Facilitating the Bootstrapping of a New ISA

Abigail Mortensen
Computer Science
Florida State University
Tallahassee, Florida, USA
mortense@cs.fsu.edu

Scott Pomerville
Department of Computer Science
Michigan Technological University
Houghton, Michigan, USA
skpomerv@mtu.edu

David Whalley
Computer Science
Florida State University
Tallahassee, Florida, USA
whalley@cs.fsu.edu

Soner Onder
Department of Computer Science
Michigan Technological University
Houghton, Michigan, USA
soner@mtu.edu

Gang-Ryung Uh
Computer Science
Florida State University
Panama City, Florida, USA
guh@fsu.edu

Abstract

Implementation of a new instruction set architecture (ISA) is a non-trivial task that involves significant modifications to the system software, such as the compiler, the assembler, and the linker. This task also includes modifying and verifying functional and cycle accurate simulators to facilitate performance evaluation of programs under the new ISA. Isolating errors in these software components becomes extremely challenging and demands automated and semi-automated mechanisms since neither the compilation infrastructure nor the simulation infrastructure can be trusted as both parties have been heavily modified. Bootstrapping a new ISA is very common in embedded systems since there is a greater variety of embedded ISAs due to often not having a need to support backward compatibility of executables. In this paper, we present the tools and the verification mechanisms we have implemented to support the development of a number of related, but distinct ISAs. Our work in developing the system software and simulators for these ISAs demonstrate that a step-by-step semi-automated approach which relies on simple *invariants* can facilitate effective bootstrapping of the complete system software and the simulator infrastructure.

CCS Concepts: • **Software and its engineering → Retargetable compilers**; • **Computer systems organization → Pipeline computing**.

Keywords: instruction set architecture, instruction set simulation, compiler optimization

LCTES '23, June 18, 2023, Orlando, FL, USA

ACM ISBN 979-8-4007-0174-0/23/06…$15.00
https://doi.org/10.1145/3589610.3596282

ACM Reference Format:
Abigail Mortensen, Scott Pomerville, David Whalley, Soner Onder, and Gang-Ryung Uh. 2023. Facilitating the Bootstrapping of a New ISA. In *Proceedings of the 24th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '23), June 18, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3589610.3596282

1 Introduction

Development of a new instruction set architecture (ISA) is a non-trivial task in itself, involving many decisions and compromises regarding instruction encoding and the instruction set features to be supported. However, once the ISA is designed, a harder challenge arises. The ISA needs to be implemented in the system software, such as the compiler, the assembler and the linker. In addition, new simulators have to be developed or existing ones need to be modified and verified. All of these software components are very large pieces of software that are required to handle millions of lines of input code and still flawlessly execute.

Desktop computers are dominated by the x86 ISA and its extensions that are backward compatible so that software (executables) that users have purchased in the past can execute on microprocessors that support extended x86 ISAs. In contrast, embedded systems have a much greater variety of ISAs since both the embedded microprocessor and embedded software are often packaged together and there is no need to support backward compatibility. In addition, the embedded software development environment will typically not be the same as the embedded execution environment. Hence supporting simulators for ISAs in embedded systems is crucial. Thus, boostrapping new ISAs is much more common for embedded systems than for conventional processors.

There are many challenges in the development process of a simulation and compilation infrastructure for a new ISA: (1) Verification of the compiler requires running the generated code using a verified functional simulator. Verification of a functional simulator requires a correctly functioning compiler to generate code to test the simulator. It

would appear both a compiler and simulator have to be developed before any meaningful testing of the new ISA can commence, creating a chicken-and-egg problem. (2) Simulators may not always correctly function. Even a simple error in the specification of a single instruction in a simulator may cause an error that can propagate thousands or even millions of cycles before it causes an observable failure, such as a crash. Isolating an incorrectly implemented instruction is even more difficult to debug when it only causes the program to produce incorrect results. The situation is worse for debugging cycle accurate simulators, since they require disproportionally longer simulation times. (3) Verification of the compiler generated code is challenging even with a verified functional simulator. An incorrectly implemented code-improving transformation may correctly work for many cases, but may occasionally generate incorrect code depending on the context. Identification of the problem in most cases requires locating the region of the code in the input program where the error has occurred.

In this paper, we discuss our experience in dealing with these challenges and present the solutions we have developed during the development of our ISA, called *SCALE* [2]. The contributions of this paper are general techniques to facilitate the bootstrapping of a new ISA. These techniques include (1) constructing a simulator without a compiler, (2) automatic isolation of simulation errors, and (3) automatic isolation of code-improving transformation errors.

2 Initial Bootstrapping of a New ISA

One needs systematic and automatic methods for isolating errors in both simulators and compilers when bootstrapping a new ISA. Any systematic mechanism must rely on some invariants that the developers can trust. Clearly, isolating simulation errors is simplified when an existing simulator can be used as the basis for comparison. Likewise, isolating compilation errors is simplified when an existing compiler can correctly generate code to facilitate isolating problems. However, neither a correctly functioning simulator, nor a correctly functioning compiler exist for a new ISA.

Our solution to this problem is to utilize a verified simulator and a compiler for a different ISA, which we call the *base ISA*. We rely on the invariance of control-flow and the premise that the data memory should receive the same updates in both simulators for validation.

For our initial bootstrapping implementation, we choose the MIPS ISA as our base ISA and translate MIPS instructions into our target SCALE ISA using assembly macros, which results in an assembler that accepts verified MIPS code and generates the binary in the target ISA, as shown in Figure 1. In a sense, this assembler is *bilingual* since it can accept two different sets of assembly instructions. Since every MIPS instruction is defined to be a macro, any MIPS instruction is translated to the target ISA, but target ISA instructions are directly handled and encoded in binary. Furthermore, the input program can be a mixture of MIPS and SCALE instructions. Having the assembler accept a mixture of base and target ISA instructions significantly helps the compiler development as well, which will be subsequently described. This technique of using assembly macros will only work if the differences between the two ISAs are such that macros are able to translate each base ISA instruction to a corresponding sequence of target instructions. If more substantial changes are needed, such as different calling conventions, then this approach will not work.

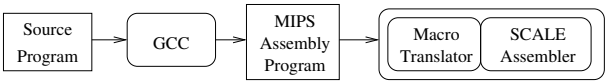


Figure 1. Compiling Using a *Bilingual* Assembler

Of course, the macro-translator needs to be verified and its verification would be quite challenging without the use of a verified functional simulator for the target ISA. Since that simulator is being developed, we use the verified simulator for the base ISA to validate both the assembly macro translator and the functional simulator for the new ISA. During debugging, we run the MIPS functional simulator side-by-side with the SCALE functional simulator such that they are connected by a data pipe through which the MIPS simulator sends control-flow information and values flowing to the memory, as shown in Figure 2. Matching the control-flow exercised and the data values communicated to the memory allows us to locate the errors before they propagate too far since an erroneous instruction implementation often causes a nearby branch instruction to change direction, or, alter values flowing to the memory. As a result, we can use the same assembly program generated by a verified compiler (GCC) for an existing ISA (MIPS) to validate the new functional simulator (SCALE), hence breaking the chicken-and-egg problem.

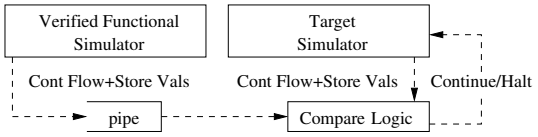


Figure 2. Simulator Verification Design Layout

For developing the new compiler for the SCALE ISA, we again rely on translating assembly instructions into the target ISA. Contrary to the typical compiler back-ends, our back end, *asopt* (assembly optimizer) translates MIPS assembly instructions to SCALE assembly instructions. In this respect, *asopt* embodies the facilities of the entire back-end of a typical compiler, but also incorporates a simple translator which translates from the base ISA to the target ISA. Since our functional simulator is bilingual, the translator part of *asopt* could be developed one instruction at a time as any untranslated instruction is automatically translated through the use of macros by the bilingual assembler to the target code. Furthermore, disabling all optimizations allows us to first verify the

correctness of the translation from MIPS to the target ISA using the verified functional simulator, and then progressively develop and apply machine specific optimizations.

3 Simulation/Compilation Environment

In this section, we describe some features of the SCALE ISA that is the target of our compilation system and is input to our simulators. We next delineate our infrastructure for producing simulators, assemblers, and linkers. We finally summarize our compilation system used for this project.

3.1 The SCALE ISAs

This research is part of the NSF SCALE project, which involves developing both simulation and compilation support for a set of related, but distinct ISAs [2]. We describe the first two ISAs implemented in this project, which are the *SCALE base* ISA and the *SCALE VLIW* ISA. We present some details about these ISAs so the reader can grasp the scope of the problem of bootstrapping these SCALE ISAs in our study.

The SCALE instruction sets were designed to communicate compiler computed information to the micro-architecture through instruction annotations. Such annotations include statically computed scheduling commands as well as additional hints to the micro-architecture. As a result, SCALE ISAs have quite different instruction formats and encodings than typical RISC ISAs, such as MIPS or RISC-V. All SCALE loads and stores support only the register indirect addressing mode, meaning that memory operations in SCALE ISAs are performed without a displacement from a base register. While the primary motivation for this approach is extracting the necessary annotation encoding space, this approach also decreases the number of stages in the instruction pipeline. In fact, many processors will dynamically split a load or store into an address generation instruction and a memory reference instruction anyway and treat them as separate instructions within the processor. By splitting these operations at compile time, redundant effective address calculations can sometimes be eliminated. We view the amount of differences between the MIPS and SCALE ISAs to be no less than the amount of differences between the MIPS and RISC-V ISAs.

VLIW architectures were introduced in the early 1980s [8] and are still used in some domains like digital signal processing (DSP) where instruction-level parallelism (ILP) can be explicitly exploited by a compiler. The generated VLIW code must be packaged into groups of independent operations encoded as a single instruction, hence enabling these operations to be simultaneously issued. We refer to such groups of operations as *VLIW packs* and the position of an operation within a VLIW pack as a *lane*. Our ISA permits the number of operations in each VLIW pack and which types of operations can be placed in each lane to be configurable at compile time. As a result, the SCALE VLIW ISA uses instructions defined by the SCALE base ISA, but supports VLIW execution.

We permit operations to follow a branch within a SCALE VLIW pack. An operation after one or more branches within a VLIW pack is only committed if the preceding branches are found to be not taken. These features, permit simultaneous execution of operations specified by each instruction in the VLIW pack to have the same semantics as the sequential handling of instructions, one at a time, left to right in a VLIW pack. Hence, we can utilize a functional simulator that processes a single instruction at a time for debugging our compiler back end, *a*sopt. Note that, supporting multiple conditional branches in a single VLIW instruction has been used in even some of the earliest designed VLIW processors [8]. Only *nop* instructions should be placed after an unconditional transfer of control within a VLIW pack.

3.2 ADL Simulation System

We use the *ADL* simulation system, which takes a microarchitecture specification file as input and automatically produces an assembler, linker, and disassembler [10]. ADL provides constructs for specifying (1) microarchitectural features including pipelines, control, and memory hierarchy and (2) the instruction set architecture including the assembly syntax and corresponding binary representation. The assembler and linker are used to produce a statically linked executable that is invoked by the ADL produced functional or cycle accurate simulators. These cycle accurate simulators perform a more realistic simulation than many commonly used simulators as instructions and data are actually fetched from the instruction and data caches, target addresses are actually fetched from a branch target buffer, values are actually forwarded through the pipeline, etc. A more realistic simulation helps to ensure that the described techniques are correctly implemented and hence provides more reliable statistics.

The SCALE base ISA is used for a SCALE functional simulator, a SCALE pipelined simulator, and a SCALE out-of-order (OoO) simulator. The SCALE VLIW ISA is used for the SCALE VLIW simulator. The SCALE functional simulator is the fastest simulator and is just used to check if the simulation provides correct results for the generated code. The SCALE pipelined simulator provides a six stage integer pipeline, which includes the stages IF (Instruction Fetch), ID (Instruction Decode), RF (Register Fetch), EX (EXecution) or MEM (MEMory access), and WB (Write Back). Note the MEM stage is performed in the same cycle as the EX stage as each load and store does not have a displacement for the base register and hence does not require the calculation of an effective address. As a result, the depth of the integer pipeline is five stages. The simulated architecture also incorporates a separate, three stage floating-point (FP) pipeline. A SCALE assembly file can be simulated by the SCALE functional, pipelined, and OoO simulators with no change in how the file is produced. Instructions within a VLIW pack are fetched, decoded, and executed together. The SCALE VLIW simulator

is pipelined like the SCALE pipelined simulator, but forwarding and stalls occur across all lanes between VLIW packs of instructions. If any instruction within a VLIW pack stalls, then all instructions in the VLIW pack are stalled. An instruction that has an antidependence with a previous instruction is never scheduled before the previous instruction in a VLIW pack. A group of instructions in a pack can be executed either simultaneously or one at a time in left to right order, as would be the case with functional simulation. The SCALE pipelined, SCALE VLIW, and SCALE OoO simulators include the simulation of caches, branch target buffers (BTBs), and various branch predictors (BPs).

3.3 Compilation System

The compilation support that we need for this project must support low-level code generation and optimizations. Figure 3 shows how we generate code for the various SCALE ISAs. We use *gcc* to produce MIPS assembly files, which allows us to compile files in a variety of source languages, such as C, C++, and Fortran, and also leverage the optimizations that are provided by the *gcc* compiler. We developed our own assembly optimizer, called *asopt*, that takes an assembly file as input, translates instructions when necessary to a new instruction set, performs a variety of analyses and optimizations, and produces modified assembly code as output. In order to properly determine which registers are live at any given point in a function, we need to know which registers are being passed to each function that is being called and which register if any is used to return a value. An optimizer implemented either in a linker or a run-time environment would attempt to perform interprocedural analysis to determine this information. We instead gather information as a side effect of the *gcc* compilation. We use an option in *gcc* to produce a *.gkd* file that contains information about each *gcc* RTL (instruction), which we use to determine which registers are passed as values in function calls. We also generate a MIPS object file with symbolic debugging information from which we generate an *.objdump* file, which contains information about the function return type that is used to determine in which register a return value is placed. Both the *.gkd* and *.objdump* files are input to our *geninf* tool that produces a *.inf* file that could be easily parsed by *asopt*. The assembly optimizer, *asopt*, reads both the *.inf* file and the MIPS assembly file to produce the SCALE assembly file. Various flags can be passed to *asopt* to both select the optimizations to be performed and to select the ISA for the assembly target file.

```
function qcount int
calls fopen $4 $5
```

Figure 4. Ex .inf File

Figure 4 shows a few lines from an *.inf* file, which contains only two types of lines. A *function* line contains a return type used to indicate which register contains the return value and could be live at the point of each function return. A *calls* line indicates which argument registers are live at the point of a function call. We generate a *.inf* file as *asopt* also needs

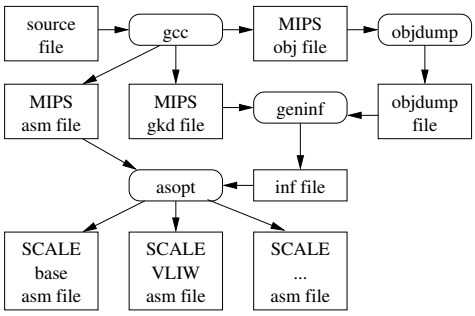


Figure 3. SCALE Code Generation Process

to process some hand-written assembly files in the libraries we utilized. Most compilers would require that all code to be processed be written at the source code level and any hand-written assembly files would have to be modified by hand to reflect changes in the ISA. In contrast, we only have to generate a simple corresponding *.inf* file by hand for each hand-written assembly file.

For each function, *asopt* reads in the instructions, identifies the type of each instruction and which registers are set and used, and builds the control flow graph. It then translates each MIPS assembly instruction to its corresponding SCALE assembly instruction when the input MIPS instruction is not in the SCALE instruction set. It also expands any pseudo instructions so that each SCALE assembly instruction has a one-to-one mapping with a SCALE machine instruction. This step is necessary when performing some low-level optimizations. Having a one-to-one mapping between assembly and machine instructions also enables many additional optimizations that *asopt* can exploit.

4 Isolating Simulator Errors

The ADL simulation system supports cycle accurate descriptions of simulators. With detailed descriptions comes the need for catching potential issues relatively early on in the code, since code may not exhibit irregular behavior until millions or even billions of other instructions have retired.

The SCALE functional simulator was the first simulator developed in the SCALE architectures. This simulator behaves similarly to a single-stage processor that executes the semantics for an instruction before progressing to the next instruction. We can verify correctness to a high degree of confidence by executing this simulator on unoptimized SCALE code and comparing the results to the output of equivalent code running on a MIPS simulator. This functional simulator serves both as a tool to pinpoint bugs with optimizations in *asopt*, as well as a tool that can be used to identify the exact point in a cycle-accurate simulator when some instruction produces a result that diverges from the proper value.

In the development for the pipelined simulator, which uses the same ISA, there are several simulation variables that can be tracked throughout the lifetime of a program, despite differences in architecture. Chief among these is the control

flow of a program in the simulator. In programs without randomness, the path taken through a program is deterministic, and minor changes in a program can drastically alter control flow. For these deterministic programs, an error isolator for the pipelined simulator can compare the program counter of retiring instructions against the dynamically created trace of the program from the functional simulation to determine proper execution. However, in longer simulations that execute potentially hundreds of billions of instructions, this trace becomes space-probitive. So, instead, we dynamically create the trace of the simulator.

Figure 2 shows the basic layout of this verification process. We run two simulators in parallel: the simple SCALE functional simulator and the SCALE pipelined simulator. Since ADL simulators can load the simulated program starting at a given virtual memory address, we can enforce the two simulators to have the same virtual address space, as well as to simulate the same program executable with identical inputs. As a result, each instruction processed by the two simulators always has the same program counter value.

We execute the functional simulator alongside the target simulator. The functional simulator, when retiring an instruction, inserts the simulated instruction's program counter (PC) into a blocking pipe. The target simulator at instruction retirement will then check the PC of the retiring instruction against the value at the pipe head. We compare instructions at retirement as the pipelined simulator can potentially mispredict branches. If at any point the program counters diverge, the compare logic sends a signal to halt execution so that the program's failure can be analyzed.

Since the functional simulator is small, overhead for functional verification is minor, as the only overhead is the comparison of retiring values to the corresponding values in the pipe from the functional simulator. Reading from a pipe is equivalent to reading from a trace file if one were to exist. The functional simulator's simple design allows it to place values into the named pipe faster than the target simulator can produce them while using minimal resources. So this target simulator error isolation technique will effectively never find the functional simulator to be a bottleneck.

This technique has also been expanded to support other key points of potential divergence. Examples of other divergences are verifying the value of the destination register of the instruction or verifying the values stored and retrieved by memory operations. This use case can be helpful in scenarios where control flow is highly independant of the data input provided. As long as instructions are compared during retirement, the basic logic remains the same.

The VLIW simulator executes packs of SCALE base instructions, and as such, the *asopt* must properly order the SCALE base instructions within VLIW packs. By treating each SCALE base instruction within a VLIW pack as a full instruction, it is possible for the functional simulator to execute the SCALE VLIW code. The key restriction is that antidependencies between instructions within a pack must remain ordered. This will allow SCALE base instructions to be sequentially and correctly executed within a pack. Instructions following a branch in SCALE VLIW packs are only committed when a branch is not taken. By following these restrictions the SCALE functional simulator can correctly simulate the VLIW code and remains a viable tool for detecting simulator errors.

5 Isolating Assembly Optimizer Errors

asopt simulation testing began with code generated by the assembly optimizer, but with applying no transformations. Note at this point each MIPS instruction not supported by the SCALE ISA is treated as a macro by the SCALE assembler. *asopt*'s pseudo expansion phase was then applied and simulated. Once all benchmarks transformed by pseudo-expansion produced the correct simulation results, this phase was considered correct and another phase could be tested in the same manner. An important aspect of the *asopt* testing process is that the correctness of one type of optimization can be tested before testing another optimization phase.

5.1 Compiler Transformations and Phases

A *code-improving transformation* consists of a sequence of changes where the semantic behavior of the code should remain the same, although its performance may be enhanced. A code-improving transformation can be viewed as optional, as compared to a required transformation that is needed for correct execution. An *optimization phase* consists of the application of zero or more transformations of the same type.

Table 1 enumerates the different types of phases that can be applied within our assembly optimizer. These transformations were developed as needed based on the output of the *gcc* compiler. We only included assembly optimizer phases that actually had opportunities to change the code. For instance, unreachable code elimination is not included as *gcc* never produces code where any instructions are unreachable.

Table 1. Types of Assembly Optimizer Phases

Phase Name	Acronym	Phase Name	Acronym
Accumulator Expansion	AE	Reorder Commut Opers	RCO
Copy Propagation	CP	Remove Empty Blocks	REB
Common Subexpr Elim	CSE	Remove Incr True Deps	RITD
Dead Assignment Elim	DAE	Remove Useless Insts	RUI
Expand Pseudo Insts	EPI	Split Basic Induction Var	SBIV
Loop Inv Code Motion	LICM	Save/Restore New Regs	SRNR
Loop Unrolling	LU	VLIW Block Scheduling	VBS
Merge Basic Blocks	MBB	VLIW Fill Slots	VFS
Merge Increments	MI	VLIW Global Scheduling	VGS
Redundant Asg Elim	RAE		

We now briefly describe some of the assembly optimizer phases that may not be commonly known. AE changes an innermost loop that has multiple commutative operations (+, *, |, &), where the same register is only used in these operations. After AE we use a different register for each operation so there are no dependences between these operations in the loop. This transformation also requires changing the

prologue and epilogue of the loop. EPI expands a pseudo assembly instruction into multiple assembly instructions, which is required when generating VLIW code. RITD removes true dependences for instructions that either add a constant to a register or are a move instruction, where the dependent instruction uses the result of an increment instruction. SBIV splits a basic induction variable in an innermost unrolled loop where there are multiple increments of the basic induction variable. This transformation removes true dependences between these increments and removes antidependences between an increment and the uses of a prior increment. SRNR saves and restores a newly allocated, but previously unused, callee-save register.

Figure 5 shows some example transformations applied by the assembly optimizer. Each subfigure shows an assembly instruction in blue if the instruction will be modified or deleted in the next subfigure and in bold if the instruction has been modified or was inserted compared to the previous subfigure. Figure 5(a) shows original assembly code consisting of three MIPS assembly instructions. The *lw* and *sw* assembly instructions are pseudo instructions since a global address of a variable, such as *gcnt*, cannot be encoded in a 32-bit instruction. Figure 5(b) shows the assembly code after expanding the *lw* instruction. The *lalui* and *laori* instructions are SCALE pseudo instructions that represent a global address as the third argument. The *lalui* instruction indicates that the most significant 16 bits of *gcnt* should be loaded into *$2*. Likewise, the *laori* instruction indicates that the least significant 16 bits of *gcnt* should be ored with *$2*. The register *$2* is used to hold the result of the *lalui* instruction since the original *lw* instruction was writing to *$2*, which makes *$2* available for use at that point. The ADL assembler/linker will convert the *lalui* and *laori* assembly instructions into *lui* and *ori* machine instructions, respectively, once the global address of *gcnt* is known. Figure 5(c) shows the assembly code after expanding the *sw* instruction. In this case *$1*, the MIPS assembler temporary register, is used since *$2* is live at the point the *sw* is executed. Figure 5(d) shows the assembly code after the assembly optimizer applies common subexpression elimination. The destination register of the first *lalui* is changed to be *$1* so that the value of the 16 most significant bits of *gcnt* address is not overwritten until the second *lalui* is encountered. The use of *$2* in the first *laori* instruction is replaced with *$1*. Finally, the second *lalui* instruction is replaced by a *move* instruction, where the source operand is *$1*. Figure 5(e) shows the assembly code after the assembly optimizer removes useless instructions. The *move* instruction that was produced in Figure 5(d) has no effect since the source operand is the same as the destination operand. Figure 5(f) shows the assembly code after common subexpression is applied again. The destination register of the first *laori* is changed to be *$3* so that the address of *gcnt* is not overwritten until the second *laori* is encountered. The use of *$2* in the first *lw* instruction is replaced with *$3*. Finally, the second *laori* instruction is replaced by a *move* instruction, where the source operand is *$3*. Figure 5(g) shows the assembly code after the assembly optimizer applies copy propagation. The *$1* register that is used in the source operand of the *sw* instruction is replaced with *$3*. Figure 5(h) shows the assembly code after the assembly optimizer applies dead assignment elimination. The *move* instruction in Figure 5(g) is now a dead assignment and is removed since there are no uses of *$1*.

```
lw      $2,gcnt
addiu   $2,$2,1
sw      $2,gcnt
```
(a) original code

```
lalui   $2,gcnt
laori   $2,$2,gcnt
lw      $2,($2)
addiu   $2,$2,1
sw      $2,gcnt
```
(b) after pseudo expansion

```
lalui   $2,gcnt
laori   $2,$2,gcnt
lw      $2,($2)
addiu   $2,$2,1
lalui   $1,gcnt
laori   $1,$1,gcnt
sw      $2,($1)
```
(c) after pseudo expansion

```
lalui   $1,gcnt
laori   $2,$1,gcnt
lw      $2,($2)
addiu   $2,$2,1
move    $1,$1
laori   $1,$1,gcnt
sw      $2,($1)
```
(d) after common subexpression elimination

```
lalui   $1,gcnt
laori   $2,$1,gcnt
lw      $2,($2)
addiu   $2,$2,1
laori   $1,$1,gcnt
sw      $2,($1)
```
(e) after removing useless insts

```
lalui   $1,gcnt
laori   $3,$1,gcnt
lw      $2,($3)
addiu   $2,$2,1
laori   $1,$1,gcnt
sw      $2,($1)
```
(f) after common subexpression elimination

```
lalui   $1,gcnt
laori   $3,$1,gcnt
lw      $2,($3)
addiu   $2,$2,1
move    $1,$3
sw      $2,($3)
```
(g) after copy propagation

```
lalui   $1,gcnt
laori   $3,$1,gcnt
lw      $2,($3)
addiu   $2,$2,1
sw      $2,($3)
```
(h) after dead assignment elimination

Figure 5. Example of Assembly Optimizer Transformations

5.2 Error Isolator Technique

When the simulation of *asopt*-produced code fails, we must isolate which code-improving transformation causes the error. To accomplish this task, we developed a tool called *asoptiso* to perform assembly optimizer error isolation. The *asoptiso* tool finds the **first** code-improving transformation causing the simulator to produce incorrect output for a given program. *asoptiso* was implemented as a C program that includes *system()* calls that allow it to perform unix shell commands, which include invoking *asopt* and perform *make* commands to invoke the assembler, the linker, and the simulator. A *transformation range* is a sequence of transformations applied by *asopt* to either a program or assembly file. A *transformation count* is a number assigned to a specific transformation in order to identify it. *asoptiso* performs a **binary**

search on all applied code-improving transformations, comparing the output of the simulation to the reference output file to check for correctness of various transformation ranges until the range is narrowed down to a single transformation.

asopt and *asoptiso* have different responsibilities relating to the transformations applied to a program. The assembly optimizer, *asopt*, applies transformations to a single assembly file within a program. *asopt* has the ability to stop applying code-improving transformations after a specified maximum limit is reached within a file, after which only required transformations will continue to be applied by the assembly optimizer. The assembly optimizer error isolator, *asoptiso*, tracks and controls the number of transformations applied to all assembly files within a program.

```
int moreopts = TRUE;

optimize() {
   ...
   setjmp(...);
   if (moreopts) {
      // apply opts
      ...
   }
   // apply remaining
   // required trans
   ...
}
```

Figure 6. *optimize()* Function

A portion of *asopt*'s *optimize()* function is represented in Figure 6, and is responsible for applying optimization phases. In *optimize()*, each optimization is applied by a call to other functions within *asopt*. Figure 6 shows that a global boolean variable called *moreopts* is set to true by default and indicates whether or not *asopt* should continue to apply code-improving transformations to the current function. When a maximum limit is reached and code-improving transformations will no longer be applied to the assembly file, *moreopts* will be set to false and the control will be returned to the *setjmp()* within *optimize()*. *asopt* will proceed with applying only required transformations to the current function. Once a maximum limit has been reached, code-improving transformations will not be applied for the remainder of the current assembly file.

```
void incropt(enum opttype opt)
   // if limit reached
   if (...) {
      moreopts = FALSE;
      longjmp(...);
   }
   totopts[opt].count++;
   totopts[ALL_OPTS].count++;
}
```

Figure 7. *incropt()* Function

We have to identify the start of a code-improving transformation and then choose whether or not to perform it. We implemented a function named *incropt()* in *asopt* that checks if a transformation should be performed. Figure 7 shows the code for the *incropt()* function, which is called at each point *asopt* determines that a transformation can be applied. The *incropt()* function takes in the argument *opt*, which represents the transformation type to be applied. If the *opt* transformation will exceed a maximum limit, then we set *moreopts* to false and use a *longjmp()* to return to the associated *setjmp()* in *optimize()* in order to stop the further application of code-improving transformations. If this transformation will not exceed a maximum limit, we increment the number of transformations. This combination of calling *setjmp()* and *longjmp()* allows us to stop applying code-improving transformations at any point when isolating *asopt* errors.

The *isolation flag* is the *asopt* command line flag, where the first erroneous transformation is suspected to be of this type. The *non-isolation flags* are an optional string of command line flags used in conjunction with the isolation flag. Note, only the isolation flag transformations will be isolated, although the non-isolation flags will also be used to apply optimizations when they are specified. The isolation flag may be set to a particular flag, that is reserved to represent all applied transformations if a first erroneous transformation type is not suspected. *asoptiso* performs a binary search on all code-improving transformations of a program associated with the specified isolation flag.

The three main steps of *asoptiso* are shown in Figure 8, which are represented by the *a*, *b*, and *c* sections, respectively. Before step *a* begins, *asoptiso* reads in a configuration file which specifies error isolation information, including which types of transformations should be applied.

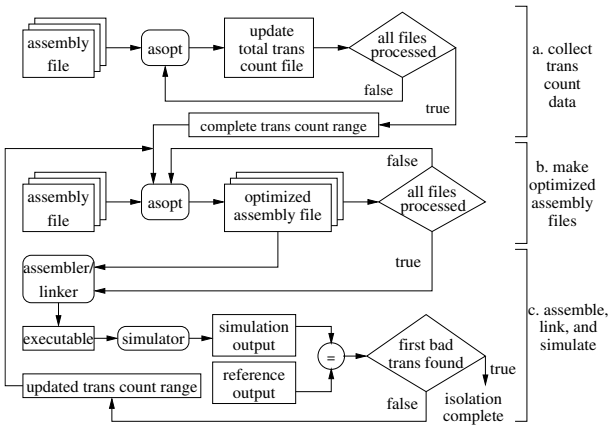


Figure 8. Optimization Error Isolation Process

In step *a* of Figure 8 *asoptiso* runs *asopt* on each assembly file in the program to apply all transformations of the types specified by the flags. *asopt* adds the number of transformations for the current file to the *total_trans_count* file.

Step *b* of Figure 8 produces the optimized assembly of the program to be simulated. When transitioning from step *a* to step *b*, the binary search range is initialized to be the range of all applied code-improving transformations of the isolation flag type within the program. If a suspected transformation type is not known, the isolation flag may be set so that the binary search range will be all code-improving transformations applied in the program. A binary search range *midpoint* is set to be the middlemost transformation within the binary search range. The midpoint is the transformation count to

which transformations will be applied inclusive. When transitioning from step *c* to step *b*, the updated binary search range is chosen based on the most recent simulation result.

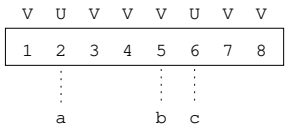
As *asoptiso* is processing each assembly file as shown in step *b* of Figure 8, *asoptiso* will first check if the file being processed is the *midpoint assembly file*, which is the file in which the midpoint transformation is applied. The isolator will inform *asopt* to apply transformations without restriction to all assembly files prior to the midpoint assembly file. Once at the midpoint assembly file, the isolator will have *asopt* apply all transformations up to, and including, the midpoint transformation. The isolator will apply no code-improving transformations to files after the midpoint assembly file. In order to further decrease the error isolation time, *asoptiso* does not have *asopt* process an assembly file when the same number of code-improving transformations would be applied to the file for the next simulation.

Step *c* in Figure 8 assembles, links, and simulates the program optimized by step *b*. The optimized assembly files are assembled and linked to produce an executable that is input to the simulator. The simulation output is compared against the program's reference output file to determine if the simulation was successful. A new binary search range is determined based on whether simulation output was correct or incorrect and the isolator returns to step *b* if the new range is larger than one transformation. The isolation is complete when the binary search range has been narrowed down to one transformation, which is the first erroneous transformation.

5.3 Error Isolator Enhancement and Benefits

asoptiso contributes an enhancement over previous work through speeding up the isolation process by decreasing the number of simulations performed. *asoptiso* takes in an isolation flag, which is useful since compiler writers often implement one new optimization at a time. If the flag combination AB has been successfully simulated, but ABC has not, we can assume the problem likely lies in the newest flag added. Knowing which transformation type on which to focus the error isolation provides a significant advantage: it decreases the initial range of transformations which may contain an error, resulting in fewer steps in the binary search and therefore minimizing the number of simulations required to isolate the first erroneous code-improving transformation.

diff minimization is performed to further enhance the assembly optimizer error isolation process. Figure 9 shows eight *asopt* transformations applied. Note that there are three transformations of interest, labeled as *a*, *b*, and *c*, which are transformations 2, 5, and 6, respectively. Let's say that isolation is performed on a program with an isolation flag of U and non-isolation flag of V, so only U type transformations are being isolated. Assume the result of *asoptiso* reveals that Figure 9's transformation *c* is the problem, which means that the U transformations up to, and including, transformation *a* are not causing incorrect output. To perform diff minimization,



a: last correct U transformation

b: transformation before first erroneous U transformation

c: first erroneous U transformation

Figure 9. Error Isolation Diff Minimization

the isolator will simulate again, applying transformations up to transformation *b* inclusive. If this simulation is correct, *asoptiso* decisively determines that transformation *c* is the first erroneous transformation among all types. Otherwise, the first erroneous transformation is located after transformation *a* but before transformation *c*, and a second binary search is now performed in this new search range.

Diff minimization also serves to increase the precision of the isolator results. Diff minimization ensures that there is only a difference of one transformation, of any type, between correct code and the first found erroneous transformation. Consider Figure 5 again. It is much easier to view the differences between the assembly code before and after a single transformation than multiple transformations.

Applying a binary search using an isolation flag could be the difference between isolating on 100,000 transformations of all types versus 100 transformations of a single type. Decreasing the number of binary search steps also decreases the numbers of simulations. When the programs being tested may take hours to simulate, even a small decrease in the number of simulations is highly beneficial.

Once the isolation is complete, the user receives three output files: the assembly file applying the first erroneous transformation, this same assembly file without the first erroneous transformation, and a simulation result data file. The first two files may be used with a diff tool to quickly spot errors since they will only differ by the one erroneous transformation. The third file includes the file-offset transformation count of the first erroneous transformation. A conditional breakpoint can be placed at the *incropt()* call to stop immediately prior to the application of the first erroneous transformation and the compiler writer can determine why the transformation was erroneously applied.

6 *asopt* Isolation Error Results

In this section we describe the number of code-improving assembly optimizer transformations on several known benchmarks and estimate the number of simulations required to isolate the first transformation that causes incorrect output. We do not show results from isolating simulator errors as there is very little overhead as compared to conventional simulation since the functional simulator runs in parallel with the cycle accurate simulator.

A large number of transformations can be applied by the assembly optimizer for a given benchmark, which indicates a need to automatically isolate the first transformation that causes incorrect output during simulation. Table 2 and Table 3 show the number of transformations applied for each

Table 2. Number of *asopt* Transformations in SPEC CINT2006 Benchmarks

Type	bzip2	gcc	gobmk	h264ref	hmmer	lib quantum	mcf	perl bench	sjeng
DAE	330	5,239	1,683	2,949	1,074	91	60	2,733	514
CSE	521	6,257	1,948	2,241	704	77	67	5,045	768
LICM	360	2,434	902	2,284	877	69	27	739	224
CP	441	5,674	1,586	3,288	1,191	109	60	3,114	562
RCO	0	2	0	0	0	0	0	0	0
AE	1	12	1	22	3	2	0	3	0
SBIV	38	437	231	394	113	11	9	140	51
EPI	6,100	315,210	74,732	55,362	28,003	3,127	1,117	114,954	12,774
VBS	4,385	339,674	58,717	34,631	21,871	2,587	949	111,908	11,502
VGS	1,790	142,854	28,571	20,814	9,141	1,019	398	51,452	7,010
VFS	1,440	99,771	17,199	10,172	6,545	652	300	29,790	3,607
LU	39	499	331	338	120	14	13	232	56
MI	4	79	6	69	7	1	3	46	1
RITD	385	23,481	11,537	3,041	3,029	630	100	7,505	644
REB	413	25,662	4,007	2,043	1,677	165	110	6,991	617
RUI	373	3,281	954	1,067	393	61	33	2,519	485
MBB	79	1,101	790	681	253	28	26	454	124
RAE	15	589	119	83	49	4	0	239	48
SRNR	9	324	78	84	82	13	3	84	16
total	16,723	972,580	203,392	139,563	75,132	8,660	3,275	337,948	38,973

Table 3. Number of *asopt* Transformations in SPEC FP Benchmarks

Type	cactus ADM	gems FDTD	lbm	sphinx3	zeus mp
DAE	1,777	1,724	85	433	1,240
CSE	1,938	2,169	67	255	2,252
LICM	1,312	1,201	27	359	291
CP	1,915	1,968	74	463	1,353
RCO	10	1	0	0	0
AE	19	1	0	4	8
SBIV	140	82	2	39	41
EPI	67,581	42,810	1,318	15,984	34,898
VBS	41,373	20,626	322	11,262	10,448
VGS	19,826	9,646	249	5,277	8,481
VFS	10,761	5,381	64	2,646	4,053
LU	152	59	4	54	34
MI	28	16	0	9	8
RITD	5,958	719	109	1,854	610
REB	3,256	1,710	19	817	264
RUI	963	451	3	157	842
MBB	303	139	6	104	70
RAE	62	173	3	24	19
SRNR	80	17	4	38	2
total	157,454	88,893	2,356	39,779	64,914

transformation type for each of the SPEC 2006 integer and FP benchmarks, respectively. The optimization phase type abbreviations and a description of each type of optimization phase referenced in Tables 2 and 3 are given in Table 1. One can see there are a number of transformations applied within each of these benchmarks. The number of transformations applied by *asopt* for the *gcc* benchmark is almost 1 million even when not including the transformations for the libraries. It is obvious that some phases apply more transformations than others. Some SPEC 2006 benchmarks are not included as we are currently having some problems resolving linking issues with the C++ benchmarks and are still resolving errors with some of the Fortran FP benchmarks. We generated code and tested all the applications listed in Tables 2 and 3. The assembly optimizer error isolation technique is even more valuable when isolating an error associated with the library code due to the large number of library files.

As described in Section 6, *asoptiso* can decrease the number of simulations if it isolates on a single type of code-improving transformation rather than all code-improving transformations being performed. Table 4 shows the number of simulations when varying the type of search and the number of transformations on which the search is performed. We show in the table the number of all code-improving transformations and the number of loop unrolling (LU) transformations. We also show the number of simulations when performing a sequential search on all transformations or a binary search on all transformations or just all LU transformations. The number of simulations for a sequential search would be on average $n/2$, where n is the number of all code-improving transformations. This naive error isolation approach will be very inefficient. The number of simulations required when isolating errors on all transformations using our error isolation binary search will be $\lceil log_2(n) \rceil + 1$. The +1 accounts for one simulation that occurs before step *a* of Figure 8 in order to check that the program correctly simulates when no code-improving transformations are applied. The number of simulations when isolating errors using our error isolation binary search on only a single transformation type will be $\lceil log_2(k) \rceil + 2$, where k is the number of code-improving transformations of that type. An additional simulation is required to verify that the isolated transformation of that type is really causing the problem. One can see that the number of simulations is decreased when isolating on a single type of code-improving transformation as opposed to isolating on all code-improving transformations. The decrease in the number of simulations will depend on the fraction of total transformations applied of the isolating transformation type.

Table 4. Assembly Optimizer Error Isolation Results

Benchmark	Number of Trans		Number of Simulations		
	All Code Improving	LU	Sequential Search on All Trans	Binary Search on All Trans	Binary Search on LU Trans
bzip2	16,723	39	8,362	16	8
gcc	972,580	499	486,290	21	11
gobmk	203,392	331	101,696	19	11
h264ref	139,563	338	69,782	19	11
hmmer	75,132	120	37,566	18	9
libquantum	8,660	14	4,330	15	6
mcf	3,275	13	1,638	13	6
perlbench	337,948	232	168,974	20	10
sjeng	38,973	56	19,487	17	8
cactusADM	157,424	152	78,712	19	10
gemsFDTD	88,893	59	44,447	18	8
lbm	2,356	4	1,178	13	4
sphinx3	39,779	54	19,890	17	8
zeusmp	64,914	34	32,457	17	8

7 Examples of Isolating *asopt* Errors

In this section we describe a couple of interesting examples of errors that *asoptiso* isolated. (1) The first example error was caused by loop unrolling. The original loop was accessing contiguous elements of an array and the loop exit condition

branch checked if the basic induction register value was *not equal* to a particular address. After unrolling *asopt* changed the loop exit condition to use a branch with a *less than* condition since it is possible that the register value could skip over the exit address value. However, the exit address had its most significant bit set and the *less than* condition treated this address as a negative value. We had to change the loop unrolling optimization to check if the basic induction register was ever used to dereference memory and to change the exit branch to instead use an *unsigned less than* condition. (2) The second example error was caused by a code improving transformation that used a new callee-save register. This transformation was then followed by a required transformation that saved and restored the new callee-save register in the function's activation record, which also required increasing the activation record size. This function received some extra arguments passed on the run-time stack, which was accessed in the caller's activation record. Because the adjustment to the stack pointer was increased to save and restore the new callee-save register, the offset from the stack pointer to access the arguments passed in the caller's activation record had to also be increased.

8 Related Work

There has been a significant amount of work to assist in the testing of architecture simulators. Much of this work revolves around determining correctness and detecting if simulators accurately reflect those of physical processors. Glam and Lilja lay out a technique for ensuring that instructions in a simulated processor are faithfully executed [9]. Our technique differs as we assume there is no physical processor to act as a source of truth, instead using a simplified functional simulator.

Beardo et al. verified the functionality of a VLIW architecture by using the memory unit to observe the state of the processor[4]. Similar to our approach, they impose restrictions on VLIW code scheduling such that the code can be represented as a single stream of instructions. However, they do not use this single stream to compare instructions through a source of truth mechanism such as a verified simulator.

There has been a significant amount of work to assist in the testing of compilers. This work includes complex methods such as compiler verification and translation validation [3], in addition to more practical methods such as constructing test programs, determining whether the output of a compiler is correct or not, optimizing the testing process, and post-processing of test results [7]. Isolating the first code-improving transformation that causes incorrect output, and then identifying the place in the compiler source code where that transformation is applied, falls in the last category.

A tool known as *bugfind* was developed to assist in the debugging of optimizing compilers [6]. The *bugfind* tool attempts to determine the highest optimization level at which each file within a program can be compiled and produce correct output. To isolate a function that was not correctly optimized, one has to place each function within the application in a separate file. This tool also relies on a different compilation of each function that produces correct code. The *bugfind* tool uses the *make* facility in Unix and is generalized enough to work with different compilers.

LLVM provides a tool called *bugpoint*, which is used to isolate problems to LLVM optimization phases (passes) [1]. It appears that *bugpoint* can isolate the phase that is causing incorrect output. In contrast, *asoptiso* isolates the first code-improving transformation that causes incorrect output.

The *vpoiso* tool finds not only the failing module, but also the first code-improving transformation within a function that causes incorrect results [5, 11]. The transformation number can be used to access the point in the *vpo* compiler when the transformation is about to be applied.

The *asoptiso* tool is most similar to the *vpoiso* tool in that both tools isolate the first code-improving transformation that causes incorrect output. *asoptiso* also allows for isolation of only a specified type of code-improving transformation to decrease the error isolation time, which is important when isolating errors using long-running simulations. We have shown that *asoptiso* can significantly decrease the required number of simulations when a specific compiler optimization is being tested that is likely the cause of the error.

9 Conclusions

In this paper we described the tools we developed to facilitate bootstrapping a simulation and low-level compilation infrastructure for a new ISA. The approach we use for initial bootstrapping allows a simulator to be used for testing a new ISA before any compilation support is provided. The tool for isolating simulator errors determines the first instruction in any benchmark that produces an unexpected value. The tool for isolating assembly optimization errors isolates the first code-improving transformation that causes incorrect results from the simulator. All of these tools are completely automatic and have proved to be invaluable when we were isolating errors as we retargeted our simulation and low-level compilation infrastructure to a new ISA.

Acknowledgments

We appreciate the suggestions by the reviewers that helped to improve the quality of this paper. This work was supported in part by the US National Science Foundation (NSF) under grants DGE-1565215, CRI-1822737, CCF-1823398, CCF-1823417, CCF-1900788, CCF-1901005, DUE-2030070, OISE-2103103, OISE-2103105, DGE-2146354, CCF-2211353, and CCF-2211354. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and may not reflect the views of the NSF.

References

[1] [n. d.]. *LLVM bugpoint tool: design and usage.* https://llvm.org/docs/Bugpoint.html

[2] 2019-2023. Statically Controlled Asynchronous Lane Execution (SCALE) Project. In *Division of Computing and Communication Foundations (CCF)*. National Science Foundation Grants CCF-1900788 and CCF-1901005.

[3] A.Pnueli, M. Siegel, and F. Singerman. 1998. Translation Validation. In *Proceedings of TACAS '98*. 151–166 pages.

[4] M Beardo, Francesco Bruschi, Fabrizio Ferrandi, and Donatella Sciuto. 2000. An approach to functional testing of VLIW architectures. In *Proceedings IEEE International High-Level Design Validation and Test Workshop (Cat. No. PR00786)*. IEEE, 29–33.

[5] M. R. Boyd and D. B. Whalley. 1993. Isolation and Analysis of Optimization Errors. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*. 26–35.

[6] J. Caron and P. Darnell. 1990. Bugfind: A Tool for Debugging Optimizing Compilers. *SIGPLAN Notices* 25, 1 (Jan. 1990), 17–22.

[7] J. Chen, J. Patra, M. Pradel, Y. Xiong, H. Zhang, D. Hao, and L. Zhang. 2020. A Survey of Compiler Testing. *Comput. Surveys* 53, 1 (May 2020), 1–36.

[8] J. A. Fisher. 1983. Very Long Instruction Word architectures and the ELI-512. In *Proc. Int. Symp. on Computer Architecture*. ACM, New York, NY, USA, 140–150.

[9] B. Glamm and D.J. Lilja. 2001. Automatic verification of instruction set simulation using synchronized state comparison. In *Proceedings. 34th Annual Simulation Symposium*. 72–77. https://doi.org/10.1109/SIMSYM.2001.922117

[10] Soner Önder and Rajiv Gupta. 1998. Automatic generation of microarchitecture simulators. In *IEEE International Conference on Computer Languages*. Chicago, 80–89.

[11] D. B. Whalley. 1994. Automatic Isolation of Compiler Errors. *ACM Transactions on Programming Languages and Systems* 16, 5 (September 1994), 1648–1659.

Received 2023-03-16; accepted 2023-04-21