Automatic Generation of Interrupt-Aware Hardware Accelerators with the M2V Compiler

Abilash Sekar
Georgia Institute of Technology
abilash.sekar@gatech.edu

Alessandro Forin
Microsoft Research
sandrof@microsoft.com

ABSTRACT

The MIPS-to-Verilog (M2V) compiler and the Basic Block Tools (BBTools) can automatically generate a hardware accelerator for selected blocks of machine code in an application. The compiler translates MIPS machine code into a hardware design captured in Verilog (an "Extension"). The BBTools patch the application binary by inserting the extension instruction that triggers the accelerator. The original code is preserved, so that execution can fall back to software when necessary.

This work extends the M2V compiler with support for memory load and store instructions, and for interrupts. We use a transactional model to handle interrupts and/or traps due to TLB misses in the Extension. We also extended the BBTools to automatically create the best encoding for an extension instruction. The tool evaluates which pair of roots in the dependency graph leads to the shortest execution cycle time for the Extension. With this addition, the process of creating Extensions for the eMIPS processor can now be fully automated and applied to practical applications, where loads and stores inside the Extension are of paramount importance. Code coverage is already at 50% of a large code base.

1. INTRODUCTION

Extensible processors have a simple RISC pipeline and the ability to augment the Instruction Set Architecture (ISA) with custom instructions. The ISA can be augmented statically, at tape-out, or it can be augmented dynamically when applications are loaded. Extensible processors differ from other accelerators in their tight integration with the basic data path, which leads to minimal latencies and therefore greater flexibility. Examples of dynamically extensible processors include eMIPS [5] and Stretch [10]. Tensilica's Xtensa [11] is an example of a statically extensible processor.

Selection of the best code to accelerate is an active area of research. The eMIPS tool-chain, shown in Figure 1, selects the best candidate blocks by executing the application on the Giano full-system simulator [17], in concert with the data obtained via static analysis of the application binary. The BBTools extract the basic blocks to accelerate and patch the binary image with the special instructions for the accelerator. The M2V compiler [13] automatically generates the design for the hardware accelerator.

In previous versions of eMIPS, the accelerator blocks could be specified and given to a hardware designer to hand design the accelerator. While this can lead to an efficient implementation, manual designs do not scale well as extensible processors are more widely used and the hardware becomes more complex. The use of the M2V tool chain can expand the use of hardware acceleration and completely automate the process of generating hardware accelerators.

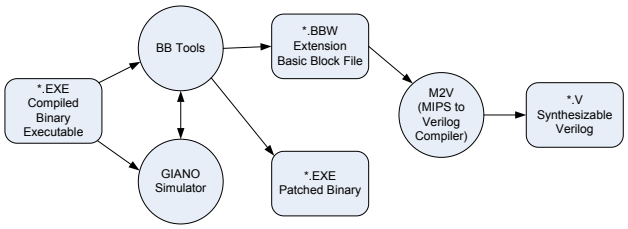


Figure 1. The eMIPS Tool Chain to automate the generation of hardware accelerators.

The work described herein addresses three limitations in the tool chain that prevented the M2V compiler from being usable in but a few practical cases. The first limitation was the lack of support for load and store operations, and more generally for variable-cost operations. M2V only handled MIPS instructions that took zero or one cycle. We added support for all the instructions that took a fixed number of cycles, accounting for the cost in the generation of the schedule. We then added support for the instructions that have variable costs, prime and foremost loads and stores. We developed a way to preserve the overall structure of the compiler, while dealing with the variable costs. The dependency graph still leads to a state machine that controls the overall execution. The transitions are now defined not only by the clock, but also by the signals that indicate completion of the variable-cost operations.

The second limitation was the lack of support for interrupts within an Extension. External interrupts could only happen before or after the extension instruction, never inside it. This assumption is invalid in the presence of TLB misses due to loads and stores. Furthermore, the assumption requires that an Extension never encounters errors, such as arithmetic overflows or unaligned addresses. For a real-time system, it is important to respond to interrupts in a timely and predictable manner. Even in a general-purpose OS it is unacceptable to allow a user process to ignore interrupts and lock the machine. To address the interrupt limitation, we used the concept of transactions in dealing with the write-backs to the register file and the stores to memory. The overall execution of the Extension is subdivided in sets that execute "atomically". Interruptions of any sort are accepted only at the transaction boundaries. On interruption, the write-back machine cancels all write-backs from future transactions, completes the write-backs for the current transaction, and then relinquishes control back to the data-path in a limited amount of time. The restart-address is set to the point in the original basic block that corresponds to the current write-back state. It is therefore mandatory that extension

instructions are simply inserted in the original image, and that they do not replace the original basic block.

The third limitation concerns the instruction encoding for the new extension instruction. We observed that the selection of which registers or constants to encode in the instruction can have an effect on the overall execution time. These values are available early in the execution pipeline. It is therefore important to select those that allow the most work to proceed before stalling on a dependency. Our algorithm uses two parameters in deciding which two register numbers to encode – fan-out and depth of the root register read nodes. The algorithm selects the pair of registers with the maximum combined fan-out and depth.

The remainder of this paper is structured as follows. Section 2 discusses related work. Section 3 gives an overview of the eMIPS hardware platform. Section 4 discusses the automatic encoding of the extension instruction by the BBTools. Section 5 defines the support for memory operations in the M2V compiler. Section 6 discusses the model and implementation for handling interrupts in the Extension. Section 7 discusses the experimental results, and Section 8 concludes the report.

2. RELATED WORK

Commercial FPGA manufacturers today all provide examples of soft-cores, microprocessor designs that the customer can modify and extend for their application [12, 2, 10]. M2V uses the eMIPS processor [5] as its underlying hardware platform. eMIPS is the first design that is secure for general purpose multi-user loads, and the set of potential applications is therefore more open-ended than those found in the typical embedded system alone.

A common approach to generate code for an extensible processor is to modify an existing C compiler. Tensilica [11] automatically regenerates a full GNU compilation system given the RTL of the new instruction. Ienne et al. [3] use the SUIF compiler. M2V accepts as input binary machine code rather than source code. There are trade-offs between accelerating from source code in a high-level language or from binaries. One of the major advantages when accelerating from binaries is that any application can be accelerated, even applications where the source code is controlled by an outside party and not available to the developer. A disadvantage is that some of the information that has been discarded must be reconstructed, and there are limits to this reversal process.

The FREEDOM compiler [14] is similar to M2V; the compiler accepts binary machine code as input and maps it to an FPGA. The Extensions generated by the M2V compiler are meant for a general-purpose environment and therefore execute in coordination with the main processor data path, whereas FREEDOM maps the entire program to the bare FPGA. M2V generates Extensions that are explicitly interrupt-aware, whereas there is no mention of handling interrupts in the FREEDOM compiler. Additionally, the Extensions generated by the M2V compiler for the eMIPS have secure access to the memory subsystem via the Memory Management Unit (MMU), which is not a requirement for the DSP-like programs handled by the FREEDOM compiler.

Another avenue of research in extensible processors is the identification of the Instruction Set Extensions (ISE) that most benefit a given program, see for instance [4] for a recent overview. Bonzini [4] advocates generating the ISE from within the compiler, Tensilica [11] from profiling data. M2V currently follows the application profiling approach; it uses the BBTools and dynamic full-system simulation with Giano to select the candidate basic blocks. The current approach can extend to handling chains of blocks e.g. in frequently executed loops that are automatically recognized via full-system simulation [15]. A possible addition to our work is to use M2V in concert with a high-level compiler. The compiler can identify the ISE and pass it to M2V for compilation to HDL.

A related area is the generation of HDL code from C, the so-called C-to-gates design flows [8, 9]. The common target is the automated generation of HDL code from sequential programs. The main difference with M2V is that the input is binary code. Using binary code supports all programming languages, included dynamically generated (jitted) code. It is the only viable option in case the high level source code is not available, e.g. for third-party code and libraries. The drawback is that it makes the problem harder. The binary code has already been optimized (register allocation, loop unrolling, etc) during its compilation hence identifying parallelism is more difficult. The BBTools framework tries to account for some of these optimizations by using a canonical form of the basic block, so that it can identify repeating basic blocks in the binary.

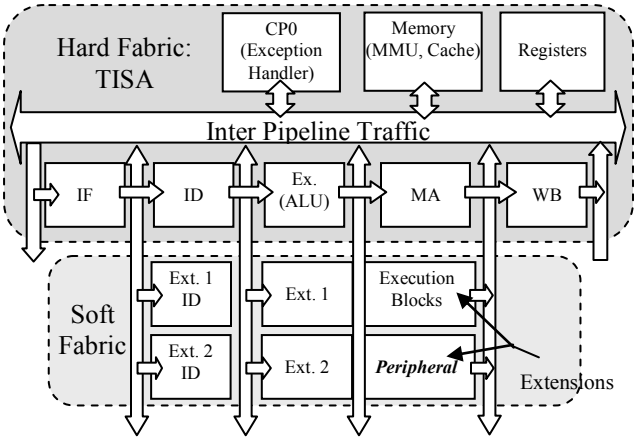


Figure 2. eMIPS block diagram. The Soft Fabric can be reconfigured at run time to extend the ISA.

3. eMIPS HARDWARE OVERVIEW

The extensible MIPS (eMIPS) processor, shown in Figure 2, is an example of a RISC processor tightly integrated with programmable logic. The programmable logic has many uses, including: extensible on-line peripherals, zero overhead online verification of software, hardware acceleration of general-purpose applications, and in-process software debugging [1]. This paper is concerned with automatically generating hardware accelerators within the context of the eMIPS extensible processor. The instruction set for the eMIPS processor is the instruction set for the R4000 MIPS processor [7]. The eMIPS pipeline follows the classic RISC pipeline [6] consisting of five stages: instruction fetch (IF), instruction decode (ID), execute (EX), memory access (MA), and register write-back (WB). The programmable logic is tightly integrated with the RISC pipeline;

it can synchronize with it and it can access the same resources as the RISC pipeline. Figure 3 illustrates the pipelining of instructions through eMIPS.

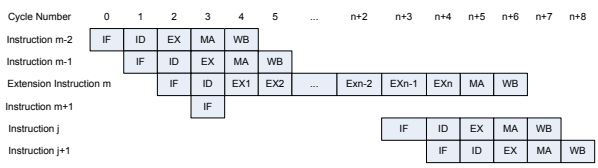


Figure 3. Instruction flow through the eMIPS pipeline.

The decode logic in the extension logic is always an observer of the main pipeline and is trying to decode the instruction in the instruction decode (ID) phase of the pipeline. When the instruction is not an extension instruction, the Extension fails to decode it and the instruction is executed in the main pipeline. If instead the extension logic successfully decodes the instruction, the extension becomes active and hardware acceleration takes over execution. Instructions flowing through the main RISC pipeline prior to the extension instruction complete normally. Instructions following the extension instruction are stalled until the Extension is near completion, in the EXn-1 cycle.

The RISC pipeline imposes micro-architectural constraints on the extension logic, for instance in the arbitration for access to the register file and other resources. The extension logic needs to read and write the register file and access the memory management unit (MMU). M2V automatically schedules all resource accesses in the extension logic to avoid conflicts with the primary RISC pipeline. Thus, register writes must be delayed by the Extension until previous instructions are retired and register reads must finish a couple of cycles before the trailing instructions get to the ID stage. As a specific example, consider the case in Figure 3 when the extension instruction is in the EX1 cycle of execution, instruction m-1 is in the MA pipeline stage and so instruction m-1 has access to the MMU. Instruction m-2 is in the WB pipeline stage and it has control of the register file write ports. The extension instruction does not have control of all the resources until stage EX3 when the previous instructions have been retired.

Opcode_name rt, rs, immediate

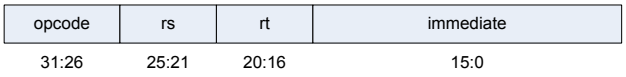


Figure 4. The MIPS "I" format encoding.

The eMIPS processor has been implemented on Xilinx Virtex 4 FPGAs using the ML401 and ML402 evaluation boards. The partial reconfiguration capabilities of this FPGA model allow software to load dynamically the hardware for the instruction extensions.

4. INSTRUCTION ENCODING

An extension instruction is an instruction that is not part of the base ISA of the eMIPS processor. It is inserted in the instruction stream for the specific purpose of triggering an Extension. If the Extension is present and active, it recognizes the instruction and takes over execution. Otherwise the instruction is treated as a NOP and execution continues with the original basic block. In this section we describe how we automatically generate the extended instruction encodings.

In general, an Extension is free to use the instruction bits in any way, provided that the top 6 bits use an invalid opcode. The implementation of the eMIPS decoder presents an opportunity for optimization. By default, the decoder expects "I" format instructions (Figure 4) and fetches the corresponding rs and rt registers in advance. If the extended instruction uses this format it can make use of those two registers immediately, without any penalty.

The input to the M2V compiler are block descriptions in the so-called BBW text file format. *BBMatch* is a program, part of the BBTools framework, used for creating BBW source files automatically, from a MIPS executable file. A second tool reads the BBW file and applies it to a MIPS ELF binary image. For each basic block that matches the BBW file the tool inserts the corresponding extended instruction before the block itself. The BBW description is actually a constrained pattern of canonical instructions, it will match any sequence with a compatible register assignment. For the example in Figure 5, the BBW file might say that rx=R1 and ry=R2. Any sequence of the two instructions OR+SLLV is a candidate for matching. An actual match might require R3=3, … R6=6, but allow any register number in the R1-R2 positions. Alternatively, it might require that R3=ry+1, R4=ry+2, etc etc.

```
[0] ext0 rx, ry, offset
[4] or r5, r1, r2
[8] sllv r6, r3, r4
```

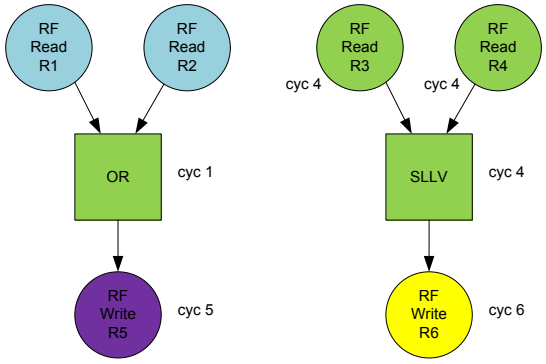
Figure 5. The choice of rx and ry in this basic block affects the performance of the generated Extension.

The selection of the registers to encode in the extension instruction plays an important part in the schedule that M2V will generate for the Extension. This is illustrated by the simple example basic block shown in Figure 5. The cost of the OR and the SLLV instructions in the basic block are 1 and 0 cycles respectively. We shall consider two cases to illustrate the importance of encoding the correct registers. M2V generates the circuit graphs shown in Figure 6 for two different encodings of the extension instruction. A *circuit graph* [13] is essentially a Control and Data Flow Graph [4] decorated with the costs of operations and the resulting execution schedule. In the graphs, the clock cycle when the respective node completes is depicted next to the node. The graphs show that though the number of states in the Extension remains the same, the number of clock cycles taken by the Extension to execute the set of instructions differs based on the encoding of the registers.

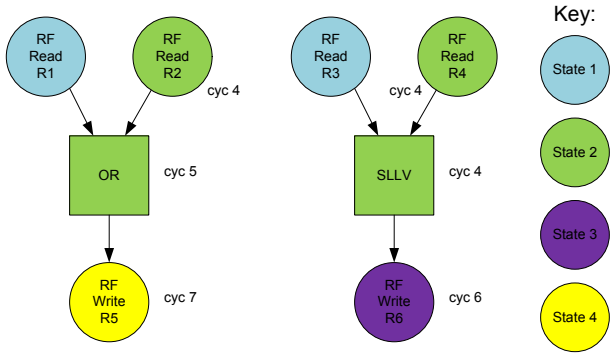
Considering the first case, registers R1 and R2 are encoded, thus making them available directly in stage 2 of the extension state machine. The OR instruction can be executed immediately, and will complete in cycle 1 since the Extension has all the registers available and no unmet dependencies. However, the SLLV instruction requires both source registers to be read from the register file, which takes 4 clock cycles. This causes the SLLV instruction to complete in cycle 4. A pipeline stage is inserted by the extension state machine after execution of the instructions at cycle 4. The two register write-backs are performed after the pipeline stage, in cycle 5 (R5) and cycle 6 (R6). Thus, the

Extension requires 6 cycles to complete execution with this encoding.

Considering the second case, registers R1 and R3 are encoded. In this case, none of the instructions can be executed directly as both have unmet dependencies and require register reads from the register file. Assuming there are at least two read ports in the register file, the OR instruction completes after 5 cycles, 4 cycles for reading register R2 and 1 cycle for execution. Similarly, the SLLV instruction completes execution in cycle 4. Again, a pipeline stage is inserted after execution of the instruct-



Case (i): Extension instruction encoded with R1, R2



Case (ii): Extension instruction encoded with R1, R3

Figure 6. Circuit graphs for the block in Figure 5, using different encoding schemes.

tions in cycle 5. The register write-backs are performed in cycle 6 (R6) and cycle 7 (R5). Thus the Extension requires 7 cycles to complete execution with this encoding. In this minimal example, a two instruction basic block shows a difference of 1 execution cycle depending on the selected encoding. The encoding scheme will have a greater impact on the execution time when there are long latency paths in the basic block.

4.1 Register Selection Algorithm

The encoding algorithm uses two main parameters in selecting the rs and rt registers – fan-out and depth of the root register read nodes. Fan-out is the number of instructions dependent on the root register read node. Depth is a count of the register nodes and the cost of the instruction nodes till a dependency is met in the graph. Using the circuit graphs in Figure 6, all the root nodes, R1, R2, R3 and R4 have a fan-out of 1. For the depth calculation, all the registers have a dependency at the instruction nodes, with the only differentiating factor being the cost of the OR instruction node compared to the SLLV instruction node. This gives the depth of registers R1-R2 as 2 and R3-R4 as 1. The algorithm takes the sum of the fan-out and depth of the register nodes and encodes the nodes with the maximum value. In the previous example, the nodes encoded by the algorithm would be R1 and R2, which is the best encoding scheme as seen from the circuit graphs in Figure 6.

Other algorithms are possible. The total number of general-purpose registers is limited though abundant in MIPS, and the calling convention further restricts the number of maximum potential roots in any practical dependency graph. It is therefore practical to perform a brute-force exhaustive search for the selection with the optimal cycle count. The worst-case number of alternatives for a processor with N usable registers is $\frac{n(n-1)}{2}$ or 465 for MIPS. We will explore this alternate approach when the compiler has full code coverage.

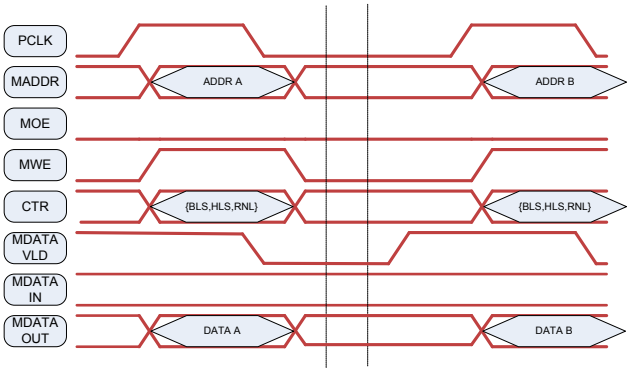


Figure 7. Memory Write Protocol.

5. MEMORY ACCESS SUPPORT

It is well known in the literature that the lack of load and store operations leads to limited speedups from hardware acceleration. The operations are not only important for performance, but in our experience they are almost always present in the most-frequently executed basic blocks of an application, precisely the blocks that M2V wants to accelerate.

The eMIPS architecture allows for Extensions to access memory through the MMU just like the main MIPS processor data path. The MMU is part of the Trusted ISA portion of the eMIPS processor. The MMU is the only path to memory available to general, untrusted Extensions. To ensure correct execution of the memory instructions, the Extension must obey the memory protocol shown in Figure 7 for a write case. In both the read and write protocols, the MDATA_VLD signal serves as an indicator that the memory request is acknowledged by the controller. The signal indicates when the data is available from, or to be written to memory. The M2V compiler implements the protocols in Verilog, in the form of a memory state machine. The memory state machine is then integrated into the existing extension state machine. M2V maintains an array of memory operations in a particular state that is integrated into the extension state machine. Once the Extension transitions from one state to a state with a memory operation, the memory state machine is activated.

On the rising edge of the Pipeline CLK (PCLK), the address is latched onto MADDR signal and the Memory Output Enable (MOE) signal is raised in case of a read or the Memory Write Enable (MWE) is raised in case of a write. MDATA_VLD then falls down once the memory request is acknowledged, and the memory state machine moves onto the next state, waiting for the MDATA_VLD to go high, indicating the availability of the data in MDATA_IN for a read or completion of the write for a write operation.

When a state involves a memory access, the main extension state machine waits on the completion of the memory state machine and transitions to the next state when all operations (e.g. register and memory reads and writes) for that state are completed.

6. INTERRUPT SUPPORT

We have modified the M2V compiler to handle interruptions while the processor is executing in the Extension. Interruptions can be due to three different sources, but we will use the single term "interrupt" to indicate any and all of them. Our approach handles all cases in the same manner. The first cause of interrupts is address translation misses and errors in the MMU while the Extension is trying to access memory. A second cause is actual interrupts from peripherals such as timers and I/O devices. A third cause of interrupts is the case of errors inside the Extension, such as unaligned addresses and overflows. We use a transaction model based approach to handle all interrupts in the Extension.

The basic block to be accelerated is analyzed and divided into "transactions". A transaction is a set of instructions that terminate just before a memory instruction. Even in the event the basic block has no memory operations, there is still a maximum number of cycles allowed before interrupts are permitted. The maximum number of operations in a particular transaction is user-selectable, default is 7. Future work should consider the actual latency/cost of the instructions rather than an arbitrary number of instructions. The maximum transaction size should still be user-selectable because it affects the interrupt latency of a real-time system.

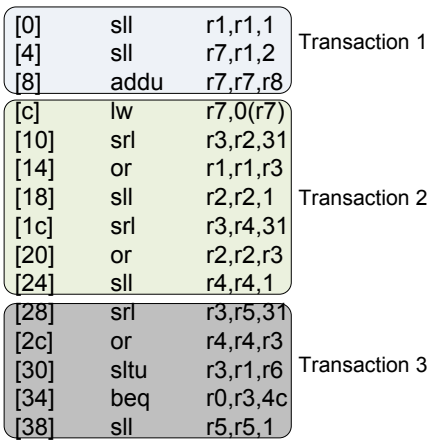


Figure 8. Example basic block, divided in transactions.

The Extension must correctly indicate to the TISA what the re-start address is, e.g. after the software interrupt handler returns. This address is termed the Virtual PC (VIRPC), as the Extension is keeping track of the PC as seen by the MIPS pipeline, even though the Extension has no concept of instruction fetch or instruction ordering. The VIRPC address simply corresponds to the start of each transaction in the original basic block.

We illustrate the subdivision of a basic block in transactions with an example in Figure 8. Transaction 1 terminates once the load instruction ([c]) is encountered. Transaction 2 is terminated at the end of the maximum allowed 7 instructions in the transaction. The remaining instructions are part of transaction 3. The basic idea behind the transactions scheme is to preserve the original program order, while at the same time allowing for more optimistic and more parallel execution inside the Extension. We decided to be very conservative in this first implementation and to leave more aggressive optimizations for future work. The Extension will recognize an interrupt at the next transaction boundary should an interrupt occur during the Extension's execution. Any write-backs that are due to a subsequent transaction are aborted.

The transaction model is used to perform write backs in-order, but from the abstracted viewpoint of a transaction, that is, the write backs in transaction 1 must complete prior to any write backs in transaction 2. However, the write backs inside a particular transaction can be performed out of order. This limits the parallelism generated by the circuit graph to some extent by imposing the restriction of performing certain write backs in order. We perceive this to not be a huge problem as the eMIPS TISA interface allows for two register writes every cycle, thus decreasing the possibility of bottlenecks at the register file.

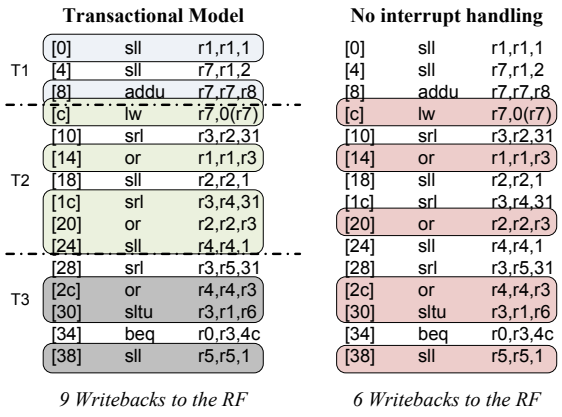


Figure 9. Overhead of handling interrupts.

Figure 9 illustrates the overhead in terms of register write-backs. In the case of generating hardware without the support of interrupts only 6 write-backs are necessary to the register file. It can be clearly seen that further optimizations can be applied to this basic transaction model. By just terminating Transaction 2 an instruction before or after would have reduced the number of write-backs by 2. This would have ensured only one extra write-back in the transactional model approach.

M2V uses transactions registers to keep track of the current transaction being written back for a particular state of the

Extension. When the Extension encounters an interrupt, the extension state machine checks to see if the current state of the Extension is an end of a transaction or not. If it is an end of a transaction, the VIRPC is updated to reflect the address of the start of the next transaction and the extension state machine is stalled in that state. The Extension then waits for the resources to be taken away by the pipeline arbiter and the Enable and Grant signal to go low. Once the enable goes low, the Extension lowers the ACK signal to signal the end of the Extension at that transaction. The program then re-starts execution from the VIRPC address on the main MIPS processor, with the registers and other structures in the correct state.

7. RESULTS

The design generated by the M2V compiler was synthesized using the Xilinx ISE tools. With the transactional model enabled, the percentage of total slices used increased from 3% to 4%, compared to the base version of M2V with memory support. This is because of the extra registers used in the transactional model and the extra state machine required to track the transactions during execution. Overall, the added complexity from interrupts causes a penalty in area utilization. This extra cost is balanced almost exactly by the improvements in the new data path interface of the current eMIPS implementation over the one presented in [13]. There is no penalty in frequency.

Table 1. M2V code coverage test results

No. files	325	
No. blocks	146,057	Percent Total
Compiled ok	25,029	17.1%
Warnings	44,800	30.7%
Failures	76,228	52.2%

We tested the changes to the compiler with the example basic block shown in Figure 8. The basic block is a 64-bit division block with an extra memory instruction (load from the stack pointer) inserted to test the working of memory accesses. To test interrupt handling, we generated timer interrupts at short variable intervals. The Extension was simulated using ModelSim and the test program simulation was run in Giano. The test program checked over 500 test vectors for the 64-bit division and the test passed successfully in all cases. The Extension always reported the correct Virtual PC (VIRPC) address and the transactional state machine worked as designed.

To test for code coverage, we ran the BBMatch and the M2V compiler on 325 executable files from the code base of the MMLite RTOS [16]. BBMatch had 100% coverage and extracted and encoded about 150,000 blocks from these files. We then ran M2V on the extracted blocks. The results are shown in Table 1. The large number of failures is due to a small number of still unsupported instructions, especially JAL, BLTZ, BGEZ, MULT, DIV, SLLV, and SRLV. This work is in progress but M2V is already useful even without them.

8. CONCLUSIONS

We have modified the eMIPS tool chain to remove the last remaining obstacles for a fully-automated generation of hardware accelerators. By supporting load and stores, interrupts, and the automatic encoding of extended instructions the compiler can now attack the single-block cases of practical applications. Code coverage is already 50% of the blocks in more than 300 executable files, with only a few unsupported instructions responsible for most of the failures. The addition of interrupt support to the M2V compiler is especially relevant because there is now no limit to the span of an accelerator, even in a general-purpose environment. An arbitrarily long sequence of instructions can be accelerated, without concerns for security or real-time responsiveness.

Support for interrupts in the compiler causes the loss of a little amount of parallelism, because of the in-order write-backs requirement. Using a transactional model mitigates this effect. Performing two write-backs to the register file in every cycle of the Extension further mitigates this effect. The overhead of transactions would be minimal in the case of large basic blocks with a large number of extension states.

9. REFERENCES

[1] Almeida, O., et al. *Embedded Systems Research at DemoFest'07*. Microsoft Research Technical Report MSR-TR-2007-94, July 2007.

[2] Altera Corp. *Excalibur Embedded Processor Solutions*, '05.

[3] Biswas, P., Banerjee, S., Dutt, N., Ienne, P., Pozzi, L. *Performance and Energy Benefits of Instruction Set Extensions in an FPGA Soft Core* VLSID'06, pag. 651-656

[4] Bonzini, P., Pozzi, L. *Code Transformation Strategies for Extensible Embedded Processors* CASES'06.

[5] Forin, A., Lynch, N., L., Pittman, R. N. *eMIPS, A Dynamically Extensible Processor*. Microsoft Research Technical Report MSR-TR-2006-143, October 2006.

[6] Hennessy, J. L., Patterson, D.A. *Computer Organization and Design: The Hardware/Software Interface.* Morgan Kaufmann Publishers, San Francisco, CA. 1998.

[7] Kane, G., Heinrich, J. *MIPS RISC Architecture*. Prentice Hall, Upper Saddle River, NJ. 1992.

[8] Kastner, R., Kaplan, A., Ogrenci Memik, S. Bozorgzadeh, E. *Instruction generation for hybrid reconfigurable systems* TODAES vol. 7, no. 4, pagg. 605-632, October 2002.

[9] Lau, D., Pritchard, O., Molson, P. *Automated Generation of Hardware Accelerators with Direct Memory Access from ANSI/ISO Standard C Functions. FCCM'06, pagg. 45-54, April 2006.*

[10] *Stretch, Inc*. http://www.stretchinc.com 2006.

[11] Tensilica, Inc. http://www.tensilica.com, 2006.

[12] Xilinx Inc. *Virtex 4 Family Overview*. Xilinx Inc., June 2005.

[13] Meier, K., Forin, A. *MIPS-to-Verilog, Hardware Compilation for the eMIPS Processor*, MSR-TR-2007-128, Microsoft Research, WA, September 2007.

[14] Mittal, G., Zaretsky, D.C., Xiaoyong Tang, Banerjee, P. *An overview of a compiler for mapping software binaries to hardware* IEEE VLSI, 2007.

[15] Chandrasekhar, V., Forin, A. *Mining Sequential Programs for Coarse-grained Parallelism using Virtualization*, MSR-TR-2008-113, Microsoft Research, WA, August 2008.

[16] Available at http://research.microsoft.com/invisible/

[17] Available at http://research.microsoft.com/research/EmbeddedSystems/Giano/giano.aspx