Dynamic Translator: Firmware-Scheduled VLIW Processor

Saurabh Gayen  Brandon Heller
saurabh.gayen@gmail.com brandon.heller@gmail.com

CSE 560: Computer Architecture
Washington University

Abstract
Our group has created a dynamically translating VLIW processor that uses firmware for instruction scheduling. The processor executes MIPS instructions by dynamically translating them into VLIW, and then executing the translated code. We have also implemented a combined toolchain that compiles C to binary for our VHDL processor. The motivation behind the project was to gain practice writing an assembler and processor, and to gain insight into the benefits of and issues with dynamic translation.

1. Introduction
CSE560 is a research-oriented graduate-level computer architecture course at Washington University. In its first five weeks, students design and simulate a five-stage pipelined RISC processor. The rest of the semester is devoted to a freely chosen final project. We chose this project as a way to learn more about processor design and dynamic translation. Dynamic translation VLIW processors require software-hardware codesign, as the hardware design affects the implementation of the firmware translator component, and vice versa. Additionally, dynamic translation is relevant to current research, being a topic that was virtually unknown in 2000, and possesses a number of advantages for simpler, faster, more flexible processor designs.

Modern superscalar, out-of-order execution processors are extremely complex and power-hungry. As Moore’s law continues to hold, transistor budgets increase, and many of those increases have gone into rename buffers, dispatch hardware, and out-of-order retire units. In this paper, we describe the operation of an in-order VLIW processor in which software replaces some of the hardware functionality, thus reducing the amount of hardware required. Usually VLIW processors require code to be recompiled for different architectures. Using software fixes this problem by using a dynamic translation approach.

Our focus was the creation of a simple infrastructure for dynamic translation, and an understanding of the modifications needed to evolve a simple five-stage pipelined to support dynamic translation. The translator implementation is extremely simple and extremely slow, but it does show that the architecture is capable of performing dynamic translation. Each section of this paper will be written in the context of creating a complete research platform for codesigned dynamic translation research.

2. Dynamic Translation
Dynamic translators (DTs) are software, hardware, or software/hardware combinations in which a guest architecture code stream is translated into the host architecture before execution. The translate phase must happen before the execute phase can occur, and can happen at instruction or block-level granularity. A generic model of DT is shown in Fig. 1.

![Dynamic Translation Model](image)

The hierarchy of dynamic translation includes software, hardware, and codesigned dynamic translation systems. Pure-software dynamic translators include four types: high-level-language (HLL) virtual machines, virtual machine monitors, dynamic binary translators, and same-ISA dynamic binary optimizers.

HLL virtual machines include the runtime components of all high-level languages that compile into an intermediate program representation. The most visible recent examples include Java Virtual Machines and the .NET runtime, although Perl, Python, and Matlab also count as HLL VMs.

Virtual Machine Monitors (VMMs) simulate all aspects of a guest system. The goal of a VMM is to provide virtualization, the ability for multiple pieces of software to each have an exclusive view of a single, shared physical system. One modern example is VMWare, which enables multiple different operating systems such as Windows and Linux to run on one physical x86 system.

Dynamic binary translators (DBTs) allow code written for one OS or platform to be executed by another. WINE, FX!32, and Rosetta are examples of these. They are used to enable smoother ISA transitions, or as a selling tool by enabling competitors’ programs to transparently run on the host architecture. Digital made a number of DBTs in the late 90’s...
to run MIPS, x86, and VAX binaries on its new Alpha architecture, without requiring them to be ported.

Same-ISA dynamic binary translators are a new category of DT with the aim of improving speed or increasing security. An example of this is HP’s Dynamo [2], which works by interpreting basic blocks. When a basic block has been executed enough times, Dynamo switches to a trace creation mode in which a new linear sequence of previously branching code is created. The translation is then optimized and scheduled. Code is realigned using branch predictor data to optimize the fast, expected code path. Dynamo’s speedup derives from the increased performance of the fetch unit from longer linear segments of code. The ADORE [8] framework is also a same-ISA dynamic binary optimizer, but it works by profiling memory references and inserting prefetch instructions where they would provide a benefit. Yet another application in the same class is DynamoRIO [9], which aims to enable secure, sandboxed code execution through dynamic translation.

Pure-hardware DTs include most recent superscalar x86 processors, like the Athlon and Pentium series, that convert instructions into a simpler RISC-type format before execution. In general, there is a one-to-n mapping between x86 instructions and micro-ops. The trace cache in the Pentium 4 saves recent instruction translations to remove the decode stage from the critical path and create more linear code.

The last category of dynamic translation system is the codesigned dynamic translator (CDT), which has hardware features to accelerate DT, but requires a software component. This paper will detail one such implementation. Recent examples include the Transmeta Crusoe and Efficeon series, which use DT to reduce die size, power, and overall complexity.

Codesigned DTs provide design flexibility between hardware and software. If a feature is too heavy on hardware resources, it can be moved to software. If a feature is too slow in software, it can be accelerated in hardware. If a mistake is made during chip fabrication, it can be fixed in software afterwards at no cost. CDTs with VLIW internal ISAs also provide power efficiency advantages, in that there is no need for complex decode units, issue units, or out-of-order retire units. Reducing hardware reduces power consumption. DT can provide better code than that produced by a compiler by continually adapting to a new profile, and performing optimizations that only manifest at runtime. One other advantage may be the most important one for the future: CDT’s provide ISA independence. There is no need to recompile all programs for a new architecture, or create large binary files to cover different implementations of an ISA. To a CDT, there is no difference between a real ISA and a virtual one, and built-in profiling and optimization mechanisms can accelerate virtual ISA execution. The vendor controls the key to their processor’s performance, and has no need to create new compilers for each internal architecture revision.

3. Software

The goal of the software components is full transparency. A programmer running code on our processor should require no knowledge of its internal details, and should only have to conform to the external ISA. In our case, the external ISA is MIPS, while the internal ISA is a custom VLIW ISA superset of MIPS. A diagram of the system structure is shown in Figure 2.

3.1 Software Toolchain

The software toolchain takes C code as its input, and produces binary as its output. The binary can be in either MIPS or VLIW format. MIPS format is supported to generate test MIPS binaries, while VLIW format is supported because the dynamic translation firmware needs to be in the native processor format for direct execution. By providing the toolchain, all software for the project can be written in high level C. This feature allows more sophisticated test programs and firmware to be created in the future.

The first part of the software toolchain, shown in Figure 3, is a GCC MIPS cross-compiler. The cross-compiler is called with the –s and –O3 options on C Code to produce a MIPS assembly file. This assembly file is used as the input to a back-end assembler written in Perl. The back-end assembler handles all the assembly pre-directives and the full MIPS3 ISA assembly instructions generated by the cross-compiler, to generate a binary that conforms to the processor’s supported MIPS ISA.

The Perl assembler can be run in two modes, MIPS binary mode and VLIW binary mode. MIPS binary mode is used to produce test programs that are supported by the hardware’s external ISA interface. When running the platform, the MIPS binary test program is loaded into the data memory.

In VLIW binary mode, the software toolchain outputs VLIW binaries. The toolchain is used in this mode to generate the firmware that is to be loaded into ROM. The assembler currently generates sparse VLIW (one valid instruction per
bundle) to be run on the native machine, but this can be improved in future versions.

3.2 Firmware

The VLIW bundle format is two instructions long. The first slot is for ALU or branch instructions and the second is for memory instructions, as shown in Figure 4.

<table>
<thead>
<tr>
<th>ALU/BR</th>
<th>MEM</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4: Bundle Format

The bundle format allows an ALU and a memory instruction to be issued in parallel. However, if a branch instruction is in the first slot, a memory instruction is not allowed in the second slot. This bundle restriction prevents a taken branch from causing a memory instruction to execute.

ALU/BR MEM

Figure 4: Bundle Format

The firmware, shown in Figure 5, follows a simple loop of translating a block of code and then executing that block of code. In translate mode, the firmware reads a basic block of MIPS code from the data memory and writes VLIW code into the translation memory. The end of a basic block is detected when a branch or jump instruction is read. After the translation has been created, a jump instruction to the start of that translation is executed. The translated basic block is in VLIW, so it runs as native code until another branch is encountered. At that point, execution jumps back to the translator, which starts its new translation from the target of the about-to-be-taken branch.

The following code snippet, Figure 6, shows an example of MIPS code that might be available in the data memory. This is a basic block because the last valid instruction is a branch type instruction.

```
addi $13,$13,1
sw $10,16($0)
addi $14,$14,1
sw $11,20($0)
sw $12,24($0)
sw $15,28($0)
nop
jr $5
nop
```

Figure 6: MIPS code

The firmware currently places a MIPS instruction in the correct bundle slot, paired with a nop in the other slot. An example of this sparse VLIW code is shown in Figure 7. Although we only show sparse VLIW code here, the processor is entirely capable of running both ALU and memory instructions concurrently.

```
addi $13,$13,1
nop
addi $14,$14,1
nop
sw $11,20($0)
nop
sw $12,24($0)
nop
sw $15,28($0)
nop
jr $5
nop
```

Figure 7: VLIW code

3.3 Optimizations

The six weeks in which to finish the bulk of the project prevented us from completing a more intelligent translator. We have, however, identified a number of opportunities for optimizing translated code. The bulk of these strategies require extra time, limiting their use to frequently-executed basic blocks for which the optimization overhead is recovered by future executions of those blocks. Optimizations performed by dynamic translators generally fall into two categories: static optimizations, those that can be performed by a traditional compiler, and dynamic optimizations, those that require runtime information.

3.3.1 Static Optimizations

The first optimization we would like to implement is instruction scheduling, in which both instructions are moved to better fill the bundles. Jump and nop removal are a type of optimization that we could add with minimal effort. Dead code elimination and common sub-expression elimination are more expensive techniques. All the techniques mentioned so far would operate at the basic block level, severely limiting their usefulness. An offline compiler has additional semantic information about the code, such as function boundaries, to perform optimizations along much larger blocks of code. Dynamic translators begin with no knowledge of the structure...
of the code; they learn that information by storing blocks as the program executes. For most static optimizations to be worthwhile, they would need to be combined with a dynamic algorithm for generating larger groups of code to optimize.

3.3.2 Dynamic Optimizations
Dynamic optimizations are those opportunities for speedup that only manifest at run time; they require knowledge that only appears as a program runs. The most common dynamic optimization is trace creation, in which branch history information is used to reorder code placement such that frequently-executed blocks of code are contiguous in memory. A number of algorithms exist for creating traces that are highly likely to execute to completion. The block alignment improves the effective fetch width, as instructions are much more likely to be executed when fetched as part of a line.

Profiling information is required to create useful traces. The simplest technique is to add a counter to each basic block, although branch direction information can be stored and correlated to form better traces. Profiling enables the dynamic translator to identify regions of code that should be more heavily optimized, especially for expensive compiler-type static optimizations. This process of profiling and optimizing is also seen in commercial Just-In-Time HLL VMs.

4. Hardware
The design of our processor occurred in stages. First, we modified a five-stage pipelined processor into a MIPS VLIW. Second, we made modifications to enable the processor read and write to from multiple memories, including a data memory, translation memory, and firmware ROM.

4.1 VLIW MIPS Conversion
To enable better compiler support, the first processor modification was a conversion to the MIPS ISA from a reduced version of the DLX ISA. The greater instruction encoding complexity of the MIPS ISA required changes to virtually every VHDL module in the processor. VLIW operation requires the ability for a processor to fetch multiple instruction words at the same time, so the imem was increased to two instructions in width. To support the multiple functional units of a VLIW, two read ports and one write port were added to the register file. Previously, the ALU was used as an adder for dmem address generation. This functionality was replicated to enable the ALU and memory to execute concurrently. A hardware diagram is shown in Figure 8.

4.2 Firmware Support
The memory mapping, shown in Figure 9, is transparent to both operating systems and applications. The external view, seen by the programmer, consists of a single memory for code and data. The internal view, seen by the hardware, consists of a firmware ROM, translation memory, and data memory with memory-mapped registers.

To enable DT, we augmented the fetch unit to pull instructions from two memories. The data memory write unit was changed to handle writes to both the data memory and translation memory. Since the translation memory is only written to when the DT is fetching its instructions from the firmware ROM, no resource dependency stalls are introduced. The branch unit automatically handles the transition between translate and execute modes. It compares the current PC location to the most recent branch location and decides whether to jump to the translator or treat the branch as usual. Much of the DT complexity is found in the branch unit. An additional signal was added to enable automatic writes of the next jump location to a fixed location in memory when execution jumps to the translator. Figure 10 shows a diagram of our DT hardware.
5. Conclusion

We have implemented a complete system to do dynamic translation, comprising hardware for DT, firmware for DT, and an external interface. Completing our project has led to an understanding of dynamic translation’s potential and pitfalls. We have come to the realization that the complexity of debugging a program running on a VHDL model is excessive. The next logical step is to implement more advanced firmware, but a simulator would be a much more efficient platform on which to work. We have also identified other areas ripe for future research.

6. Future Work

Research should be done to better understand the characteristics of code in the context of dynamic translation. Key to the performance of any DT system is the distribution of basic blocks executed dynamically. How large is each block, how often is it executed, and what is the temporal locality of a group of blocks? How does the type of benchmark affect such numbers? Such knowledge could lead to more efficient cache designs for dynamic translation, in addition to models of the costs of dynamic translation and its potential for speedup.

The process of creating this system revealed a number of shortcomings in the tools available to researchers wanting to study DT. We need an environment in which we can more easily codesign. The interplay of hardware and software is difficult to see. We had no easy way to add instructions to the compiler, and would love to have the ability to write the interpreter primarily in an HLL, with inline assembly for speed-critical regions. We need a simulation framework for benchmarking and performance modeling. We can’t synthesize a processor and see the real effects of new instructions or hardware structures on run frequency.

The closest work to what we seek is the DAISY project from IBM. Unfortunately, DAISY works with a simpler ISA than x86 that reduces DT’s potential for power efficiency gains. It also includes no synthesizable hardware component. Transmeta’s Crusoe is similar to our approach, but is a fully closed system.

The existing platforms for CDT research are insufficient to truly understand its performance potential. There is no fully open-sourced DT platform with a compiler, translator, and synthesizable hardware model. We find this surprising, considering that DT often leads to smaller, simpler pipelines that work at high clock rates, in addition to efficiency gains. We assert that Washington University should build such a comprehensive platform for DT research.

6. Acknowledgments

We would like to thank Professor Young Cho for supporting our choice for a final project, and turning CSE560 into a flexible, research-oriented course.

7. References

[7] Ars Technica: HP’S Dynamo http://arstechnica.com/reviews/1q00/dynamo/dynamo-1.html


With modifications to make the paper more modern, such as those in http://http://delivery.acm.org/10.1145/350000/349303/p1-bala.pdf?key1=349303&key2=0681954311&coll=GUIDE&dl=ACM&CFID=60117518&CFTOKEN=85289365