2. Source code is parsed and kept in an Abstract Syntax Tree representation in the compiler front-end (FE). This representation is transformed into GENERIC and from there directly into GIMPLE. This compiler intermediate representation (IR) allows high level optimizations, such as SSA based optimizations. From there, the IR is transformed into the Register Transfer Language (RTL), a lower level representation closer to the machine. Low level optimizations, such as register allocation and instruction selection, are performed at this level. The final results are in tree form at this IR level. As a final step, these trees are matched against a machine description file (MD). As a match is made, assembly instructions are written out to an assembly file. This file is then passed on to the assembler, which produces object code. Note that the differently colored box labeled MAO in Figure 2 is not part of the standard GCC pipeline.

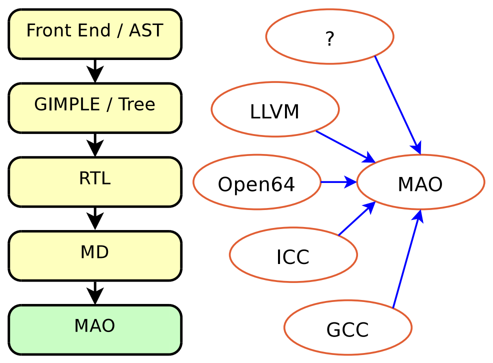
This architecture allows for a portable compiler, which is re-targetable to new ISAs with minimal effort. However, it is important to note that the compiler basically stops reasoning about instructions after the RTL level. It writes textual assembly instructions into an output file using simple print routines. For complex instruction set architectures, it has little or no information about instruction lengths, alignments, or aliasing effects, which prevents it from effectively performing micro-architectural optimizations (even assuming it had all the platform specific proprietary information from the chip vendors).

Other code generating infrastructures, e.g., simple dynamic code generators and opcode threading dynamic compilers, typically seek the benefits of compiled code but don't have the time or bandwidth to implement a full blown low level optimizer. Besides GCC, there are several other open source static and dynamic compilers, e.g., Open64 [21] and LLVM [13]. All these development infrastructures run into similar issues. For example, it was observed that for a particular hot loop, LLVM would create the same code sequence as Intel's `icc` compiler, but that `icc` would place a strategic `nop` instruction into the loop, resulting in a substantial performance gain [4].

In this paper we present MAO, an extensible micro-architectural optimizer for x86/64. It accepts as input an assembly file and converts it into an IR. This IR is only a thin wrapper around the internal data structures of GNU's binutils, thus guaranteeing full future ISA compatibility. It performs optimizations on this IR and emits as output another assembly file, which can then flow through the standard toolchain.

Since all of the aforementioned compilers have the ability to emit assembly code, an assembly to assembly micro-architectural optimizer has the potential to benefit all these infrastructures, as indicated in Figure 2b. Furthermore, since all instructions are represented via a single C struct in this IR, MAO can be easily integrated into dynamic code generators without requiring them to take a compilation detour through textual assembly files. Referring back to Figure 2a, MAO represents another layer below the assembly level to allow for micro-architectural optimizations.

The characteristics of the performance cliffs and micro-architectural features and deficiencies change from processor stepping to processor stepping, differ between generations of architectures, and vary between ISA implementations from different vendors. We named MAO not just an optimizing assembler, but a full blown micro-architectural optimizer, because of its ability to programmatically discover performance cliffs and micro-architectural features and deficiencies.



(a) MAO and GCC. (b) Possible input to MAO.

Fig. 2. Various IR levels in GCC. MAO is not part of the original lowering. When integrated, MAO adds another layer below the abstraction level of MD files and allows reasoning about assembly instructions.

In this paper, we make the following contributions:

- We provide a detailed description of the design and implementation of a flexible infrastructure, based on GNU binutils, to perform micro-architectural optimizations.
- We present several micro-architectural performance cliffs (some unknown to the community) and optimization passes to address them.
- We outline how this infrastructure can be used to automatically or semi-automatically discover micro-architectural features and deficiencies.
- We provide performance anecdotes to highlight several aspects of the infrastructure and the performance effects it addresses.

As a caveat to the reader - this paper clearly describes an imminent and important problem, its performance impact, as well as strategies and parts of solutions. However, it does not provide a single, conclusive cure. More research needs to be done and the community is encouraged to contribute to this open source project.

While this paper focuses on x86/64, we believe the insights are applicable to a wider range of platforms, in particular to those with variable-length instruction encodings and complex micro-architectures. The underlying GNU binutils infrastructure supports multiple platforms, and we believe MAO can be made to support multiple platforms as well with modest effort. MAO is open source under GNU GPL v3 and available online [15].

The rest of the paper is organized as follows. Section II describes the core properties of the IR. Section III describes a variety of passes with a range of characteristics. This section displays the power of our simple approach and makes the case for an automatic system for feature detection which is outlined in Section IV. Section V presents anecdotal performance results. Section VI discusses related work and Section VII concludes.

II. Infrastructure

MAO utilizes the GNU assembler (`gas`) from binutils 2.19 (and later) for parsing and binary encoding of assembly files and instructions. Changes have been submitted to the open-source binutils to allow its usage in the MAO context. As MAO itself is being built, some minimal modifications still have to be made to gas to enable integration with MAO, specifically, to suppress binary file creation and to allow repeated relaxation (explained in detail later). Corresponding small patches are delivered as part of MAO's source distribution. Our goal is to improve binutils to avoid the need for any local patches in the future.

Since MAO is based on gas, it accepts assembly files in either Intel or AT&T syntax, for 32-bit or 64-bit build models. The input is parsed with gas' table driven encoder, which encodes every possible x86 instruction into a single C struct type. In regular operation mode, gas performs relaxation and generates a binary object file from sequences of these structs. In MAO mode, however, the generation of binary output is suppressed. Instead, the encoded instructions become part of the MAO IR. After parsing has completed, control is given to the optimizing passes of MAO, which add, delete, or modify IR nodes and instructions. All instruction modifications are done on the underlying C struct type. At the end of the optimization phase, MAO writes out the content of these structs in legible textual assembly. It should be noted that while gas is written in C and MAO reuses types from it, MAO itself is written in C++.

After parsing, all assembly directives and instructions form one long list of MAO IR nodes. To reflect the structure of assembly files, MAO offers a notion of sections and functions and provides easy access to these higher level concepts via corresponding iterators. For example, if a function located in a code section happens to be split into two sections by an an intermittent data section, which is a pattern commonly emitted by compilers during translation of C switch statements, the linker will produce a single continuous function body and move the data section somewhere else. MAO's optimization passes should not have to care about such details and MAO's function and instruction iterators handle this transparently.

Every instruction provides access to all potential modifiers, opcode, operands (including all registers and offsets participating in the various addressing modes), the instruction's binary encoding, and its encoded length.

MAO offers a simple data flow apparatus, but no alias or points-to analysis. Since many assembly instructions work on registers, this data flow mechanism is powerful and solves many otherwise difficult to reason about problems for the optimization passes. MAO uses a table-driven approach to model side effects. A tiny configuration language specifies opcodes, operands being modified, flags set, and other potential side effects. A generator program constructs C tables for use by MAO.

MAO offers a per-function control-flow graph (CFG). In the presence of indirect jumps, building this graph can be undecidable. However, we rely on the fact that we handle compiler generated assembly files and recognize a handful of patterns to handle indirect jumps properly, e.g., to find jump tables. If, for a function, a particular branch cannot be resolved, the function gets flagged and optimization passes can decide whether or not to proceed. In practice, this mechanism is brittle and needs constant maintenance. We tested this functionality on a source base of high complexity containing many inline assembly sequences. When we updated the internal compiler to a newer version, we found that 246 out of 320 indirect branches could no longer be resolved. After adding a single pattern that uses the data flow framework's reaching definitions functionality, only 4 out of the 320 indirect branches (1.2%) remained unresolved. The remaining patterns were complex, uncommon cross-basic block scenarios.

MAO offers a loop detection mechanism based on Havlak [8]. It builds a hierarchical loop structure graph (LSG) representing the nesting relationships of a given loop nest. In this graph, each node contains the nested loops, as well as the remaining basic blocks. The algorithm allows distinguishing between reducible and irreducible loops and it is up to particular optimization passes to decide how to proceed in the presence of irreducible loops. In practice, irreducible loops are seen rarely, and we have only encountered a handful in legacy spaghetti FORTRAN code.

MAO also offers the ability to perform repeated relaxation, which is essential for alignment-based optimizations. Relaxation is the process of finding proper instruction sizes for branches based on branch target distances. This is essential to determine the start addresses of all instructions. As an example, consider the following assembly sequence with instruction offsets and encoding:

```
  0:  55                      push   %rbp
  1:  48 89 e5                mov    %rsp,%rbp
  4:  c7 45 fc 05 00 00 00    movl   $0x5,-0x4(%rbp)
  b:  eb 7f                   jmp    8c <main+0x8c>
  d:  83 45 fc 01             addl   $0x1,-0x4(%rbp)
 11:  83 6d fc 01             subl   $0x1,-0x4(%rbp)
   <instructions>
 8c:  83 7d fc 00             cmpl   $0x0,-0x4(%rbp)
 90:  0f 85 7a ff ff ff       jne    d <main+0xd>
```

The branch at offset b has a 2 bytes encoding (0xeb, 0x7c). Consider a scenario where the branch target moves down, e.g., because an optimization pass inserted a single-byte nop instruction right before the cmpl instruction at offset 0x8c:

```
  0:  55                      push   %rbp
  1:  48 89 e5                mov    %rsp,%rbp
  4:  c7 45 fc 05 00 00 00    movl   $0x5,-0x4(%rbp)
  b:  e9 80 00 00 00          jmpq   90 <main+0x90>
 10:  83 45 fc 01             addl   $0x1,-0x4(%rbp)
 14:  83 6d fc 01             subl   $0x1,-0x4(%rbp)
   <instructions>
 8f:  90                      nop
 90:  83 7d fc 00             cmpl   $0x0,-0x4(%rbp)
 94:  0f 85 76 ff ff ff       jne    10 <main+0x10>
```

Then the jump at offset b requires a 5 byte encoding. As a result, the cmpl instruction moves down by 4 bytes, out of which 1 byte is due to the nop instruction, and 3 bytes are due to the changed encoding. Now, since the instruction's length and other instructions' offsets changed, the encodings of other branches might change and hence another iteration of this algorithm becomes necessary.

Relaxation in the general case is an NP-complete problem. In the implementation there is a built-in limit of 100 iterations, but in practice almost every relaxation succeeds in a few iterations, and it never fails. The `gas` assembler only performed relaxation once before generating a binary output file. In MAO, however, multiple relaxations are necessary and changes were made to `gas` to support this feature.

The requirement for relaxation complicates optimization passes. If, for example, two branches are carefully placed in two separate 32-byte bundles to avoid branch predictor aliasing problems, relaxation and the corresponding shifting of instruction addresses can invalidate that placement. Some of these problems can be resolved with iterative approaches or via clever phase ordering. Finding a more general solution remains an interesting research problem.

MAO's IR can also be annotated with hardware counter profile information. Tools like oprofile [14] associate hardware event samples to offsets within functions. Since MAO has instruction sizes available, samples can be directly mapped to individual instructions. Similar to Chen [3] we plan to construct edge profiles from this information as future work, as that information can make a large performance difference in certain contexts.

Finally, the MAO IR itself is intentionally kept simple. This design enables performance experts knowledgeable in x86 assembly but lacking knowledge of a compiler's internals, to write optimization passes. In our environment, experts used MAO to prototype optimizations, to modify short instruction sequences, or to validate a given performance assumption.

III. Optimization Passes

MAO optimization passes can be roughly classified into categories such as pattern matches or peep-holes, alignment specific optimizations, scalar optimizations, advanced optimizations, and experiments. We first outline how to construct and invoke an optimization pass and then provide examples of passes in each category. As a sample code base we used a core library at Google which consists of approximately 80 complex C++ files containing many inline assembly sequences.

A. Pass Definition

MAO supports two types of passes: function specific passes, which get invoked for every identified function in an assembly file, and passes which process the full IR for an assembly file. There are only a few full IR passes, e.g., reading/parsing of the input and emission of the output assembly. Most other passes operate at the function granularity.

Writing a pass is easy and follows the template shown in Figure 3, which implements a pass that just prints function names only. The optimization pass is a C++ class derived from a base class `MaoFunctionPass` and contains a `Go()` function, which serves as the main entry point for the pass. In the example, the `Go()` function simply prints the name of the current function, using the standard tracing facility that is available to every pass by default. To make passes externally visible, an invocation of `REGISTER_FUNC_PASS` is required to register a pass under a name.

```
#include "Mao.h"

namespace {
  MAO_OPTIONS_DEFINE(MAOPASS, 0) { };

  class MaoPass : public MaoFunctionPass {
  public:
    MaoPass(MaoOptionMap *opts, // specific options
            MaoUnit      *mao,  // current asm file
            Function     *fct)  // current function
        : MaoFunctionPass("MAOPASS", opts, mao, fct)
    { }

    bool Go() {
      Trace(3, "Func: %s", fct->name().c_str());
      return true;
    }
  }
REGISTER_FUNC_PASS("MAOPASS", MaoPass)
}  // namespace
```

Fig. 3. Structure of a minimal optimization pass

Passes can be statically linked into MAO, or dynamically loaded as plug-ins. To make a pass a plug-in only requires adding another macro. Since all passes are derived from a common base class, they all offer common functionality, e.g., dumping the current state of the IR before or after a given pass in various formats, or to specify pass specific parameters.

Passes are named and their invocation is controlled via command-line options. MAO-specific options are prefixed with `--mao=`. Options without this prefix are passed through to the underlying `gas` assembler. For example, the following command line

```
...mao --mao=LFIND=trace[0]:ASM=o[/dev/null] in.s
```

invokes a MAO pass named LFIND, turns on tracing and generates the assembly output to `/dev/null`.

The order of passes on the command line specifies the pass invocation order. When running analysis-only passes, the assembly generation `ASM` pass can be omitted. Reading and parsing the input and converting it to the IR is a pass as well, but called by default as the first pass.

To verify correctness of basic MAO functionality, we run MAO on the `gas` test suite and also compile large source bases. For each source file we take the compiler generated assembly file A_1 and run the assembler on it to generate an object file O_1. Then we run MAO on A_1, construct the CFG and perform loop recognition, and generate an assembly file A_2. We run the assembler and generate an object file O_2. We then disassemble O_1 and O_2 and verify that both disassembled files are textually identical. Since MAO didn't perform any transformations, the disassembled files must match.

B. Pattern Matching

a) Redundant Zero Extension: Pattern matching passes try to cleanup redundant or bad code sequences which typically come from weaknesses or deficiencies in the compiler.

For example, GCC 4.3 and 4.4 does not model sign- or zero-extension well. As a result, many code sequences like the following are generated:

```
andl $255,%eax
mov  %eax,%eax
```

The second `mov` instruction is meant to zero-extend register `%eax`. However, this operation is a by-product of the preceding `andl` instruction and therefore redundant. In the sample Google core library we find approximately 1000 occurrences of this pattern.

This transformation serves well to make a key point about MAO. We implemented the same transformation, redundant zero extension, in the GCC 4.3 compiler. We had to go through several rounds of implementation attempts until we got it right, and, more importantly, until the upstream reviewers accepted the implementation. After a lot of effort the patches were committed, only to find that the performance effect on the target code base was minimal. A simple prototype implementation in MAO catches more than 90% of the opportunities handled by the compiler. This prototype was implemented in less than a day and could have helped guide decisions on where to better spend time and resources.

b) Redundant Test Instructions: GCC does not model the x86/64 specific condition codes well. Certain instructions set certain condition codes in a rather non-canonical fashion. As a result, we find code sequences like the following:

```
subl   16, %r15d
testl  %r15d, %r15d
```

The `testl` instruction is redundant, as the proper condition codes are being set by the `subl` instruction. On the same core library, we find a total of 79763 test instructions, of which 19272 (24%) are redundant. MAO precisely models the x86/64 condition codes, enabling it to remove the redundant tests.

c) Redundant Memory Access: Because of phase ordering issues and how register allocation is performed in GCC, we find many code sequences of this form:

```
movq   24(%rsp), %rdx
movq   24(%rsp), %rcx
```

This can be expressed with shorter encoding by reusing the `%rdx` register. The resulting code sequence is two bytes shorter, and furthermore only performs one single explicit memory access:

```
movq   24(%rsp), %rdx
movq   %rdx, %rcx
```

In the same code base used above, we find 13362 occurrences of this pattern.

d) Add/Add Sequences: Even more trivial code patterns seem to escape in today's mature open-source and closed-source compilers. For example, GCC 4.3 generates patterns of multiple `add` instructions in a row, e.g.:

```
add/sub rX, IMM1
  ... no re-definition/use of rX,
  ... no use of condition codes
add/sub rX, IMM2
```

This pattern can be replaced with a single `add/sub` instruction after folding the constants `IMM1` and `IMM2`. In addition to these described patterns a plethora of similar non-optimal patterns is being generated by many different compilers. With MAO, these patterns can be found and corrected easily.

C. Alignment Optimizations

e) Short Loop Alignment: Alignment specific optimizations seek to change instructions' relative placement to utilize processor resources in a more effective manner. For example, we found a 7% performance degradation in the SPEC 2000 int benchmark 252.eon between GCC 4.3 and the previous GCC 4.2. The performance degradation was caused by a very short loop sequence:

```
426100: movss  %xmm0,(%rdi,%rax,4)
426105: add    $0x1,%rax
426109: cmp    $0x8,%rax
42610d: jne    426100
```

The degraded version was identical, except it crossed a 16-byte alignment boundary:

```
423c78: movss  %xmm0,(%rdi,%rax,1)
423c7d: add    $0x4,%rax
423c81: cmp    $0x20,%rax
423c85: jne    423c78
```

After careful examination of the hardware performance counters, an instruction decoding bottleneck was suspected. The x86/64 Core-2 decodes instructions in 16-byte chunks. Aligning the loop at 16 byte boundary resulted in decoding of only one line instead of two. Note that at higher optimization levels, GCC and other compilers insert many alignment directives in the generated assembly files. However, the placement of these directives is very crude and based on imprecise assumptions about the hardware. In particular, this loop remained unaligned.

f) Loop Stream Detector: The Loop Stream Detector (LSD) is a specialized hardware structure on various Intel platforms meant to bypass instruction fetching and decoding. There are strict requirements for loops to be streamed from the LSD. The loop must execute a minimum of 64 iterations, must not span more than four 16-byte decoding lines, and may only contain certain types of branches. These requirements change for more modern processor generations. For example, these three basic blocks forming a loop might be physically located as to span six 16-byte decoding lines, as illustrated in Figure 4.

```
l0:
  cmp %r1d, %d2d
  jne l1                  l2:
[...]                       add $0x1, %r10d
l1:                         add $0x9, %r8d
  add $0x7, %r9d            add $0x1, %esi
  mov %r1d, %d2d            add $0x1, %r1d
  cmp %r2d, %d1d            cmp $0x12345678, %r10d
  jne l2                    jl  l0
```

Inserting six `nop` instructions moves the code so as to now only span four 16-byte decoding lines, as seen in Figure 5. The insertion of these `nop` instructions speeds the loop up by a factor of two.

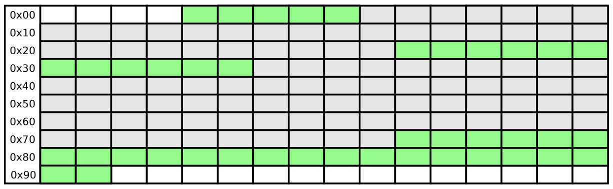


Fig. 4. Initial layout of instructions (green) which are spread out over 6 16-byte decoding lines.

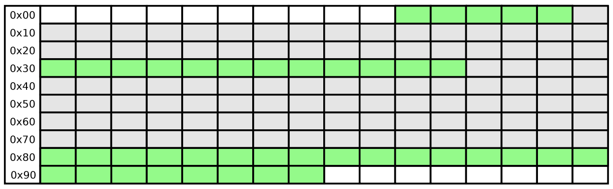


Fig. 5. Improved instruction layout after insertion of 6 `nop` instructions. Instructions now span 4 decoder lines only.

g) Branch Alignment: We found an example where a two-deep loop nest of two short running loops would lead to the placement of the back branches close to each other at the bottom of the loops, as shown in this code sequence:

```
---------------    # 32-byte alignment boundary
 je l1
 add $1, %eax
 add $2, %ebx
 je l2
```

In many Intel platforms, branch predictor structures are indexed by `PC >> 5`. As a result, the backward branches of both the loops above use the same branch prediction information. Since both loops were short running with iteration counts of 1 or 2, the branch predictor gets constantly confused and makes consistently bad predictions. Moving the second branch instruction down via NOP insertion so that the two branch instructions with targets `l1` and `l2` have two different `PC >> 5` values speeds up a full image manipulation benchmark by 3%.

h) Alias Issues: There are other alignment specific alias issues, as many hardware features, e.g., the prefetchers, use tables indexed by address bits at certain granularities, leading to alias effects. For example, on a specific Intel platform prefetchable loads should not be located at multiples of 256 bytes. We have not yet implemented a pass to address this issue.

D. Scalar Optimizations

We added a few scalar optimizations as well, e.g., for unreachable code elimination and constant folding. There is typically not much opportunity left in compiler generated output files. However, as we seek to make MAO useful in simple code generators, offering a standard set of scalar optimizations appears valuable.

E. Experimental Optimizations

i) Nopinizer: Inspired by ideas from Diwan [11], this pass inserts random sequences of `nop` instructions in the code stream. A random number seed can be specified to produce repeatable experiments. Furthermore, the insertion density can be specified as a random function of `nop's` pre-existing instructions, as well as the length of the NOP sequences.

The idea is that by inserting `nop` instructions, code gets shifted around enough to expose micro-architectural cliffs, e.g., by removing unknown alias constraints, or limitations in the branch predictor. Performing a large number of experiments found a 4% opportunity in compression code on an older Pentium 4 platform, which as of today, remains a mystery.

j) Nop Killer: The compiler inserts alignment directives based on some rough ideas about an underlying microarchitecture. For example, it tries to align branch targets to 8 or 16 bytes. The assembler generates `nop` instructions of varying length (e.g., `xor %eax, %eax` for the assembly `.align` or `.p2align` directives).

The question was how effective these alignment directives actually are. This pass allows removal of all these `nop` instructions. We found that on several Intel and AMD platforms, the performance effects were in the noise range for our set of benchmarks. The pass resulted in a code size improvement of about 1%.

k) Inverse Prefetching: On Intel Core-2 platforms, a load instruction can be turned into a non-temporal load by inserting a `prefetch.nta` instruction to the same address before it. This results in these loads always replacing a single way in the associative caches. This technique can be used to reduce cache pollution. We used a novel memory reuse distance profiler to identify loads with little reuse and used MAO to insert the `prefetch` instructions to make these loads non-temporal. The implementation overhead was minimal and dramatically simpler than attempting to do this in the compiler. Results of this technique are promising and will be detailed in another paper.

l) Dynamic Instrumentation Support: Dynamic binary instrumentation is a complicated task, in particular in the presence of variable-sized instructions. For example, if the instrumenter wants to modify existing code and instrument a branch to trampoline code to perform a specific task, existing instructions need to be modified, and, e.g., a 5-byte branch must be inserted. This must be an atomic operation, otherwise one thread could be executing in this 5-byte range as it is being replaced. Instrumenters can work around this issue by monitoring all threads, and single-stepping threads out of instrumentation points to allow insertion of branches.

A simpler approach is to guarantee that single 5-byte (`nop`) instructions reside at the desired instrumentation points, and that those instructions do not cross cache lines. MAO offers an experimental pass that performs this transformation at all function entry and exit points to allow a specific form of dynamically instrumented function profiling.

It would have been a significant effort to facilitate implementation and deployment of this prototype in a compiler.

Writing this transformation in MAO took engineers not familiar with MAO only a few days and it could be deployed immediately, allowing further experimentation on overhead, code-size issues, and of course the profiling results themselves. Remarkably, while the insertion of the `nop` instructions was expected to result in degradations because of larger I-cache footprint and added instructions, it actually resulted in no degradations overall, as well as an unexpected 8% improvement in an image processing benchmark. This is due to an alignment effect and is not fully understood at time of this writing.

m) Instruction Simulation: In another effort we implemented a sampling-based race detector. The idea [19] is to construct probabilistic lock sets by sampling instructions with memory addresses and simultaneously tracking instrumented locks. This sampling based approach reduces overhead by an order of magnitude compared to existing approaches. The same paper shows that the probability of finding races can be increased by increasing the number of sampled addresses. Instead of increasing the sampling frequency, which could be prohibitive in regards to runtime overhead, we implemented a MAO pass performing forward and backward instruction simulation, handling only a small subset of all instructions. While simple, this pass is able to generate additional addresses following address calculations using register content snapshots.

For example, consider this instruction sequence:

```
IP1:   mov  -0x08(%rbp), %edx
IP2:   mov  %edx, (%rax)
IP3:   addl 0x1, -0x4(%rbp)
```

For each PMU sample, we also get the content of the register file for the sampled instruction. Assume the hardware samples on instruction `IP1`. Since the value of `%rax` is not being killed by this instruction, and since we got this register's value when we sampled `IP1`, we can use this register's content to compute the address used in instruction `IP2` via simple forward simulation. Similarly, assume we only sampled instruction `IP3`. Since we also got the value of register `%rax` at this point, we can do a backward simulation and get the address used at `IP2` as well.

Using this technique, for the benchmarks presented in this paper, the number of sampled effective addresses could be increased by factors ranging from 4.1 to 6.3.

F. Scheduling Optimizations

We found significant performance opportunity (21%) in one of our hashing micro benchmarks, simply from scheduling instructions differently. Consider the following code snippet found in that benchmark:

```
xorl   %edi, %ebx
subl   %ebx, %ecx
subl   %ebx, %edx
movl   %ebx, %edi
shrl   $12, %edi
xorl   %edi, %edx
```

The `xorl %edi, %ebx` feeds three other instructions. Performance differences were observed by reordering those three instructions. The instructions in question did not have dependencies among them, had the same latencies, and can all potentially execute in the same cycle without any structural hazards. By analyzing various PMU counters, it was found that the performance degradation correlated with a proportional increase in reservation station stalls as measured by the hardware event `RESOURCE_STALLS:RS_FULL`.

Based on the PMU values, our hypothesis was that there is some bandwidth limitation while forwarding the values from an executed instruction to its dependent instructions. If, due to forwarding bandwidth limitation, the result of the `xorl` instruction does not reach the dependent instructions in the same cycle, that would cause one or more dependent instructions to wait in the reservation stall resulting in an increase in `RESOURCE_STALLS:RS_FULL` events. Since the numbers from various hand-modified schedules supported that hypothesis, we added a scheduling pass in MAO. The pass provides a framework for list-scheduling at the assembly instruction level. By changing the cost functions associated with the instructions, different scheduling heuristics can be implemented. The current cost function ensures that, when scheduling successors of an instruction with multiple fan-outs, the instructions on the critical path are given a higher priority. This resulted in a 15% performance improvement in the hashing microbenchmark and a 0.6% improvement across our benchmark suite.

In addition, we noticed more machine-dependent scheduling opportunities. A basic block in the hot loop had this layout:

```
.L5:
   leal   (%r8, %rdi), %ebx
   movl   %ebx, %ecx
   sarl   %ecx
   movl   %ecx, %edx
   xorb   $01, %dl
   leal   2(%rdx), %r8d
```

The particular Intel chip has execution ports with asymmetric capabilities. The `lea` instruction can only be executed on port 0, the `sarl` instruction can only be executed on ports 0 and 5. We suspect that other micro-operations were competing for the same execution ports, which introduces another interesting scheduling constraint.

There are many mysteries that this scheduling pass is trying to address, without any particular knowledge about the actual effects. Based on these experiences, we decided to write a more automated mechanism to extract and discover micro-architectural features. An ambitious goal is to discover highly beneficial schedules for a target micro-architecture automatically. While we are far from this goal, detecting features automatically remains interesting and the mechanism is described next.

IV. Micro-Architectural Parameter Detection

The success of micro-architecture specific optimizations depends on a good model of the the target micro-architecture. Building an accurate model for modern processors is virtually impossible. The main difficulty lies in the inherent complexity of these processors that makes any attempt to precisely model a processor prohibitively expensive. This is compounded by the fact that processor manuals and other documents released by the processor manufacturers do not completely specify the microarchitecture. For instance, consider the branch predictors in current-generation Intel and AMD processors. While the manuals describe the broad category under which the predictor falls ("two level predictor with global history", for example), they usually do not contain specific information such as the size of the history tables, specific bits used to index the tables, and so on.

The missing details in the processor manuals force architecture-specific tools to rely on third party documentation (such as the one published by Agner Fog [2]) or to discover the parameters by experimentation. This experimentation typically involves crafting microbenchmarks in assembly language, running them in isolation on the target architecture, collecting various performance metrics, and interpreting the results to infer specific parameters of the system. Multiple microbenchmarks may need to be created and each run several times to infer a parameter with high confidence. This makes the process cumbersome, time consuming and less scalable. These difficulties magnify many fold if the parameters of several processors need to be inferred.

To overcome these hurdles, MAO contains a framework to simplify the creation and execution of microbenchmarks. This framework consists of the following key abstractions implemented as Python classes:

a) Processor: This class encapsulates information specific to a target architecture. This primarily consists of the set of registers and the set of instructions.

b) Instruction: This class represents an assembly instruction. The implicit and explicit operands of an instruction, including their types, positions of source and destination operands, and any other operand constraints are managed by this class.

c) InstructionSequence: As the name implies, this class encapsulates an acyclic sequence of instructions. A sequence is specified by the set of candidate instructions that can appear in the sequence and the dependencies among the instructions. The candidate set is described by a set of instruction attributes. The common attributes include instruction templates (such as `'add %r, %r'`) and the type of instructions (arithmetic, memory, etc.). This could be easily extended to support arbitrary attributes. For instance, certain experiments might require the use of only long latency operations. This can be specified by having latency as an instruction attribute and specifying constraints based on that. The dependencies among the instructions are specified using dependence graph types. The supported types include CHAIN (each instruction in the sequence has a RAW dependence on the previous instruction), CYCLE (a CHAIN where the first instruction depends on the last), RANDOM (arbitrary dependencies between instructions) and DISJOINT (each instruction is independent of other). The InstructionSequence class generates a random sequence satisfying the specified constraints.

```
# Form a loop with a cycle of instructions,
# one dependent on the other. Execute the chain,
# collect CPU cycles and obtain the latency
#
def InstructionLatency(proc, template):
  seq = insseq.InstructionSequence(proc)
  seq.SetInstructionTemplate(template)
  seq.SetDagType(insseq.DagType.CYCLE)
  seq.Generate()
  loop_list = loop.LoopList(
              [loop.StraightLineLoop([seq], proc)])
  bench = benchmark.Benchmark(loop_list)
  results = bench.Execute(proc, [proc.CPU_CYCLES])
  insns_in_loop = loop_list.NumDynamicInstructions()
  latency = round(float(results[proc.CPU_CYCLES])/
              insns_in_loop)
  return latency
```

Fig. 6. Program to determine instruction latency

d) Loop: One or more instruction sequences are enclosed within a loop with a specified trip count. The simplest form of a loop is a straight line loop which does not have any control-flow inside the loop. More general loops could be generated by explicitly specifying the control-flow between the different instruction sequences.

e) Benchmark: This class is used to construct an assembly program from the specified loops, assemble the program, execute the program on a target architecture in isolation and collect any specified PMU counters.

A. Case Study: Instruction Latency

Figure 6 shows a simple example of how the framework can be used to determine the latency of an instruction. The function `InstructionLatency` takes two parameters: a target processor and an instruction template. The function generates an instruction sequence where all the instructions have to match the template and each instruction depends on the preceding instruction. This dependence pattern ensures that exactly one instruction is in the execution unit every cycle. The source and destination operands are generated by the framework randomly from a set of valid operands for each instruction. This instruction sequence is wrapped inside a straight-line loop with a fixed trip count. This microbenchmark is executed on a host with the specified target processor in isolation and the number of CPU cycles taken to execute the microbenchmark is obtained. Since the instruction executions are completely serialized, the latency per instruction is obtained by dividing the CPU cycles by the number of instructions in the loop.

V. Performance Analysis

MAO, at the time of this writing, is not yet tuned for either compile-time or runtime performance, and we therefore decided to present performance results in an anecdotal fashion which allows highlighting the *potential* of this infrastructure.

A. Compile-Time Performance

MAO is based on `gas`, which, during normal operation, only performs one "pass" over the assembly instructions. MAO performs multiple passes, e.g., one of for each optimization pass. As a result, one would expect the compile-time to be much higher than for `gas` and indeed, for a typical set of passes, MAO is about five times slower than `gas`. We believe this can be improved significantly. Since assembly time only represents a very small fraction of overall compile time, we find that a full integration of MAO in the compiler, including a final `gas` invocation, only slows down `gcc -O2` by 5-10%, and `gcc -O0` by 25-30%. We find this acceptable as we only use MAO for higher levels of optimization.

For all experiments we integrated MAO the following way. We first configured and built a standard GCC installation. We find the assembler binary named `as` in this directory tree, rename it to `as-orig`, and add a replacement script called `as`, which gets invoked by the compiler driver as if it were the original assembler. The script parses all command-line arguments passed to it. If it finds MAO-specific options, it filters those out and executes MAO first. MAO will generate a temporary assembly output file. The script then starts the original assembler `as-orig` on this output file with the remaining options and generates the expected binary object file. The `as` script is available in the open-source MAO distribution.

B. Runtime Performance

For runtime performance results, we integrated MAO into a stock GCC 4.4.1 compiler as described above. For SPEC 2000 int, and various optimization passes, we find that most performance results stay flat, in particular for the Nop Killer pass (NOPKILL), which leads to the conclusion that most alignment directives are not helping. However, for 252.eon, Nopinizer (NOPIN) and Nop Killer result in unsurprising performance regressions, but removal of redundant tests (REDTEST) also resulted in a regression. As alignment directives stay present in the assembly for this pass, this may point to a problem with small loop alignment, described earlier.

Benchmark	NOPIN	NOPKILL	REDTEST
C++/252.eon	-9.23%	-5.34%	-5.97%

On an Intel Core-2 platform, aligning small loops (LOOP16) has benefits for three benchmarks, while degrading 252.eon, which is counter-intuitive and indicates that further analysis and tuning of these passes is necessary.

Benchmark	LOOP16
C++/252.eon	-4.43%
C/175.vpr	1.25%
C/176.gcc	1.41%
C/300.twolf	1.18%

The same transformation on an AMD Opteron platform appears to be helping a different set of benchmarks, yet still degrades 252.eon:

Benchmark	LOOP16
C++/252.eon	-5.86%
C/181.mcf	2.47%
C/186.crafty	2.45%

The picture looks similar for SPEC 2006, where we see a few low percentage swings both to the upside and downside. However, on an AMD Opteron platform, FP 454.calculix improves significantly (over 20%) with either removal of redundant move instructions (REDMOV) or removal of redundant test instructions (REDTEST). Since both passes only remove instructions, we suspect that another second order effect takes hold, such as the loop stream detector. However, we are not aware of a published LSD-like structure on AMD platforms, therefore this result points to yet another unknown micro-architectural effect.

The same effect, but to a lesser extent, can be seen for 447.dealII. Interestingly, simply removing alignment directives results in an 8.8% degradation for 454.calculix, again indicating that more careful optimization must be applied.

Benchmark	REDMOV	REDTEST	NOPKILL
447.dealII	2.78%	3.21%	-0.12%
454.calculix	20.12%	20.58%	-8.81%

Scheduling (SCHED) helps a few benchmarks as well. The gains are still modest as this pass does single basic block scheduling only. We expect the impact to become much higher once we extend the pass to schedule across basic blocks.

Benchmark	SCHED
410.bwaves	1.29%
434.zeusmp	1.20%
483.xalancbmk	1.25%
429.mcf	1.43%
464.h264ref	1.75%

The table in figure 7 shows how often some of the basic optimization passes transformed the code as generated by a stock GCC 4.4.3 for SPEC 2000 int. It also shows the aggregate performance results, which were obtained on a Intel platform. In order to obtain consistent results, we ran the SPEC benchmarks more often than the three suggested times and performed statistical valuation, ensuring that the results were statistically significant. The aggregate numbers are lower than numbers obtained from applying single optimizations. The results, again, show the potential for this approach, but more work needs to be done to consistently exploit the opportunities. Excluding the degradation for `253.perlbmk`, this set of transformations would have resulted in an overall performance gain of 0.6%.

VI. Related Work

There has been considerable past work on low-level and post-link optimizers. An early assembly to assembly optimizer was described in Sites [20]. ALTO [16], PLTO [18], FDPR-Pro [24], and Ispike [9] are post-link optimizers for Alpha, x86, POWER, and Itanium, respectively. In all cases, the reasons for post-link optimization are fairly consistent: performing optimizations that are hard at higher levels of abstraction (e.g., jump optimizations based on actual instruction addresses), optimizing code for which source code is unavailable, re-targeting code to take advantage of new architectural or micro-architectural features (e.g., vector instructions [6]), or incorporating profile-feedback to re-optimize. Some past approaches even use pattern matching like MAO [12]. While this list of low-level optimizers and reasons to use them

Benchmark	L	NOP	M	T	SCHED	Perf
164.gzip	1	664	-	5	427	+0.02%
175.vpr	3	1425	7	4	1778	+1.06%
176.gcc	62	27471	35	57	8891	+1.29%
181.mcf	-	185	1	-	236	+0.13%
186.crafty	3	1987	7	18	2648	+0.43%
197.parser	13	2134	4	-	1106	+0.18%
252.eon	1	2373	10	6	12215	+1.01%
253.perlbmk	21	11870	9	21	5178	-2.14%
254.gap	62	9216	23	9	6466	+0.12%
255.vortex	1	6860	3	5	6905	+0.44%
256.bzip2	2	396	3	-	637	+1.04%
300.twolf	18	3009	24	43	2800	+0.97%
					Geomean	0.38%
					Geomean w/o 253.perlbmk	0.61%

Fig. 7. Number of optimizations performed, and aggregate performance gains. L: Small loop alignment (LOOP16), NOP: Nopinizer, M: Redundant Mov Removal (REDMOV), T: Redundant Test Removal (REDTEST), SCHED: Instructions moved during scheduling

is by no means exhaustive, MAO is unique in that it is an accessible low-level optimizer designed to help software developers (both compiler experts and non-experts) navigate the complex landscape of performance cliffs.

MAO is not unique in its goal to try and extract peak performance from an often opaque micro-architecture. The literature is replete with work recognizing the difficulty of achieving optimal performance. For example, Mytkowicz et al. demonstrate that subtle changes to an experimental setup, such as changing the link order or the UNIX environment a program runs in, can dramatically affect application performance [17]. While the authors' goal was to demonstrate the significant effect of measurement bias, this work clearly demonstrates the significance of performance cliffs. In a later work, the authors propose blind optimization [11] which does an automatic search of a variant space to find the best performing program. Significant work exists in this space of iterative compilation [5], [10], [1], [22], [23]. Unfortunately, iterative compilation has thus far not taken hold for production compilation. MAO offers developers an opportunity to benefit from the micro-architectural optimization potential that these iterative approaches seek to exploit.

VII. Conclusions

We presented MAO, an extensible micro-architectural optimizer. We discussed the features of the IR and several classes of optimization passes. We believe that these issues must be addressed more aggressively in the future, both for performance results, but also for robustness and quality of the results. We motivated an automatic approach to hardware feature detection. We believe the results point to a potentially rich area of research and we invite participants from academia and industry to contribute to this open-source project for the benefit of the community.

VIII. Acknowledgments

We would like to thank the anonymous reviewers for their numerous suggestions to improve this paper.

References

[1] Bas Aarts, Michel Barreteau, Franois Bodin, Peter Brinkhaus, Zbigniew Chamski, Henri-Pierre Charles, Christine Eisenbeis, John R. Gurd, Jan Hoogerbrugge, Ping Hu, William Jalby, Peter M.W. Knijnenburg, Michael F.P. O'Boyle, Erven Rohou, Rizos Sakellariou, Henk Schepers, Andre Seznec, Elena A. Stohr, Marco Verhoeven, and Harry A.G. Wijshoff. OCEANS: Optimizing compilers for embedded hpc applications. In *Lecture Notes in Computer Science*, August 1997.

[2] Agner Fog. www.agner.org.

[3] Dehao Chen, Neil Vachharajani, Robert Hundt, Shih-wei Liao, Vinodha Ramasamy, Paul Yuan, Wenguang Chen, and Weimin Zheng. Taming hardware event samples for FDO compilation. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 42–52, New York, NY, USA, 2010. ACM.

[4] Chris Lattner, Private Communication.

[5] Keith D. Cooper, D. Subramanian, and L. Torczon. Adaptive optimizing compilers for the 21st century. October 2001.

[6] Anshuman Dasgupta. Vizer: A framework to analyze and vectorize Intel x86 binaries. Master's thesis, Rice University, 2003.

[7] GCC, the GNU Compiler Collection. gcc.gnu.org.

[8] Paul Havlak. Nesting of reducible and irreducible loops. *ACM Trans. Program. Lang. Syst.*, 19(4):557–567, 1997.

[9] Chi keung Luk, Robert Muth, Harish Patil, Robert Cohn, and Geoff Lowney. Ispike: A post-link optimizer for the Intel Itanium architecture. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization*, pages 15–26, 2004.

[10] T. Kisuki, P. M. W. Knijnenburg, M. F. P. O'Boyle, F. Bodin, and H. A. G. Wijshoff. A feasibility study in iterative compilation. In *Proceedings of the International Symoposium on High Performance Computing*, 1999.

[11] Dan Knights, Todd Mytkowicz, Peter F. Sweeney, Michael C. Mozer, and Amer Diwan. Blind optimization for exploiting hardware features. In *Proceedings of the 18th International Conference on Compiler Construction*, pages 251–265, Berlin, Heidelberg, 2009. Springer-Verlag.

[12] Rajeev Kumar, Amit Gupta, B. S. Pankaj, Mrinmoy Ghosh, and P. P. Chakrabarti. Post-compilation optimization for multiple gains with pattern matching. *SIGPLAN Not.*, 40(12):14–23, 2005.

[13] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization*, March 2004.

[14] John Levon. *OProfile Manual*. Victoria University of Manchester, 2004.

[15] MAO - An Extensible Micro-Architectural Optimizer. http://code.google.com/p/mao.

[16] Robert Muth. *ALTO: A Platform for Object Code Manipulation*. PhD thesis, University of Arizona, 1999.

[17] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing wrong data without doing anything obviously wrong! *SIGPLAN Not.*, 44(3):265–276, 2009.

[18] Benjamin William Schwarz. Post link-time optimization on the Intel IA-32 architecture. Master's thesis, University of Arizona, 2001.

[19] Tianwei Shen, Neil Vachharajani, Stephane Eranian, Robert Hundt, Wenguang Chen, and Weimin Zheng. RACEZ: a lightweight and non-invasive race detection tool for production applications. In *ICSE '11: Proceedings of the 33rd International Conference on Software Engineering*, 2011.

[20] R. L. Sites. Instruction ordering for the cray-1 computer. 27-cs-023. Technical report, 1978.

[21] The Open64 Compiler Suite. www.open64.net.

[22] Spyridon Triantafyllis, Manish Vachharajani, Neil Vachharajani, and David I. August. Compiler optimization-space exploration. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 204–215, Washington, DC, USA, 2003. IEEE Computer Society.

[23] Michael E. Wolf, Dror E. Maydan, and Ding-Kai Chen. Combining loop transformations considering caches and scheduling. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 274–286, Washington, DC, USA, 1996. IEEE Computer Society.

[24] Yaakov Yaari. Post-link optimization for Linux on POWER, http://www.alphaworks.ibm.com/tech/fdprpro.