Design and Applications of a Virtual Context Architecture

David Oehmke, Nathan Binkert, Steven Reinhardt, Trevor Mudge

Department of Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, MI 48109-2122

ABSTRACT

This paper proposes a new register-file architecture that virtualizes logical register contexts. This architecture makes the number of active register contexts—representing different threads or activation records—independent of the number of physical registers. The physical register file is treated as a cache of a potentially much larger memory-mapped logical register space. The implementation modifies the rename stage of the pipeline to trigger the movement of register values between the physical register file and the data cache.

We exploit the fact that the logical register mapping can be easily updated—simply by changing the base memory pointer—to construct an efficient implementation of register windows. This reduces the execution time by 8% while generating 20% fewer data cache accesses. We also use the large logical register space to avoid the cost of a large physical register file normally required for a multithreaded processor, allowing us to create an SMT core with fewer physical registers than logical registers that still performs within 2% of the baseline. Finally, the two are combined to create a simultaneous multithreaded processor that supports register windows. This architecture achieves a 10% increase in performance over the baseline architecture even with fewer physical than logical registers while also reducing data cache bandwidth. Thus we are able to combine the advantages of register windows with multithreading.

1 INTRODUCTION

Registers are a central component of both instruction-set architectures (ISAs) and processor microarchitectures. From the ISA's perspective, a small register namespace allows the encoding of multiple operands in an instruction of reasonable size. Registers also provide a simple, unambiguous specification of data dependences, because—unlike memory locations—they are specified directly in the instruction and cannot be aliased. From the microarchitecture's point of view, registers comprise a set of high-speed, high-bandwidth storage locations that are integrated into the datapath more tightly than a data cache, and are thus far more capable of keeping up with a modern superscalar execution core.

As with many architectural features, the abstract concept of registers can conflict with real-world implementation requirements. For example, the dependence specification encoded in the ISA's register assignment is adequate given the ISA's sequential execution semantics. However, out-of-order instruction execution requires that the ISA's logical registers be renamed into an alternate, larger physical register space to eliminate false dependencies.

This paper addresses a different conflict between the abstraction of registers and its implementation: that of *context*. A logical register identifier is meaningful only in the context of a particular procedure instance (activation record) in a particular thread of execution. From the ISA's perspective, the processor supports exactly one context at any point in time. However, a processor designer may wish for an implementation to support multiple contexts for several reasons: to support multithreading, to reduce context switch overhead, or to reduce procedure call/return over-

head (e.g., using register windows) [18, 33, 35, 7, 23, 29]. Conventional designs require that each active context be present in its entirety; thus each additional context adds directly to the size of the register file. Unfortunately, larger register files are inherently slower to access. Thus additional contexts generally lead to a slower cycle time or additional cycles of register access latency, either of which reduces overall performance. This problem is further compounded by the additional rename registers necessary to support out-of-order execution.

The context problem has already been solved for memory. Modern architectures are designed to support a virtual memory system. From the processes perspective, each process has it's own independent memory, only limited by the size specified by the ISA. The architecture and operating system efficiently and dynamically manage the hardware memories (cache, RAM and disk) to maximize the memory performance of each process.

What is needed is a system with the advantages of a register, but with the context free nature of memory. Such a system can be realized by mapping the registers to memory. The question becomes when does this mapping occur. The registers go through several transformations between source code and execution in a processor. This steps are summarized in Figure 1. At initial compilation, the source code variables and temporary values are converted into virtual registers. At register allocation, the compiler maps the virtual registers into logical registers and the assembly code/binary is generated. Finally, in an out of order processor, at rename, the logical registers are mapped into physical registers to remove any false data dependencies imposed by the logical registers. The two obvious choices for memory mapping the registers are at register allocation or decode/rename. The compiler can map virtual registers directly into memory at register allocation. In this case, the ISA will no longer have a notion of registers, and will instead directly work with memory addresses. The microarchitecture could also manage this mapping. In this case, the compiler generates a more traditional assembly with register specifiers. In the frontend of the pipeline, the microarchitecture maps the register into memory, thus alleviating the context problem.

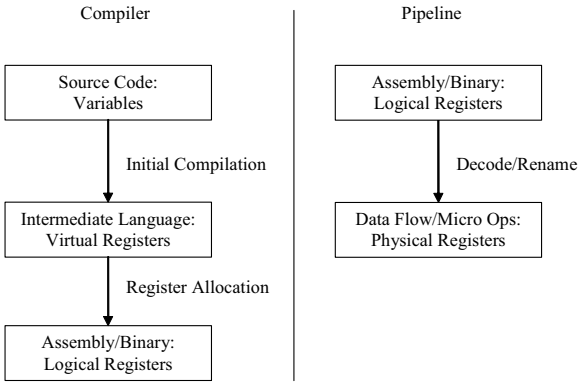

Figure 1: Register Specifier Transformations. The registers go through several transformations from source code until the time the assembly instructions are executed. At initial compilation, the source code variables and temporary values are converted into virtual registers. At register allocation, the compiler maps the virtual registers into logical registers and the assembly code/binary is generated. Finally, in an out of order processor at rename, the logical registers are mapped into physical registers to remove any false data dependencies imposed by the logical registers.

1.1 Mapping compiler virtual registers to memory: The Mem-machine

By mapping the compiler virtual register directly to memory, we completely remove registers from the instruction set architecture. This creates a memory to memory instruction set architecture. Such an ISA would have the advantage of unlimited resources (all of virtual memory) to use as storage locations. The code would also be free of explicit loads and stores, resulting in efficient execution. However, this type of instruction set architecture would also suffer from some serious

disadvantages. As previously stated, registers allow for efficient encoding, unambiguous data dependencies and tight coupling into the microarchitecture's data path. One serious disadvantage would be the size of the instructions. As stated previously a register specifier can be encoded in a small number of bits. Encoding a memory address would require a large number of bits per operand. Section 2 contains a detailed description of the mem-machine and reports some initial results.

1.2 Mapping ISA logical registers to memory: The Virtual Context Architecture

Delaying the memory mapping to the frontend of the pipeline, gives all the advantages of registers with the automatic context management of memory. We seek to bypass this trade-off between multiple context support and register file size by decoupling the logical register requirements of active contexts from the contents of the physical register file. Just as caches and virtual memory allow a processor to give the illusion of numerous multi-gigabyte address spaces with an average access time approaching that of several kilobytes of SRAM, we propose a register cache that gives the illusion of numerous active contexts with an average access time approaching that of a conventionally sized register file. Our design treats the physical register file as a cache of a practically unlimited memory-backed logical register space. We call this scheme the virtual context architecture (VCA). An individual instruction needs only its source operands and its destination register to be present in the register file to execute. Inactive register values are automatically saved to memory as needed, and restored to the register file on demand. A thread can change its register context simply by changing a base pointer—either to another register window on a call or return, or to an entirely different software thread context. Compared to prior register cache proposals (see Section 7), VCA:

- unifies support for both multiple independent threads and register windowing within each thread;
- is backwards compatible with existing ISAs at the application level for multithreaded contexts, and requires only minimal ISA changes for register windowing;
- requires no changes to the physical register file design and the performance-critical schedule/execute/writeback loop, building on existing rename logic to map logical registers to physical registers and handles register cache misses in the decode/rename stages;
- completely decouples physical register file size from the number of logical registers by using memory as a backing store, rather than another larger register file; and
- does not involve speculation or prediction, avoiding the need for recovery mechanisms.

The virtual context architecture provides a near optimal implementation of register windows, improving performance while greatly reducing traffic to the data cache (by up to 8% and 40%, in our simulations). It also enables more efficient implementations of SMT, maintaining throughput while requiring fewer physical registers than is possible on a normal architecture. Finally, VCA easily supports both SMT and register windowing simultaneously on a conventionally sized register file, avoiding the multiplicative growth in register count that a straightforward implementation would require.

Section 2 describes the mem-machine, the precursor to the virtual context architecture. Section 3 describes the architecture in more detail. Section 4 specifies the testing environment and the specific machine architecture assumed. Section 5 evaluates the architecture as an implementation of register windows. Section 6 evaluates the architecture on a simultaneous multithreaded processor. Section 7 presents previous work. Section 8 concludes and Section 9 discusses future work.

2 MEM-MACHINE

The mem-machine is a memory to memory instruction set architecture. The ISA supports directly accessing memory for all operands, thus eliminating the need for general purpose registers. It has several advantages over a traditional register to register architecture. One performance advantage is the elimination of all explicit load and store instructions. The elimination of the general purpose registers also greatly reduces the size of the context necessary for each thread. This would allow efficient context switches and large scale multithreading. The architecture is also very efficient at emulating other types of machines. To emulate a machine the internal state, including the registers, is kept in memory. To execute an emulated instruction for a register to register machine requires a load of the all the values from memory into registers, then the instruction is executed and finally the result copied back to memory. An architecture that can access memory for each operand, can directly execute the operation without any extra loading or storing.

The main disadvantages of this architecture are its cache performance and code size. All values are stored in memory, and all operands access memory. This will place a much heavier burden on the caches than a register to register architecture. The code size is also likely to be much larger. To provide the information for an operand to access memory will require more bits than simply specifying one of a small set of registers. For example, a memory address is usually 32 bits, while a register can be specified using only 5 bits for most architectures. This means that instructions will be several times larger for this architecture than instructions for an equivalent register to register architecture.

The project consisted of three phases. The first phase was the design of the architecture, instruction set and application binary interface (ABI). The second phase was the implementation of the infrastructure. The final phase was the evaluation of the new architecture.

2.1 Phase 1: Design of the ISA and ABI

2.1.1 *Design Decisions*

To support a modern programming language like C, an ISA needs to have support for function calls and the separate compilation and linkage of execution units. This is usually accomplished by creating a stack in memory. In most ISAs the top of the stack (stack pointer) is kept in a specific register. Any values that need to be communicated across a function call can be placed in memory at specific offsets from the stack pointer. Although many ABIs use additional specific registers to pass some of the information, for example the return address and several parameters, this is only done for efficiency. The notion of a stack is also convenient because it provides a simple and efficient mechanism for each function to allocate local storage in a dynamic fashion. This dynamic allocation is easily accomplished by growing the stack (adjusting the stack pointer). Separate compilation and linkage can be achieved because the stack pointer is a global resource available to all functions. The ABI specifies the location of all the arguments on the stack, and the current location of the top of the stack is communicated to the called function.

For the mem-machine to support a stack based ABI the stack pointer must somehow be communicated from the calling function to the callee. Another important issue is that in these stack based systems, all local values are addressed via offsets from the stack pointer. Thus, not only must the stack pointer be in some shared location, but for reasonable performance, there must be an efficient way to access these local values. This becomes especially important if you eliminate all general purpose registers. In this case not only will local variables be kept on the stack, but so must all temporary values. This leads to an additional requirement: an operand or destination of an instruction must be able to address a stack location. In other words, a stack location becomes the basic location type, or at least the minimal that must be supported.

One solution is to assign the stack pointer a specific address. Since one fixed address is used, easy communication of the value can be accomplished. The callee can simply read the value from this location to determine the current position of the stack pointer. To support the minimal addressing mode for operands requires that each operand be able to access a location relative to the stack. In this case an address and offset would be specified for each operand. The address is read to provide a base pointer. The offset is added to the base pointer to generate a new address. The final value is read from this new address. There are two problems with this approach. One obvious problem is that both an address and offset must be allowed for each operand in an instruction. Since, all local values and temporaries will be stored on the stack, large offsets should be supported. When combined with a 32 bit address, this would mean a large number of bits per operand. The second problem is that this would require two memory accesses to read/write a value to/from a location on the stack. Since, all locals and temporaries are located on the stack, a typical two source and one destination instruction would require six memory accesses to execute.

The other possibility is to support a stack pointer register. The system will no longer be completely registerless, but it still won't have any general purpose registers, only a single special purpose one. As part of the opcode of the instruction a bit would specify whether to use the stack pointer. This would allow the operand to be reduced to a single value. If the bit is set, this value would be treated as an offset from the stack pointer. If the bit isn't set, it could be treated as an absolute address. This would reduce the operand to a more manageable size. A typical instruction would also require only three memory accesses to execute. Considering that all values are stored in memory, this is as efficient as it can be. The advantages this solution has over using a fixed address make it the obvious choice.

This simple displacement addressing is not adequate. Languages like C support accessing a value through a pointer. To support this, the ISA must be able to read in a value from an address determined at runtime. At this point the operands can only do this based on the stack pointer. A more general solution is required. In particular the ISA must support reading a value from an address stored in a location on the stack. A few special instructions could be added to accomplish this. They would act like the standard load/store in a register ISA, but would only be needed when reading/writing to/from a dynamic address. However one of the goals of the ISA is the elimination of explicit loads and stores. A more natural solution is expanding the addressing modes allowed for each operand. Specifically, one more level of indirection could be allowed for each operand. Instead of reading the value directly from the location on the stack, the value would be treated as an address and the final value would be read from that location in memory. A similar scheme would be used for destinations with the final access being a write instead of a read. This provides all the necessary functionality to support C.

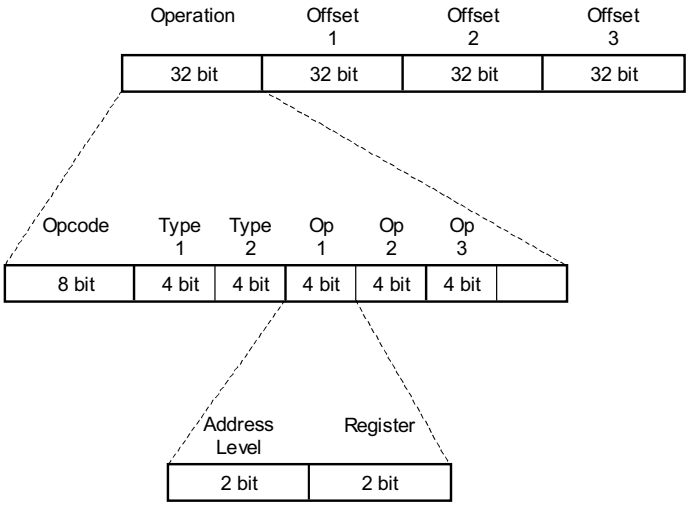

Figure 2: Instruction Binary Encoding Format. The mem-machine instructions are encoded in a 128 bit instruction. The instruction format is separated into set fields for easy decoding. The operation field contains the opcode of the instruction in the first 8 bits. The next 8 bits specify up to two value types for the instruction. Finally, the operation field contains one 4 bit field for each operand. The first 2 bits in this field specify the address indirection level of the operand. The other 2 bits specify one of three possible registers to use for the operand: none(zero), stack pointer, or frame pointer. The rest of the instruction is composed of three 32 bit offsets, one for each of the operand.

2.1.2 *ISA and ABI*

The ISA for the mem-machine is modeled on the Portable Instruction Set Architecture(PISA)[4]. The mem-machine supports all the standard operations and is similar to most RISC like ISAs (see Appendix A for a list of supported operations). The ISA was designed to be extremely easy to target for a compiler and easy to decode. In particular the operation has separate fields for the various options. The basic instruction is 128 bits (see Figure 2). It includes a 32 bit operation and a full 32 bit offset for each operand. The first field is the opcode and signifies the type of instruction. It supports all the standard operations including a full set of conditional branches and conditional sets. This was done to minimize the amount of work needed to translate from the compiler intermediate format to the final instruction. The opcode is independent of value type. The next two 4 bit fields specify up to two of the ten supported value types. The architecture supports 32 and 64 bit floating point types, and 8, 16, 32, and 64 bit signed and unsigned integer types. Depending on the opcode, zero, one or two types are needed. For example, a jump requires no type, add requires a single type, and convert requires two types. Finally, there are three 4 bit fields specifying how to treat the three operands. Each of these fields has two subfields. The first is the address level. This specifies the number of levels of indirection to use when reading or writing the value (see Table 1). The second field specifies the register to use - none (zero), stack pointer, or frame pointer. The zero register is used to specify absolute addresses or constant values. The frame pointer is needed to support alloca. See Appendix B for an example of the assembly language for a simple function.

Address Level	Read (Source)	Write (Destination)
0	Value = Register + Offset	Offset must be zero and Register must be the stack pointer or frame pointer. Register = Value
1	Value = Mem[Register + Offset]	Mem[Register+Offset]=Value
2	Value = Mem[Mem[Register + Offset]]	Mem[Mem[Register+Offset]]=Value

Table 1: Address Indirection Level Description. The mem-machine supports three levels of address indirection for each operand. Level 0 is used for constant values, addresses and to read or write to registers. This level requires no memory accesses. Level 1 is used to read or write to stack locations or global variables. It requires one memory access. Finally, level 2 is used for read or writing through a pointer value. It requires two memory accesses.

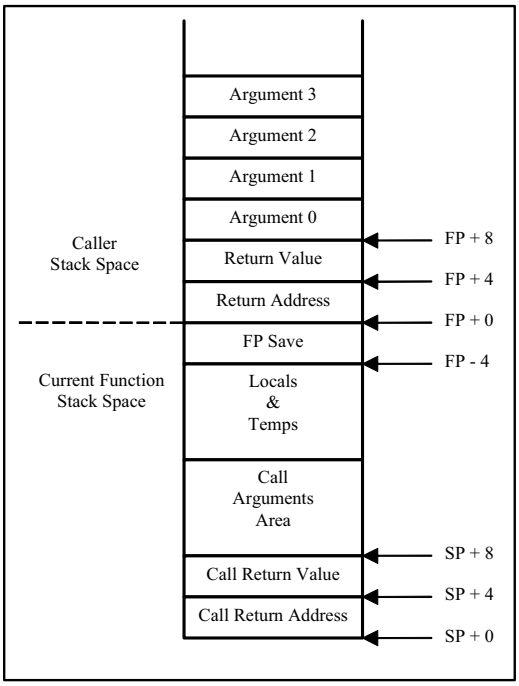

Figure 3: Mem-Machine Stack Layout. The stack layout of the mem-machine is similar to other RISC machines. The frame pointer (FP) is used to access the function arguments, function return, local variables and temporary values. The stack pointer (SP) is used to access the arguments and return value of any called functions. The return address is located at frame pointer + 0. The return value placeholder is at frame pointer + 4. Even if the function does not return a value this space is reserved. The function arguments start at frame pointer + 8. The arguments are placed in order aligned to the appropriate address.

The ABI for the mem-machine is also similar to PISA. Like PISA, the stack grows down. PISA uses registers to pass some of the arguments, the return address and to return any values. The mem-machine uses the stack pointer register to communicate the location of the stack, all other values that need to be communicated are placed in fixed positions on the stack, see Figure 3. The return address is place at the stack pointer. The return value placeholder is next on the stack at

the stack pointer plus 4. If the return value is four bytes or less, it is placed directly at this location by the callee and no initialization needs to be done by the caller. If the return value is larger, the calling function allocates stack space for the value and initializes the placeholder with the address of this location. The callee uses this pointer to store the result. This is similar to how PISA handles structure return values. Even if the function does not return a value, this space is reserved. This allows the arguments to always start at the same location and enables the compiler to setup a function call without knowing any information about the function being called. The arguments are placed starting at the stack pointer plus 8 and aligned accordingly. The function prologue and epilogue are similar to other stack based systems. The prologue is composed of three things. First, the current value of the frame pointer is saved. Second, the stack pointer is copied into the frame pointer. Finally, a constant value is subtracted from the stack pointer to allocate all the local storage. The epilogue reverses the prologue. First, it restores the original stack pointer by setting it equal to the frame pointer. Second, it restores the original frame pointer by loading it back from it's save location. Finally, the function returns using the return address saved at the stack pointer.

2.2 Phase 2: Implementation

The implementation of the Mem-Machine required creating a complete build tool chain and simulation environment. This required four major things. First, a compiler needed to be modified to target the Mem-Machine. Second, a C library needed to be ported. Third, a complete set of binary utilities needed to be created, including an assembler, linker and archiver. Finally, a simulator was ported to the new architecture to allow it to be evaluated.

2.2.3 *Compiler*

The compiler is a retargeted version of MIRV[12]. This required writing a checker module, printer module and modifying the register allocation. The checker module is responsible for translating the internal assembly instructions into a form that could be executed on the target machine. For this ISA, this required two main things. First, the operands of each instruction had to be normalized into a form that the ISA could handle. In particular operands that used displaced addressing (besides from the stack pointer) or indexed addressing had to have an add instruction inserted before them to calculate the address and place it into a temporary variable. The problem operand was then switched to de reference this temporary value. Secondly, the ABI had to be implemented. This involved setting up the call stack for function calls and inserting the function prologue and epilogue. The printer module simply had to print the instructions in the format expected by the assembler. A very easy to parse format was used. Finally the register allocation needed to be modified. The standard allocator allocates architectural registers for each virtual register. If not enough are available or for certain other cases, the virtual register is instead assigned a spill location on the stack. This is appropriate for an ISA with general purpose registers, but is not appropriate for one using memory operands. A very simple allocator for this new architecture was implemented. It assigns a separate spill location to each virtual register. While not very efficient, it is very simple to implement. This has several repercussions though. MIRV relies on the register allocator to provide copy propagation. This simple allocator doesn't do it. Therefore the generated code will have quite a few extra moves. Secondly, this means that no reuse of stack locations will take place. This will have a negative effect on the cache performance and on the compressibility of the code.

2.2.4 *C Library*

A C library was provided by porting newlib[5]. Newlib was chosen because it was created to be as portable as possible. The task involved two main challenges. The first was to prepare the library for compilation by MIRV instead of a gnu compiler. This involved working around gcc specific options and defining the appropriate symbols. The second task was to provide the target specific code. This was primarily the startup code, and the interface to the system calls. The system call interface is provided by a set of functions written in assembly language.

2.2.5 *Binary Utilities*

The binary utilities were created from scratch. The utilities included an assembler, archiver, ranlib and linker. In each case, the simplest but still functional version was created. Specifically, they only needed to support the options and the functionality required by MIRV and the newlib makefiles. The assembler is responsible for reading in the assembly file produced by the compiler and converting it into an object file. No attempt was made to perform any optimizations or to keep the file size down. The object file format is essentially a stored version of the structures used by the assembler. The purpose of the archiver is to bundle several object files together to form libraries. In order to build the library, it needed to both insert and extract object files. A ranlib executable was also created, but it was just a stub and had no functionality; its only purpose was to exist because it was called by the newlib makefiles. The linker takes a set of object files and libraries and creates the final binary. Its two primary functions are to link all the labels and to layout the code and data. The label linkage needs to resolve all symbol labels to their final address. Some labels are local and defined in the same object file they are used. Other labels are global; these are defined in one object file, but may be used in other object files. To keep the code size of the executable small, the minimal set of object files should be linked. To accomplish this, the linker starts with the object file that defines the entry symbol and links it. As each object file is linked it is checked for any undefined global labels. For each undefined label, the object files are searched until the one that defines it is found. If this object file is not already linked, it is linked now, and in turn checked. This recursive procedure continues until all the globals are resolved, in which case the compilation is successful. If a global label cannot be resolved the linker exits with an unresolved symbol error. Once all the files have been linked, it inserts a special label to mark the end of the data. This is used by the code as the starting point for the heap. Next, the code and data layout is done. This determines the address of everything in the object files, obeying any alignment issues and size requirements. Finally, the executable is written to a file in a binary format.

2.2.6 *Simulator*

The simulator was provided by a new target for SimpleScalar. The port required creating several files. The header file describes the basic features of the architecture, including the instruction layout and register file description. The instruction definition file contains descriptions of the instructions, how they are decoded and their C implementation. The decode could be done directly, similar to PISA, because of the simplicity of the binary encoding. The loader file provides a function to load the executable from its file, and copy the code and data into the simulated memory. It is also responsible for setting up the arguments and environment in the simulated memory in the positions expected by the target code. The final file primarily provides debugging features, in particular a function to print out a disassembled instruction in a human readable format. Most of the simulator files required no modification. However, no attempt was made to try to port the out of order simulator because of the complexity of the operands.

	art00			compress95			equak00		
	O0	O1	O2	O0	O1	O2	O0	O1	O2
Single Float	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Double Float	11.01	20.54	20.54	0.43	0.46	0.47	11.27	18.19	18.11
Signed Byte	0.00	0.00	0.00	3.59	3.53	3.54	0.37	0.60	0.60
Signed Half	0.00	0.00	0.00	0.00	0.00	0.00	0.07	0.12	0.11
Signed Word	11.04	9.55	9.55	28.98	33.12	33.23	11.31	16.01	16.22
Signed Long	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Unsigned Byte	0.01	0.01	0.01	7.45	7.99	8.02	0.40	0.65	0.64
Unsigned Half	0.00	0.00	0.00	1.97	2.11	2.12	0.20	0.32	0.32
Unsigned Word	74.57	57.94	57.94	57.52	52.50	52.35	73.70	58.53	58.35
Unsigned Long	3.37	11.95	11.95	0.06	0.28	0.28	2.70	5.60	5.64

Table 2: Dynamic instruction value types. The percentage that each value type appears in the complete dynamic execution of each of the benchmarks. The results are given for the three benchmarks at all three optimization levels.

2.3 Phase 3: Evaluation

A few of the SPEC benchmarks[14] were compiled using the new system and the results verified against the reference output. The benchmarks were art00, compress95 and equak00. Each was compiled using three different levels of optimization: O0, O1, and O2. Two studies were performed. In the first, the dynamic instructions are characterized to generate a code profile. In the second, the mem-machine is compared against PISA to evaluate it's effectiveness.

2.3.7 *Code Profile*

This section contains a profile of the dynamic instructions executed by the benchmarks. Three statistics are examined: instruction value type, operand register usage, and operand indirection level.

The results of the value type profile are shown in Table 2. In every case the most common type by far is the unsigned word. This is the type used for pointer arithmetic and for moving 32 bit values, irregardless of the type. Any extra moves in the system, for 32 bit values, will be of this type. In general as the optimization level is increased, the percentage of this type tends to decrease. This is most likely due to the reduction in extra moves and a general increase in the efficiency of the code. The two benchmarks that contain a significant amount of floating point code show an additional similar trend. As the optimization level increases, the percentage of code involving the double precision type increases. This is a good indication of the increasing efficiency of the code. Note that at the same time the percentage of unsigned long type also increases. Similar to the unsigned word, this type is probably used for moves involving 64 bit values, in this case double precision values. This is most likely the result of the reduction of address calculation and other extra code. In general, there is a relatively large difference between no optimization and the first level, but little difference between the next two.

The results of the register usage profile are show in Figure 4. Unlike value type, in most cases there is very little difference between optimization levels. Most of the operands use the frame pointer register. This is the register used for temporary values and local variables. For compress

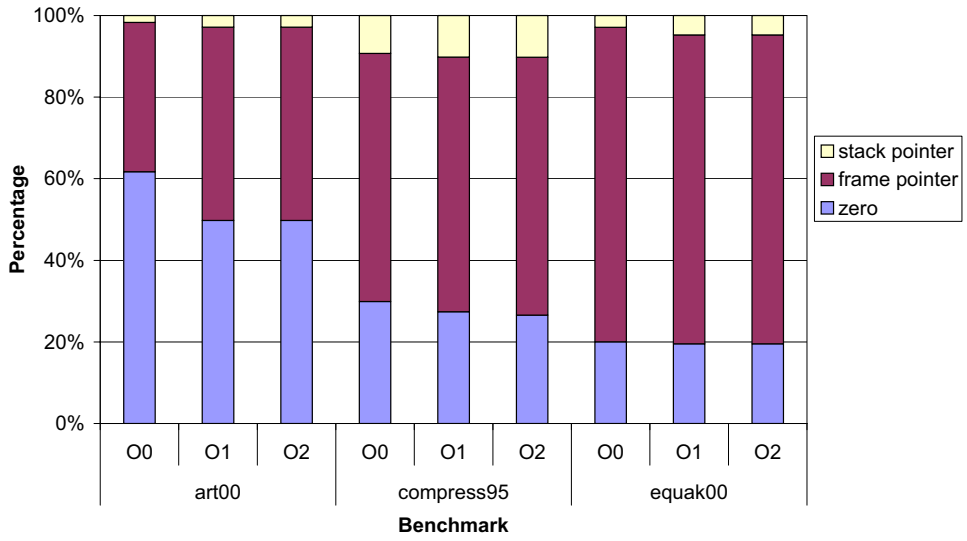

Figure 4: Dynamic register usage. Breakdown of the operand register usage. The results are given for all three benchmarks at all three optimization levels.

over 60% of the instructions use this type, while for equak it is over 75%. The next most often used register is the zero register. In most cases this is used when using a constant, or specifying a label. The label could either be global data or a control flow instruction. It's interesting that art uses such a high percentage of zero register operands, and the marked decrease in usage once optimization is applied. Art seems to make heavy use of global data, and in particular global arrays. The heavy zero register use is probably in address calculation instructions for these globals. The stack pointer isn't used very often. It is primarily specified for parameters passing. It can be used as an indirect indicator of the number of function calls occurring.

The results of the indirection level profile are shown in Figure 5. Like register use, optimization level seems to have very little effect on it. In general one level of indirection is the most common. This is the level used to access local values and parameters. The next most common is no indirection. Its percentage is almost exactly equal to the percentage of zero register usage. This would primarily be used for constants or labels, although it is also used to directly access the registers. If used for constants or labels, the zero register is used; this explains why the two percentages tend to be equal. In general, the direct register access would only be in the function prologue or epilogue. Two levels of indirection is not very common in the compress or equak benchmarks, but is used quite a bit more in the art benchmark. The art benchmark also has a relatively high percentage of no indirection operands. Once again this can be explained by its heavy use of global arrays. The no indirection operands are probably used in address generation. Two level indirection is relatively infrequent in compress and equak. However, it art it comprises nearly 20%. The two levels of indirection are necessary for reading or writing values using the generated addresses for array accesses.

2.3.8 *Comparison vs. PISA*

The same benchmarks were compiled using MIRV for the Portable Instruction Set Architecture(PISA) target and simulated using SimpleScalar. They also used a ported version of newlib. Using the exact same compiler and C library ensures a fair comparison.

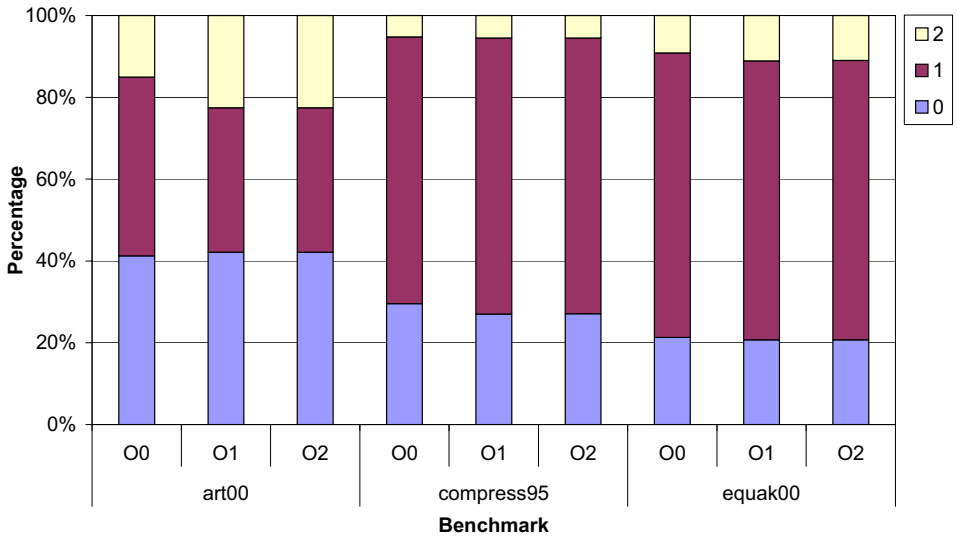

Figure 5: Dynamic indirection level. Breakdown of the operand indirection level. The results are given for all three benchmarks at all three optimization levels.

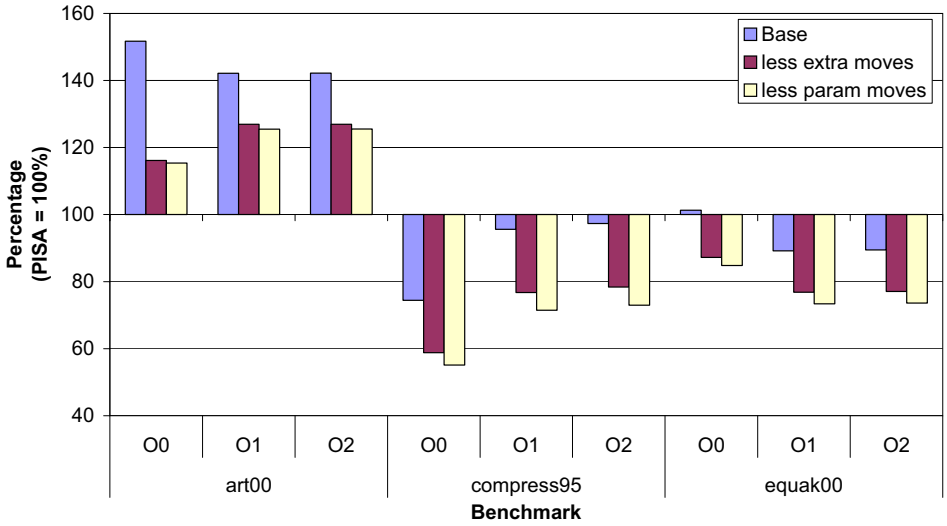

Figure 6: Dynamic instruction count comparison versus PISA. Less extra moves adjusts the number of dynamic instructions by removing those instructions that appear to be extraneous moves caused by the lack of copy propagation. Less param moves includes both the regular extraneous moves and the moves that use the stack pointer (param moves). The param moves are more likely to be actually required.

The benchmarks were run to completion using the simulators and a comparison of the number of executed instructions was made (see Figure 6). The number of instructions for PISA was normalized to 100 and the number executed by the mem-machine was compared to this. The base numbers correspond to the full number of instructions executed. However, the lack of copy propagation leads to extra moves. Any move with a source operand with one or zero levels of indirection and a destination with one level of indirection is a potential extra move. Any operand that

uses the destination of the move as a source could theoretically use the original source of the move. Therefore, this is a rough estimation of the potential number of moves that could be eliminated. However, this would include moves to or from parameters and return values. These are much less likely candidates for elimination. Any moves that involved the stack pointers are probably in the second category. The less extra moves bar adjusts the number of instructions by removing all the potential extra moves. The less param moves bar also removes the parameter move instructions. This gives an approximate indication of the number of instructions once the compiler is more advanced. The number of extra moves is non trivial for all these benchmarks. In most cases it is around 15% of the instructions executed.

The performance of both compress and equak on the mem-machine compared favorably with that of PISA. Compress at no optimization only executed about 75% of the instructions compared to PISA. However, as the optimization level increases, this gap is reduced until at full optimization they are almost the same. It seems that the optimization of the PISA target is more effective on this benchmark then on the mem-machine. If the extra moves are taken into consideration, even at the highest optimization the mem-machine executed about 20% less instructions. Equak has the opposite trend. At low optimization both executed about the same number of instructions. However, as soon as any optimization is done, the mem-machine required about 10% fewer instructions. Obviously some optimizations work better for RISC like targets, while others have more effect on the mem-machine. The performance of art on the mem-machine was very inefficient in comparison to PISA. Even when the potential extra moves are removed, it still required around 25% more instructions. Once again this comes down to the characteristics of the benchmark. In this case, art does quite a bit of access to global arrays. PISA is able to handle these using either displacement or indexed addressing. However, the mem-machine cannot support these directly for an operand. Therefore, it needs to use add instructions to do explicit address calculations. MIRV assumes that these addressing modes are allowed, so the internal assembly is generated to reflect this. The assembly code for any type of array access is optimized to make use of these address modes for efficient code on its usual RISC targets. The mem-machine backend is forced to insert the necessary adds, and it does it on a case by case basis and doesn't do any intra instruction optimization. If the assembly generation could be adjusted to take the lack of these instructions into account from the start, a more efficient scheme could probably be used.

The benchmarks were run over a range of cache sizes. All the caches are direct mapped with a 32 byte line size and only the most optimized code was run. As expected the mem-machine accesses the data cache much more often then the PISA target (see Figure 7). In general it averages between 2 and 2.5 accesses per instruction depending on the benchmark while PISA averages between 0.3 and 0.7. Interestingly the benchmarks display the same relative trend on both machines. This is somewhat unexpected. The mem-machine needs to access memory for all its temporary and local values besides just global data. In particular about the same number of access should be required regardless of whether the data is global or local. I would have expected the average number of accesses to stay relatively consistent.

The most important cache consideration for the mem-machine is how will all the extra accesses affect performance. A simple comparison of miss ratios would not be of any use. The mem-machine has so many more accesses that the miss ratio should be very low. Therefore, a comparison of the absolute number of misses is needed. In particular the number of extra misses that the mem-machine suffers will directly translate into decreased performance. To provide some context, this difference is given as a percentage of executed instructions. Therefore, it becomes the percentage of additional instructions (compared to PISA) that suffer a data cache miss (see

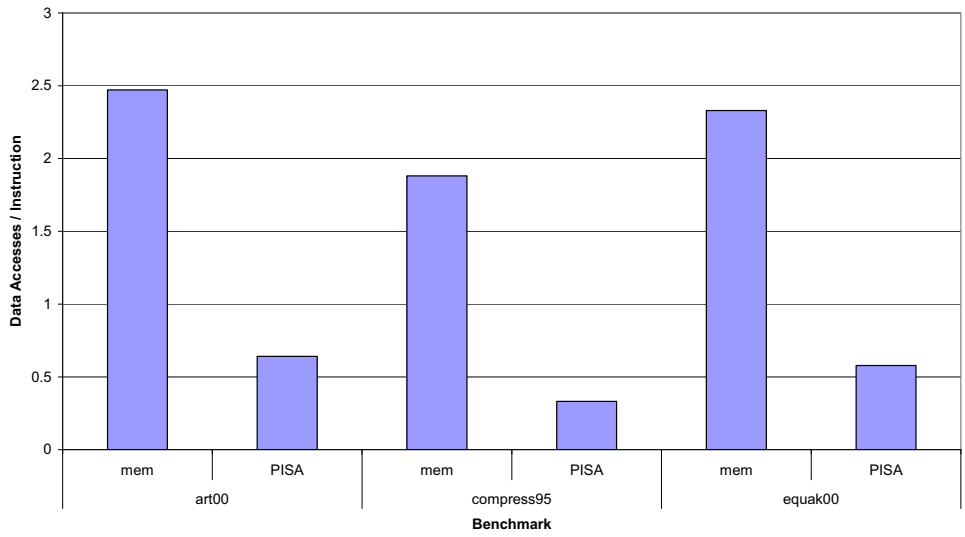

Figure 7: Average data accesses per instruction. The data cache accesses per instruction for the mem-machine and PISA. The results are given for all three benchmarks at all three optimization levels.

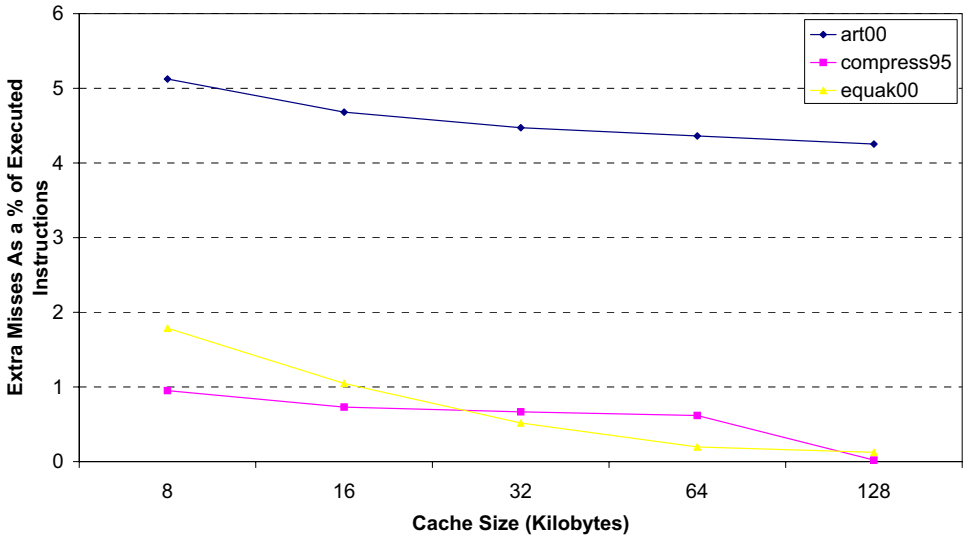

Figure 8: Performance impact of extra data cache misses. The difference in the number of data cache misses between the mem-machine and PISA, normalized to the number of instructions. This gives an indication of the relative performance cost of any extra data cache misses. The results are given for the most optimized compilation. All caches are direct mapped.

Figure 8). For compress this is around 1% for most of the cache sizes and remains relatively stable. Once the cache reaches 128 kilobytes, this percentage drops to almost zero. The sudden drop off seems to indicate this isn't a capacity problem but something else, possibly conflict problems. Equak shows a different trend; the percentage steadily decreases as the cache size increases. This seems to indicate that it's primarily capacity issues for this benchmark. The percentage starts at almost 2% for an 8 K cache, but quickly drops to 1% at 16 K, then 0.5% at 32, and eventually settles around 0.1%. The performance penalty these benchmarks suffer due to the additional data

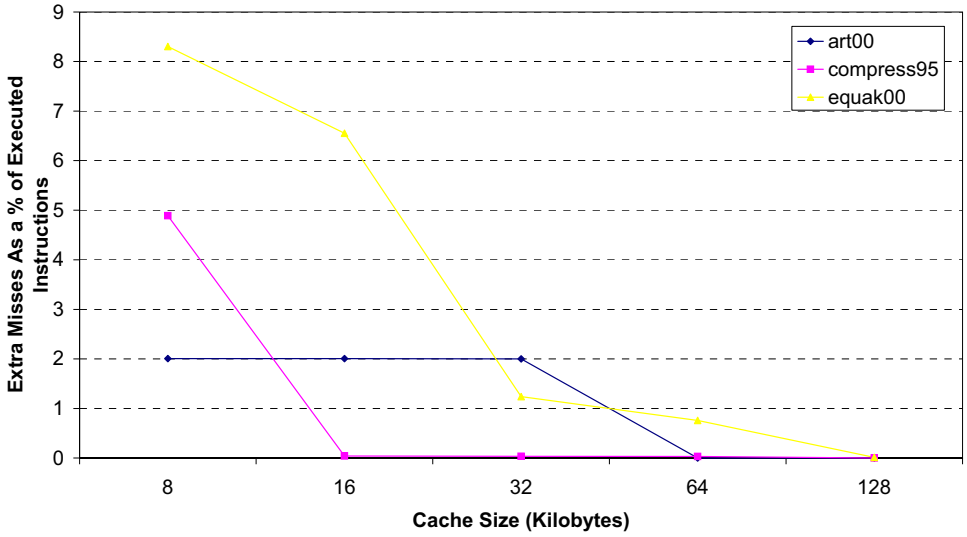

Figure 9: Performance impact of extra instruction cache misses. The difference in the number of instruction cache misses between the mem-machine and PISA, normalized to the number of instructions. This gives an indication of the relative performance cost of any extra instruction cache misses. The results are given for the most optimized compilation. All caches are direct mapped.

cache misses should be relatively low for reasonable size caches. With a 32 K cache, for both benchmarks only about a half a percent more instructions suffer a data cache miss. Depending on the cycle penalty for a cache, this is probably not enough of loss in performance for the PISA target to out perform the mem-machine. The art benchmark has a much larger penalty then the other two benchmarks, even with a 128 K cache, over 4% of the instructions are suffering a cache miss. Although it decreases as the cache sizes increase, it only falls from about 5.1% with an 8 K cache to 4.25% with a 128 K cache. Both PISA and the mem-machine suffer a large number of misses for this benchmark regardless of the size of the cache. However, because the mem-machine makes so many more accesses the number of misses is greater by an almost fixed ratio.

The instruction cache performance is quite a bit different then the data cache (see Figure 9). In this case, the mem-machine is not making more accesses, however each access is for 16 bytes instead of 4. The cache performance is primarily going to depend on how well loop bodies fit into the cache. If the entire body fits in, the cache performance will be good; if it doesn't it will be very bad. For a small instruction cache, the mem-machine suffers quite a larger performance penalty. Eventually when the threshold is reached, and the loops fit in the cache, the penalty drops to almost zero. As expected it takes a cache about four times larger to show the same performance for the mem-machine as PISA. For these benchmarks an 8 K cache is definitely too small to give good performance. At 32 K the performance penalty is less than 2%, and it reaches about zero for a 128 K cache.

Code size is another important consideration for the mem-machine. The instructions for the mem-machine are four times the size of the PISA instructions; however, the mem-machine executables tend to contain fewer instructions (see Table 3). For these benchmarks, the code size for the mem-machine is about 3.5x the size of the code for the PISA executable (see Figure 10). If the possible extra moves are removed, it becomes only about 2.8x the size. Code compression should work well on the mem-machine, because although it has 32 bit offsets and a 32 bit operation, the

	art00	compress95	equak00
PISA instructions	13,644	11,465	15,433
Base instructions	11,753	9,953	13,478
Less extra moves	9,717	8,188	11,070
PISA instruction size	32.00	32.00	32.00
Mem-machine instruction size	128.00	128.00	128.00
Base compressed instruction size	33.33	32.62	37.37
Less moves compressed instruction size	34.13	33.39	38.15

Table 3: Code size statistics. The first three rows show the static number of instructions contained in each benchmark. The last rows show the average instruction size. The mem-machine statistics are shown both for the base and with the potential extra moves removed. The instruction size is shown for both uncompressed and with a simple compression scheme employed.

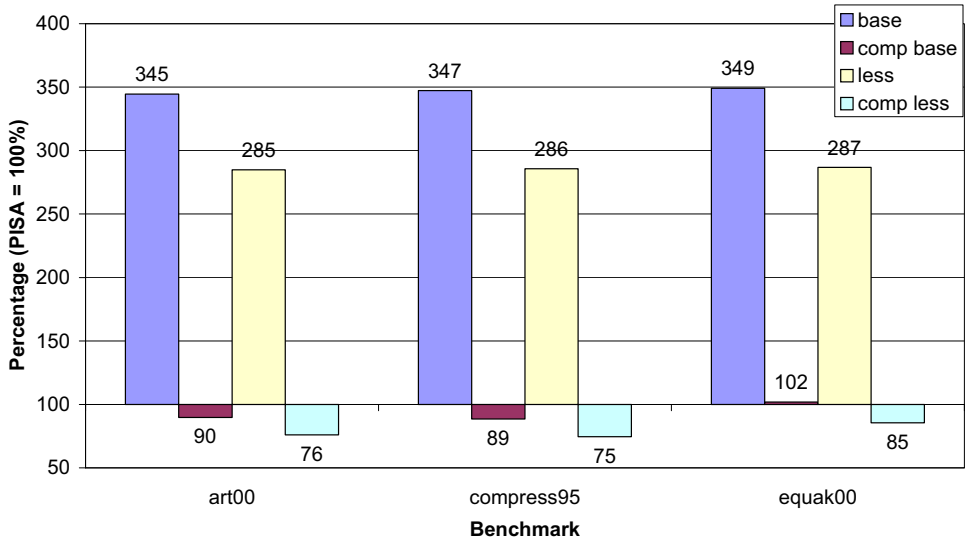

Figure 10: Code size comparison versus PISA. The static code size for all three benchmarks normalized to the size of the PISA code. The results are shown both base and with the extra moves removed (less) and uncompressed and compressed (comp).

number of unique values these have in any executable should be low. In particular, some operations, for example unsigned word move, will be much more common while others are either not possible or not likely. The offset behavior is a bit more varied. Some operations require less than 3 operands, so zero will be a very common value. Standard offsets for locals and parameters will also be common. However, labels (both code and data) will tend to be unique, at least to the executable. A compression scheme could be used to reduce the size. A simple one is to sort the values by order of use and use a bit encoding to encode the most common values. For the mem-machine it makes sense to treat the operation and offset fields separately (see Table 4 for the encodings used). If this simple compression scheme is applied, the base size of the mem-machine code

Operation		Offset	
Size (bits)	Number of Values	Size (bits)	Number of values
4	8	1	1
7	32	7	32
11	256	12	512
36	-	36	-

Table 4: Compression scheme. The simple compression scheme used to compress the mem-machine code. The operation and offset fields are compressed separately. The values are sorted by order of frequency and a simple replacement encoding scheme is used. Four different encodings are used, the first three provide compression. They encode the value in the number of bits specified by the size, but only for the number of unique values given by the second column. The final encoding is used for all the rest of the values.

shrinks from about the same size to about 10% smaller then the PISA. If the extra moves are removed, the code size becomes 15% to 25% smaller than the PISA. The average size of a compressed mem-machine instruction is still a couple of bits larger than a PISA instruction (see Table 3), but because there are fewer instructions the total size is smaller. Notice that the average size of the compressed instruction increases when the extra moves are removed. This is because these moves probably all have the same or at least a very common opcode and they only need two parameters. Therefore, they are probably some of the more compressible instructions.

2.4 Conclusion

The mem-machine exhibited mixed results. Some of the benchmarks were able to show significant performance improvements compared to a traditional RISC ISA. However, the disadvantages far outweighed the advantages. Perhaps the most sobering result is the amount of work necessary to implement a build tool chain for a new instruction set architecture. Compilers especially are complex and difficult to create. Although I was able to modify one to generate code, it could only do so correctly for a small number of benchmarks. The compiler also generated relatively inefficient code. Like most software systems, several assumptions were made when designing the system. Although it is possible to work around these assumptions, the generated code usually suffers.

Although the final ISA had no general registers, compilers usually generate intermediate code that uses an unlimited pool of virtual registers to describe the data dependencies. My modified version simply mapped these virtual registers into stack space offsets. As the compression results show, these offsets can then be compressed into approximately the same size as a register specifier in a RISC ISA. This in combination with the mixed performance results provide some proof that register allocation is not the place to perform the memory mapping of the registers.

Perhaps the greatest complexity in the mem-machine would designing an implementation. A simple single issue in order core would be possible. However, it would still require complex hazard detection and a large number of data cache ports. The only realistic out-of-order superscalar implementation would require renaming the memory locations into physical registers (see Section 3 for an example of such a rename). This would be necessary both for dependency tracking and providing a practical data path. In such a system, all zero and one level indirection operands could be directly mapped into physical registers. However, the dynamic nature of a two level indirection operand makes this impossible. Instead, the operand would require the insertion of an

additional micro op. Such a micro op would be responsible for generating the address (the first level of indirection). This is reminiscent of how loads and stores work in conventional RISC microprocessors. This relationship is further strengthened by the fact that two level indirection operands occur at a rate similar to loads and stores.

3 ARCHITECTURE

3.1 Theory

Our design builds on that of a typical superscalar out-of-order processor [31]. Instructions are dynamically issued from an instruction queue out of program order, but commit in program order using a reorder buffer. To eliminate false dependencies and buffer speculative values, register renaming is used to dynamically assign logical registers to physical registers. We assume a merged register file implementation, like the ones used by the MIPS R10000 and Alpha 21264 [36, 13]. This implementation maintains both the speculative and architectural state of the registers in the physical register file with the reorder buffer containing just the physical register tags.

The virtual context architecture maps logical registers to memory locations using a base address for each context. Depending on the usage model, a register context can be as specific as a particular activation record within a particular thread. Adding the scaled logical register index to the base address produces a memory address that is unique across all contexts. The physical register file caches register values based on memory addresses. As a result, a physical register can hold a value from any context at any point in time.

To avoid modifying the scheduling and execution stages of the pipeline, we guarantee that the physical register indices used by instructions in the instruction queue are valid. Instructions issuing from the queue simply read their operands from (and write their results to) the physical register file without any tag checking or potential miss condition. Instruction scheduling is also performed based on physical register indices as in a conventional processor.

To guarantee valid physical registers for each instruction in the queue, the caching behavior of the physical register file is managed as part of the register renaming process. If a logical register does not have a corresponding physical register, the rename stage allocates a physical register and initiates a *fill* operation to bring the register value in from memory. If necessary, the rename logic will first *spill* an existing physical register's value back to memory so that the register can be reallocated.

Unlike the two states of a typical physical register (free and not free), each physical register is considered to be in one of three states: unavailable, available, and free. Registers with outstanding references from instructions in the queue are *unavailable* for reuse. Registers that contain valid state but do not have outstanding references are considered *available*. Available registers are additionally considered dirty if they have been modified since the last time they were spilled. Finally, Registers without valid state (e.g., because their logical register was overwritten by a later instruction) are *free*. Free registers are allocated in preference to available registers. If there are no free registers, an available register is used. If the available register is dirty, it is first spilled. If all registers are unavailable, rename is stalled until a register becomes available or free. Note that the physical register file contains all the operands and destination registers needed by instructions past the rename stage, so instructions will continue to execute, eventually making registers available and preventing deadlock. The new operation of the rename process is shown in flowcharts a and b in Figure 11.

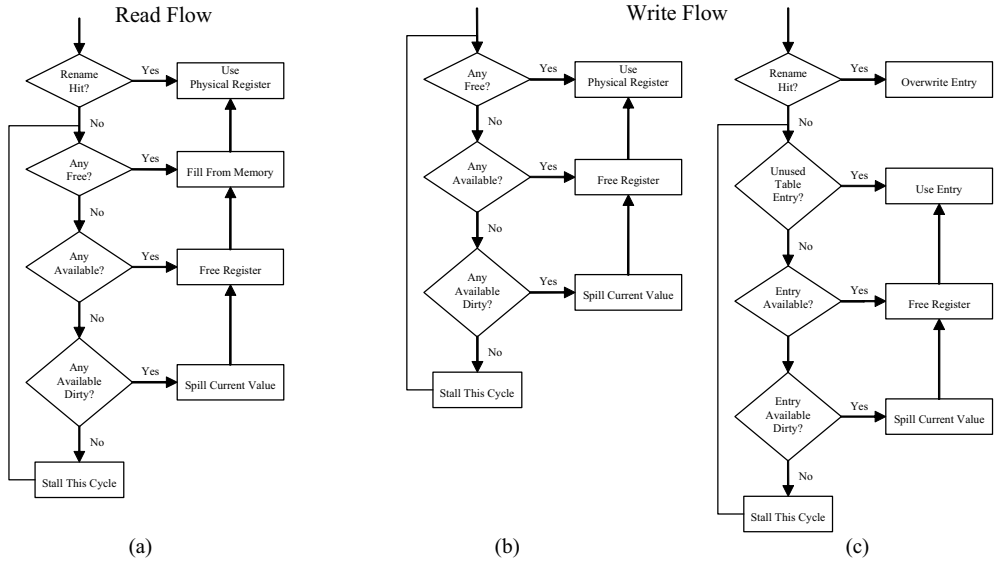

Figure 11: Rename Table Operation. (a) Translating a source logical register address into a physical register. (b) Translating a destination logical register address into a physical register. (c) Allocating an entry for a destination register.

3.2 Implementation

Building a machine based on the virtual context architecture requires some modifications to the traditional pipeline. These modifications can be grouped into three categories: rename, spill/fill implementation, and branch recovery. Most changes focus on the rename stage, including extensive changes to the rename table itself. The pipeline requires minor modifications to allow register spilling and filling to/from memory. Lastly, the increased size of the rename table influences the choice of branch recovery scheme.

3.2.1 *Rename Stage*

There are two main modifications that are needed to the rename stage. First, the rename table is indexed by memory address instead of logical register index requiring modifications to the table itself. Second, the pipeline must track the additional states of the physical registers.

Since the rename table is no longer indexed by logical register, and is instead indexed by a memory address, the rename table can no longer be sized to handle all possibilities. This requires the addition of tags and valid bits to the table. To minimize conflicts and eliminate potential deadlock (a result of a windowed register operand and non windowed register operand of a single instruction mapping to the same set), a set associative table with a small number of ways is used. The rename logic also requires changes to reflect the fact that the rename table is now a limited resource and therefore, must implement a replacement policy for entries. Flowchart c in Figure 11 shows the logic the rename table implements to allocate an entry.

The addition of full memory address tags to the table may be prohibitively expensive. However, the rename table will only contain a small number of addresses with very high locality (a few function contexts from a small number of threads). A simple address translation scheme from a full virtual address into something we call a register space address can be used to significantly reduce the size of these tags (see Figure 12). A translation table that supports a small number of base addresses is added. Each base address covers a large number of function contexts. The upper

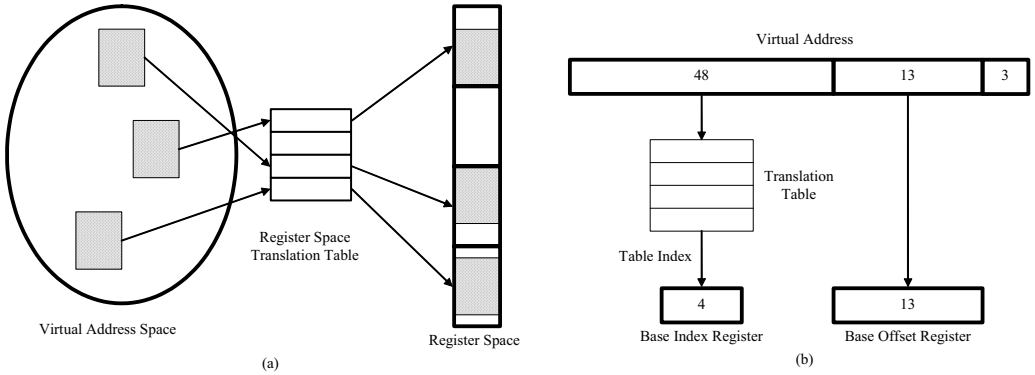

Figure 12: Address Translation. (a) The rename table will contain addresses from a small number of locations corresponding to the few active thread contexts for the small number of threads. A simple translation table can be used to map the large virtual addresses into the more compact register address space. (b) A 64 bit virtual address can be translated into a 17bit register space address. The upper 48 bits of the virtual address are translated into a base index using a small 16 entry table. The next higher 13 bits become the base offset. This is a large enough offset to 182 register windows. The lower 3 bits are discarded because all the registers are 64 bit.

bits from the virtual address are translated into a base index using the translation table. The rest of the virtual address bits become the base offset. The base index and base offset are kept in pipeline registers at the rename stage. A pair of these registers (one for windowed registers and one for global registers) are required for each thread supported by the pipeline.

The combination of the base index and the base offset becomes the base register address of the current context, reducing the required tag length significantly. The address of a logical register is formed by adding the base offset and the logical register offset together, and concatenating the result to the base index. The lower bits from this address are used to index into the rename table, while the upper bits become the tag. A call or return instruction simply increments/decrements the base offset register by a constant amount. If an overflow or underflow occurs the base index is used to index into the translation table and recover the upper bits of the virtual address. The upper bits are then incremented or decremented by one and retranslated to create the new base index.

The translation table only needs to be accessed on base offset overflow or underflow, or on a context switch. For replacement purposes, a small counter is kept for each entry that tracks the number of physical registers mapped using each entry. In the unlikely event that an entry needs to be released but still has physical registers mapped, a special instruction can be used to flush the physical registers to the data cache. The same instruction is used if a physical page that contains register values needs to be paged out by the virtual memory system.

The second modification is that the pipeline must be able to track the additional states of each of the physical registers. A traditional pipeline only tracks if a physical register is free or not free. The physical registers in the VCA can be in three states (free, available, not available). Therefore, the pipeline needs to track availability. A physical register is available when two conditions are met. First, the physical register must contain architectural state, i.e. a committed value. Second, there can be no instructions in the pipeline after the rename stage that are waiting to read the value.

The first condition can simply be tracked with a commit vector, with one bit for each physical register. When an instruction is committed, the bit corresponding to the destination physical register is set. For the second condition, the pipeline must keep track of which registers are sourced by

instructions in the reorder buffer. As in previous work [21,1], this is accomplished by a set of counters for each physical register that tracks the number of instructions after rename in the pipeline that use the physical register. The counters are incremented in the rename stage, and decremented at commit.

3.2.2 *Implementing Register Spills and Fills*

To facilitate register spills and fills, a small queue—the architectural state transfer queue (ASTQ)—is added to the pipeline. On one end, the rename stage adds an entry to the queue when a spill or fill is necessary. On the other end, the ASTQ feeds a mux that merges cache accesses coming from the ASTQ and the instruction queue. If a memory function unit does not issue an instruction during a cycle, the next entry from the ASTQ is issued. A conventional tag broadcast is used to signal dependent instructions for register fills. In the case of spills, the rename stage is stalled until the spill has issued.

In order for the rename stage to add a spill entry to the ASTQ, it requires the address of a physical register to spill it. A table is needed to store these mappings. The table requires one entry for each physical register and contains the memory address tag. At commit, the table is updated for each destination register. The table is only read when a register is spilled. It requires as many write ports as the commit bandwidth of the pipeline, but only a single read port since the rename stage stalls whenever a spill is generated. A fill or spill requires the reverse translation of the register space tag back into a full memory address. This can be efficiently implemented by using the base index portion of the tag to index into the translation table to recover the upper bits of the virtual address, and combine that with the rest of the tag.

3.2.3 *Branch Recovery*

Branch recovery is a problem that all speculative pipelines face. Commonly, architectures checkpoint the rename table at each branch. Our rename table is larger than a conventional one, so this solution would be expensive in area. Another recently proposed solution, that is used in the Pentium 4 process, is a retirement map table [1]. This solution is more practical in the VCA because it requires only a single duplicate of the rename table. This duplicate rename table is kept in the commit stage. As each instruction is committed, it updates this retirement map table. When a mispredicted branch is committed, the retirement map table is now the correct rename table. To recover, the retirement rename table is copied to the rename table, and the pipeline is flushed. A simple optimization is to detect the misprediction at writeback time. The retirement map table is copied immediately to the rename table. The ROB is then walked from the oldest instruction backwards until the branch is reached, and each entry updates the rename table as if it was just renamed.

This branch recovery scheme integrates well with the counter scheme used to track the use of a physical register mentioned in Section 3.2.1. When the retirement map is copied to the rename table, all the use counters are reset to zero. If the optimized recovery is used, the counters are updated as the ROB is walked.

3.3 SMT Implications

To support multiple threads in the pipeline typically means that many structures in the pipeline must be duplicated. In the case of the virtual context architecture, most of the additional structures are completely independent of the number of threads, including the mapping table and use counters. Unlike a traditional pipeline, the rename table itself is not replicated. In fact, the only structures replicated are the pipeline registers holding the current base index and base offset.

Register	Description	Windowed
$0	Function Return	no
$1 - $8	Temp, caller saved	yes
$9 - $14	Temp, callee saved	yes
$15	Frame pointer	yes
$16 - $21	Function Arguments	no
$22 - $25	Temp, caller saved	yes
$26	Return Address	yes
$27	Procedure Value	no
$28	AT	yes

Register	Description	Windowed
$29	Global Pointer	no
$30	Stack Pointer	no
$31	Zero	-
$f0	Function Return	no
$f1 - $f9	Temp, callee saved	yes
$f10 - $f15	Temp, caller saved	yes
$f16 - $f21	Function Arguments	no
$f22 - $f30	Temp, caller saved	yes
$f31	Zero	-

Table 5: Register window ABI. The base Alpha application binary interface was modified to include the concept of register windows. Any register used to communicate values across a function (with the exception of the return address register) is non windowed. All other registers are treated as windowed. Each function invocation has access to a complete set of windowed registers.

4 EXPERIMENTAL METHODOLOGY

The baseline instruction set architecture chosen was the Alpha [30]. To support register windows, several modifications were made to the ABI. Any register used to communicate values across a function call (in either direction) is treated as non-windowed (mapping address is fixed for a thread). All other registers are treated as windowed (mapping address changes on function calls and returns). The windowed/non-windowed assignments are summarized in Table 5. This scheme has the advantage of maintaining compatibility with pre-existing binaries. For simplicity, the call and return instructions were chosen to allocate and deallocate register windows. The call instruction allocates a new register window for the next instruction, while the return instruction deallocates the register window before it executes. This was done so that no instruction uses registers from more than one set. The GNU compiler suite (gcc 3.3.3) [11] was modified to support the new ABI. The GNU standard C library (glibc 2.3.2) was recompiled to support the new register window ABI.

To evaluate the architecture, the SPEC CPU2000 [14] benchmarks were used (except for the four Fortran 90 benchmarks the GNU compiler suite could not compile). The benchmarks were compiled at -O3 optimization, which includes function inlining. Each benchmark was compiled and linked twice: once with the standard compiler and library, and once with the modified compiler and recompiled library. The reference input sets were used to generate SimPoints [28] for both sets of binaries, each of which was simulated for 100 million instructions with a 5 million instruction warm-up.

Benchmark	Ratio	Benchmark	Ratio
bzip2	0.92	mcf	0.93
crafty	0.93	parser	0.92
eon	0.94	perlbmk	0.85
gap	0.91	twolf	0.99
gcc	0.93	vortex	0.83
gzip	0.96	vpr	0.92

Table 7: Path Length Ratio. The ratio of the number of dynamic instructions required to execute the full benchmark for the register window binaries vs. the number for baseline binaries. For benchmarks with multiple inputs, the ratio given in the table is the average of all the inputs.

The architecture was simulated using the M5 simulator, a detailed execution driven simulator [3]. The simulator was modified to support the virtual context architecture. The processor model was a modern four issue processor described in Table 6 with a separate instruction queue and reorder buffer.

4 wide
IQ: 128
ROB: 192
ASTQ: 16
Rename: 64x4
2 read/write ports
DL1: 64K 4-way 3 cycle hit
IL1: 64K 4-way 1 cycle hit
L2: 1M 4-way 15 cycle hit
Memory: 250 cycle

Table 6: Processor description.

5 REGISTER WINDOW EXPERIMENTS

This section summarizes the register window experiments. The methodology subsection describes the workloads used in the experiments and the statistics measured to evaluate the performance of the VCA. The results subsection contains several studies that explore the virtual context architecture as an implementation of register windows.

5.1 Methodology

Up to 10 SimPoints were generated for each of the 43 benchmark/input set combinations. Separate sets of SimPoints were generated for the baseline binaries and the register window binaries. The results of each input were averaged to calculate the result for a benchmark.

Because the register window binaries eliminate explicit register save and restore instructions, IPCs of the windowed and non-windowed runs cannot be directly compared. For each experiment, we calculated the execution time, number of 1st level data cache accesses, and number of 1st level instruction cache accesses. Execution time was calculated by multiplying the average committed CPI (cycles per instruction) of the benchmark (from detailed simulation of SimPoints) by the number of dynamic instructions needed to execute the complete benchmark (from a fast functional simulation of the entire benchmark). The cache accesses are calculated similarly, by multiplying the rate at which the cache is accessed (measured as average accesses per committed instruction) by the total number of dynamic instructions. Table 7 contains the path length ratio between the register window binaries and the baseline binaries.

The results are reported as the values of the register binaries normalized to the baseline binaries. The baseline runs are done with an infinite number of physical registers, so the processor is

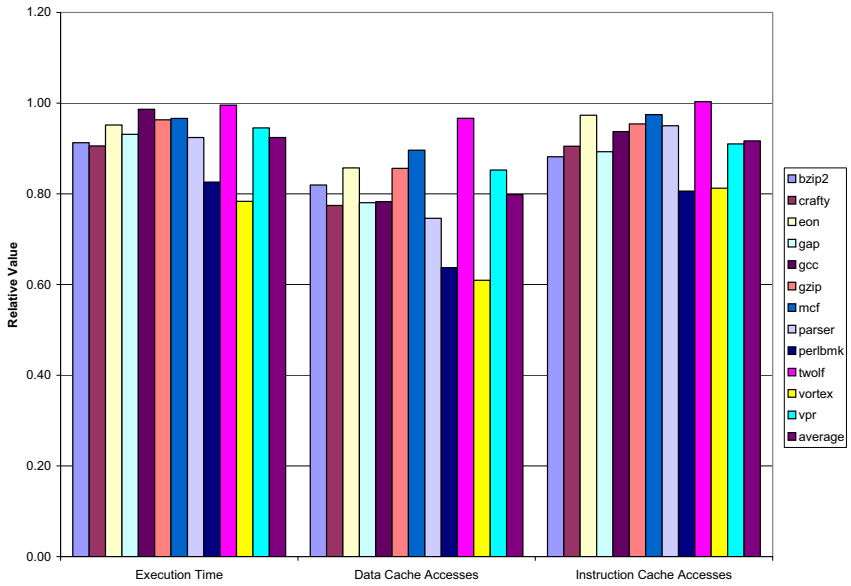

Figure 13: SpecInt Results. The performance of the register window binaries normalized to the baseline binaries. The results use an ideal implementation that never generates any spills or fills.

never forced to stall. The results for the SPEC integer benchmarks using an infinite physical register file for the virtual context architecture are shown in Figure 13. Since no spills or fills are generated, this configuration represents the theoretical limit of performance improvement. The SPEC FP benchmarks were also run, but show very little effect from register windows (a 2% decrease in execution time and 5% decrease in data cache access rate). Therefore, the results only use the SPEC integer benchmarks.

5.2 Register Window Results

This section contains the results of three studies performed to evaluate the performance of the VCA: pipeline, cache and VCA parameters. The pipeline study explores the VCA with four and two issue cores and various physical register file sizes. The cache study tests the performance of the VCA with varying first level data caches. The VCA parameters study looks at the effects the various parameters of the architecture (rename table configuration, ASTQ size) have on its performance.

5.2.1 *Pipeline Study*

Figure 14 shows the results when the number of physical registers is varied. The results show a significant reduction in data cache accesses and a decrease in the execution time of the benchmarks. An infinite number of physical registers shows a decrease in execution time of over 7.5% with a decrease in data cache accesses of 20%. The improvement decreases as the number of physical registers is decreased, and the number of spills and fills generated increases. At 192 physical registers, a reasonable number for this processor, the execution time is still 7% lower than the baseline, with a 19% decrease in data cache accesses. At 128 physical registers, the savings in execution time is reduced to just over 5% and cache access savings is reduced to 13%. The virtual context architecture is able to take advantage of the fact that most if not all of the floating point registers are not being used and can be spilled to memory. This allows it to run efficiently with a very small number of registers. At 64 physical registers there are not even enough registers to keep the entire architectural state. Even with this number of registers, the execution time is still slightly over 1% better. However, the total data cache accesses are now within 1% of the baseline. A decrease in the total instruction cache accesses is achieved for all the configurations because fewer instructions are needed to execute the benchmark. As the number of physical registers decreases, the pipeline slows down and thus does less speculation, which in turn reduces the accesses to the instruction cache.

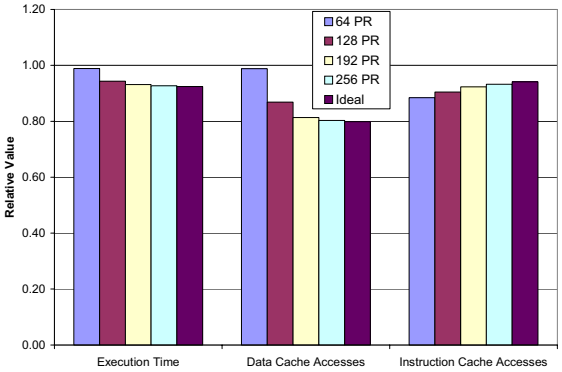

Figure 14: 2 Port Results. The average relative performance of the register window binaries on the VCA with varying numbers of physical registers. The results are normalized to the baseline binaries.

Figure 15 shows the CPI and cache access rates for the same configuration as Figure 14. The CPI of the virtual context architecture is slightly higher than the baseline, but is more than offset by the decrease in dynamic instructions necessary to execute the benchmark. The data cache access rate is less than the baseline for physical register files greater than 64. This reflects the fact that the register window binaries are able to remove much of the save/restore memory traffic. As expected, as the number of physical registers decreases, the data cache access rate increases, until it is 7% greater for 64 registers. The instruction cache access rate is similar to the baseline, with the rate slightly decreasing as the pipeline slows down when there are fewer physical registers.

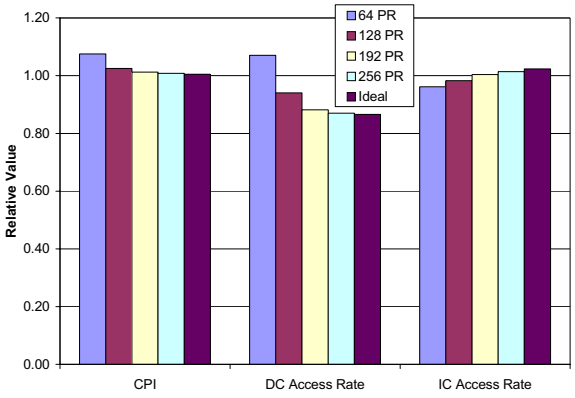

Figure 15: 2 Port Rate Results. The average execution rate and cache access rate of the register window binaries on the VCA with varying numbers of physical registers. The results are normalized to the baseline binaries. The execution rate is measured by cycles per instruction (CPI). The cache access rates are in accesses per committed instruction.

Figure 16 shows the results when the number of data cache read/write ports is reduced to one for both the VCA processor and the baseline. The rest of the processor parameters remain the same as in the two port study. The virtual context architecture is able to take advantage of the decreased memory traffic to show even more improvement in execution time for the larger numbers of physical registers, resulting in a savings of almost 12% at 256 physical registers, decreasing to 10% at 192. For this range of physical registers, the data cache savings are within 2% of the savings with 2 ports (see Figure 14). However, at 128 physical registers, the execution time savings decreases to just 1%, versus almost 6% for 2 ports. The data traffic jumps to within 4% of the baseline. At 64 physical registers, the VCA starts to suffer. The execution time jumps to 13% slower than the baseline and generates 18% more data cache accesses. With fewer physical registers, the VCA is forced to generate more spills and fills. The pipeline only has one cache port to use for fills and therefore instructions began to stall waiting on a fill. This in turn increases the number of instructions resident in the ROB which increases the pressure on the physical register file, leading to even more spills and fills.

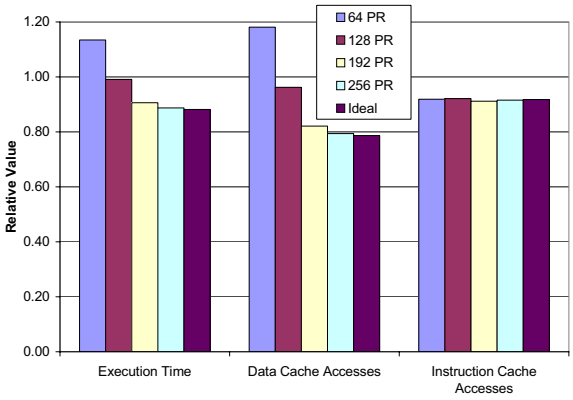

Figure 16: 1 Port Results. The average relative performance of the register window binaries on the VCA with one data cache port. The results are normalized to the baseline binaries run using only one data cache port.

Figure 17 compares the VCA with only one data cache port to the baseline architecture with two ports. With 256 physical registers or more, the virtual context architecture with one cache port is able to outperform the baseline architecture with two by 1%. With 192 physical registers, the VCA is only 1% slower. The virtual context architecture is able to provide comparable performance to the baseline even when it has only half the number of data cache ports. Once the number of physical registers is reduced below 192, the advantages of register windows are eclipsed, with the VCA being 10% and 27% slower with 128 and 64 physical registers.

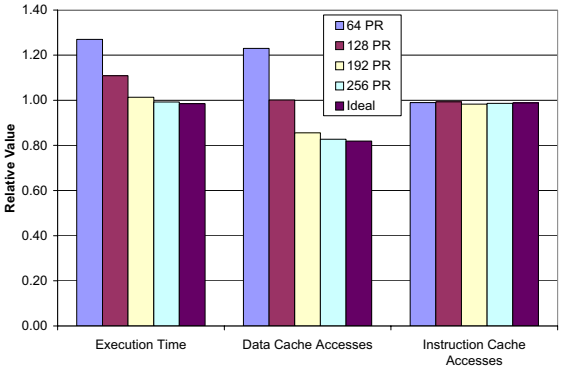

Figure 17: 1 Port VCA Versus 2 Port Baseline Results. The average relative performance of the register window binaries on the VCA with one data cache port. The results are normalized to the baseline binaries run using two data cache ports.

Figure 18 shows the relative performance when the processor is reduced from four issue to two issue. The reorder buffer is decreased to 128 entries and the instruction queue to 64 entries and there is only one data cache port. The results are similar to the four issue results. However, the relative execution time and data cache accesses increase much more slowly when the number of physical registers is decreased. The virtual context architecture is able to outperform the baseline even with only 64 registers, giving a 4% decrease in execution time and 9% decrease in data cache accesses. This shows that the virtual context architecture scales down to less aggressive pipelines. Even with only an issue width of two, the VCA is able to handle spill and fill traffic without slowing the pipeline.

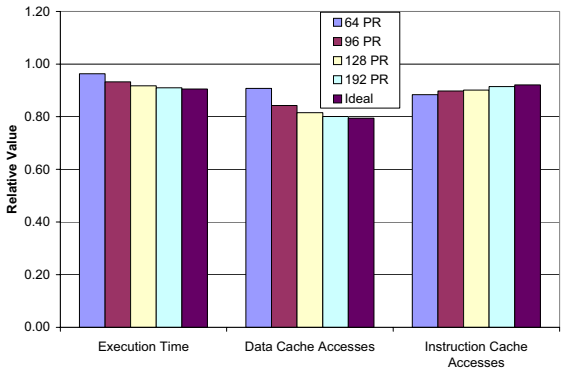

Figure 18: 2 Issue Results. The average relative performance of the register window binaries on a two issue version of the VCA. The results are normalized to the baseline binaries run on a similar two issue baseline architecture.

5.2.2 *Cache Study*

Figures 19and 20 show the relative values as the cache configuration is varied for both the VCA and baseline. Figure 19 is 128 physical registers, while Figure 20 is 192 physical registers. We vary the cache size from 32KB to 64KB, the hit latency from 1 to 3 cycles, and the number of ports from 1 to 2. The number of ports has the most effect on performance. With only one data cache port, the execution rate of the benchmark slows down, increasing the occupancy of the reorder buffer and in turn the number of physical registers needed. Therefore, the virtual context architecture will be forced to generate more spill and fill traffic to accommodate this. For 192 physical registers, there is an increase in data cache accesses of only 1%, however for 128 physical registers the increase is almost 10%. The number of ports has opposite effects on execution time depending on the number of physical registers. For 192 the slight increase in data cache traffic is still much lower than the baseline leading to a relative decrease in execution time for the register window binaries. With 128 registers, the extra cache accesses are enough to increase the relative execution time of the VCA. The other cache parameters only have a minor influence on the relative performance. As the results show the virtual context architecture provides good performance irrespective of the cache configuration.

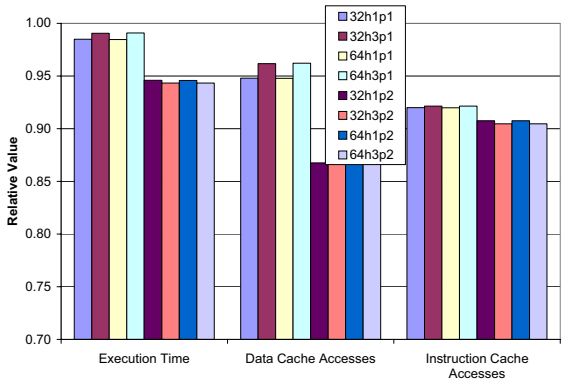

Figure 19: Cache Study for 128 Physical Registers. The relative performance of the register window binaries on the VCA with 128 physical registers and the given cache configuration, normalized to the baseline binaries on the baseline machine with the same cache configuration. The cache configuration is given in the form ShLpN, where S is the size of the cache in kilobytes, L is the hit latency in cycles and N is the number of ports.

The virtual context architecture does have a slight adverse effect on the level two cache. However, it is very slight and on average less than 2% more accesses with any of the configurations compared to the ideal.

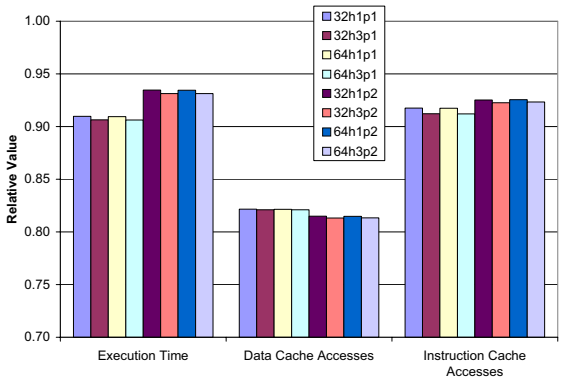

Figure 20: Cache Study for 192 Physical Registers. This is the same test as Figure 19, except the VCA has 192 physical registers.

5.2.3 *Virtual Context Architecture Parameters Study*

This study considers the performance implications of the rename table and ASTQ. The size of the rename table has little effect on the performance of the VCA. A 64 by 4 way table is within 0.01% of a fully associative table with one entry for each physical register. The rename table should be sized according to the number of physical registers. A good rule of thumb seems to be one set for each logical register and enough associativity to have the total entries approximately equal to the size of the physical register file. Smaller tables can be used, but will cost a few percent in performance. For example, a 64 by 2 way table decreases the execution time by over 1%.

We also ran several different experiments using various ASTQ sizes. Doubling the size of the queue has no appreciable advantage, while halving it to 8 entries shows a small decrease in performance. Removing the queue completely (forcing the frontend to stall on any spill or fill) also provides good performance. In the future, we will perform a more detailed study to determine the optimum size of the ASTQ.

6 SMT EXPERIMENTS

This section summarizes the SMT experiments. The methodology subsection describes how the workloads used in the experiments were generated and the statistics measured to evaluate the performance of the VCA. The results subsection contains the results of two studies. In the first study, the baseline binaries were run on the virtual context architecture with varying numbers of physical registers. The second study combines an SMT processor with register window binaries to show the full potential of this new architecture.

6.1 Methodology

The best single SimPoint was generated for each of the 43 benchmark/input combinations for the baseline binaries. A single input was selected for each benchmark. In the case of benchmarks with more than one input, the input closest to the average IPC of all the inputs was selected. The corresponding SimPoint for the register window binaries was generated by executing the benchmark until the same number of basic blocks were encountered.

We used a scheme similar to that of Raasch [25] to generate representative two and four thread workloads. First, each of the 253 possible two-thread workloads was run using the baseline architecture. Next, we chose a set of statistics. The statistics fall into two groups. The first group characterizes the benchmark's absolute single-thread behavior, and includes data cache accesses, floating point register usage, total number of unique addresses registers are mapped to, ratio of dynamic instruction count between register window binary and baseline binary, and the ratio of data cache accesses between the register window binary and the baseline binary. The second

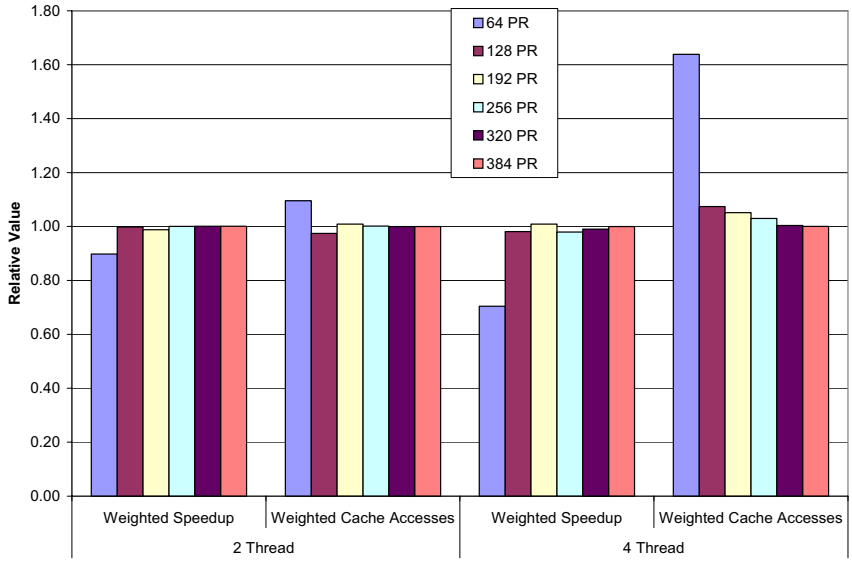

Figure 21: Baseline SMT Results. The relative performance of the workloads composed of baseline binaries run on the VCA with varying numbers of physical registers. The results are normalized to the baseline workloads running on the baseline architecture with enough physical registers so that it never stalls.

group represents the relative performance of a thread in an SMT workload, and is composed of committed IPC, fetch rate, issue rate, branch predictor accuracy, instruction queue occupancy, instruction queue ready rate, reorder buffer occupancy, data cache miss rate and second level cache miss rate. These statistics are normalized to the same benchmark's single thread values.

The statistics are all scaled to a mean of zero and variance of one. Principle components analysis [10] is then used to eliminate correlation. Enough components are used to cover more than 99% of the variance (11 for our two-thread workloads). The workloads are then mapped into 22-dimensional space and a linkage clustering algorithm is used to cluster the workloads. The Bayesian Information Criterion (BIC) score is calculated for each cluster assignment. The smallest number of clusters that has a BIC score within 2% of the maximum BIC score is used. This resulted in 49 two thread workloads.

The process with a few small changes was repeated to generate the four thread workloads. The full combination of four thread workloads is too large to run. Instead, the initial set of workloads was the 1201 unique combinations of the two thread workloads. The same statistics were used, and principle components generated. In this case, the first 10 principle components were used. Finally, the workloads were mapped into 40-dimensional space and the clustering algorithm run. This resulted in 164 four thread workloads.

Each workload was warmed up until one thread reached 5 million instructions, then run until all the threads committed 100 million instructions, or one thread committed 200 million instructions. Two statistics are used to measure performance. The first is weighted speedup [27,32]. This metric is calculated by summing the speedups of all threads—the IPC of each thread in the SMT workload, divided by the IPC of the same benchmark running as a single thread. The second statistic is weighted cache accesses. It is calculated similar to weighted speedup, but using data cache accesses per instruction instead of IPC.

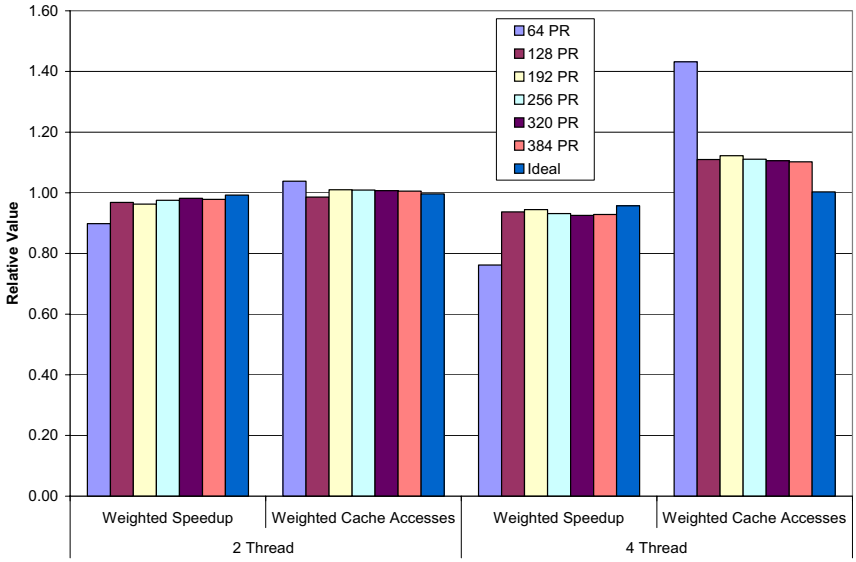

Figure 22: Register Window SMT Results. The relative performance of the workloads composed of register window binaries run on the VCA with varying numbers of physical registers. The speedups are computed using the register window binaries run on the VCA with the same number of physical registers. The results are normalized to the baseline workloads running on the baseline architecture with enough physical registers so that it never stalls.

6.2 SMT Results

6.2.1 *Baseline Binaries*

Figure 21shows the baseline binaries executing on the virtual context architecture. With two threads, the virtual context architecture is able to efficiently execute the workloads with as few as 128 physical registers. This is true even though the architectural state of the two threads is just over 128 registers. This demonstrates the much greater register file efficiency the VCA can achieve. For four threads, our scheme is able to provide good performance (within 2%) even with as few as 128 physical registers. This is a register file half the size needed to contain the architectural state. This comes at a cost of only 7% more cache accesses. With only 64 physical registers, the performance for 2 threads is 10% lower with 10% more data cache accesses and the 4 thread results are even worse with 30% decrease in performance and almost 65% more data cache accesses.

6.2.2 *SMT Register Windows*

Figure 22 shows the register window binaries on the VCA. The values are normalized to the single thread workloads running with the same number of physical registers. Unlike the baseline binaries, the register window binaries are able to take advantage of the extra registers because of the larger number of unique registers. Therefore, the addition of extra threads using the physical register files has a more profound effect. However, the virtual context architecture is still able to support simultaneous register window threads with almost the same speedup as the baseline architecture, even with fewer physical registers than logical registers. With two threads, the weighted speedup drops immediately to 2-3% less than that achieved on the baseline architecture with the

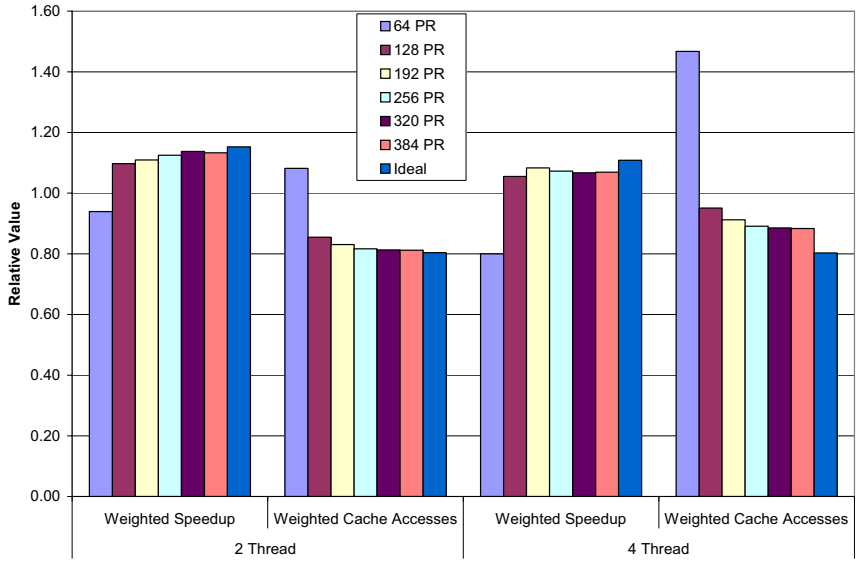

Figure 23: Adjusted Register Window SMT Results. The relative performance of the workloads composed of register window binaries run on the VCA with varying numbers of physical registers. The speedups are computed using the baseline binaries run on the baseline architecture. The speedups are then adjusted based on the ratio of the dynamic instruction count between the register window binary and the baseline binary. The results are normalized to the baseline workloads running on the baseline architecture with enough physical registers so that it never stalls.

baseline binaries. This drop remains relatively constant until the register file only has 64 registers, then it drops to over 10%. This is paralleled in the cache overhead which remains at about 1% until 64 registers, then it increases to 4%. The pipeline is able to accommodate the extra traffic, and therefore the speedup remains close. The four thread workloads show a similar response, although the scale is different. Instead of the 3% drop, there is an immediate 7-8% drop. This remains relatively steady, until 64 registers, and then it drops to 24%. The cache overhead once again shows a similar trend with a 10% overhead until 64 registers where it jumps to nearly 45%.

Figure 23 shows the statistics when the weighted values are calculated against the baseline single thread binaries, and adjusted to reflect the difference in dynamic instruction count of the register window binaries. This is measuring the full advantage of the register window binaries versus the baseline binaries. It incorporates not only the reduced memory traffic of the register windows, but also their shorter dynamic instruction count. The two thread workloads show very promising results. With 128 physical registers or greater, they achieve a speedup approximately 10% greater than the baseline. This is achieved in concert with a 15% to 20% reduction in cache accesses. As seen in the previous chart, the four thread workloads have a much greater cache overhead, and this is reflected in these results. Because each instruction is actually a greater percentage of the execution, the four thread workloads are still able to show an improvement in performance with around a 6-8% advantage for a register file greater than 128. The cache access reduction is much smaller with a 12% reduction for 384 physical registers to a 5% reduction for 128. As expected, with only 64 registers neither the two or four thread workloads are able to achieve near the performance of the baseline. When the virtual context architecture is provided

with a reasonable number of physical registers, it is able to achieve a 10% speedup over a conventional SMT machine while making fewer cache accesses. This is true even when the size of the physical register file is too small to hold the logical registers of all the threads (something a conventional SMT core cannot have).

7 RELATED WORK

This section reports on a number of previous studies that implemented efficient large physical register files, provided more aggressive physical register management, or sought to reduce the memory traffic caused by the save/restore of registers.

The idea of using memory to provide a backing store to the register file has been explored before. Ditzel et al. [9] proposed the C machine stack cache. In this machine, the register file is replaced by a large circular buffer that is mapped contiguously onto the stack. The VCA maps each register individually, thus using registers more efficiently and adding support for multiple threads, at the cost of more complex mapping hardware. Huguet and Lang [15] use a table of memory addresses and compiler support to allow the hardware to perform background saves/restores. Their architecture allows the saving and restoring to be demand driven, but does not support more than one instance of a logical register. Nuth et al. [22] proposed the Named State Register File, which like the VCA treats the physical register file as a cache and memory maps the logical registers. Unlike our design, the Named State Register File requires a CAM of the entire register file on every access, which could potentially increase the cycle time of the processor. Both the Huguet and Nuth architectures are based on simpler in-order processors, and misses are not detected until an access is made. While this is not a problem in in-order cores, it is a problem in modern out-of-order superscalar designs. With these architectures, the register access time is no longer constant, requiring the addition of complex control logic to the already performance-critical schedule/execute/writeback loop. The VCA does not require any complex changes to these stages.

Register caches [2, 24] and banking [2, 8] have been proposed to improve the access latency of large register files. In these designs, the full architectural state is still kept in the register file; thus die area continues to limit the number of supported contexts. In contrast, the VCA is able to provide the appearance of an almost infinite number of logical registers. These techniques (in particular banking) could seamlessly be integrated into our architecture if desired.

Monreal et al. [20] and Yung et al. [37] have also proposed architectures that attempt to use the physical register file more efficiently. Monreal's virtual-physical register scheme delays the allocation of a physical register until writeback. It greatly decreases the lifetime of a physical registers and allows better performance with the same sized register file. Yung observed that a lot of the values sourced by instructions are read from the bypass network and the rest could be supplied by a small cache. In both cases, they seek to get the same performance out of a much smaller register file, but don't increase the number of logical registers that the architecture can handle. Their work is orthogonal to ours and could be combined with the virtual context ideas we propose.

One of the major costs of SMT is the larger physical register file needed to accommodate additional architectural state. Earlier proposals sought to reduce this burden by dividing the logical registers among multiple threads [34, 26]. These designs require extensive compiler and operating system support. Our design requires no compiler support for SMT and is backward compatible with legacy binaries.

Martin et al. [19] and Lo et al. [17] propose software identification of dead register values to reduce save/restore traffic and to free physical registers for other SMT threads, respectively. Our

design targets similar goals, and also achieves them in part by moving dead values out of the register file into memory. Our scheme has two advantages: it does not require software annotations, and it also seamlessly manages the efficient saving and restoring of values that are not dead but have not been accessed recently (e.g., live values from calling procedures or from active but stalled threads). The proposed dead-value annotations would be a useful addition that should integrate easily into our design, allowing us to avoid spilling dead values to memory and to reclaim dead registers preferentially over live but inactive ones. We hope to explore this extension in future work.

Two commercial architectures that use register windows [29, 23, 9] are the SPARC [35] and Itanium [7]. SPARC processors use a software approach, trapping to the operating system on an underflow or overflow condition. The Itanium uses a hardware register stack engine (RSE). When an overflow or underflow occurs, the processor halts the execution of the program and the RSE proceeds to move registers to or from memory. In either case, the handling of an underflow or overflow halts the execution of the program and adds a significant amount of overhead. The VCA is able to provide a nearly optimal implementation of register windows by spilling/filling on single-register granularity instead of entire register windows. It also does not require the entire register window to be resident in the physical registers, and thus uses the register file much more efficiently. Spilling and filling are handled in the front end of the pipeline, allowing the back end to continue execution in parallel. An out-of-order core enables the processor to tolerate the extra latency of infrequent spills/fills just as it enables conventional systems to tolerate memory latencies.

Previous work has also focused on reducing the memory bandwidth of stack accesses [6, 16]. These works split out all or part of the stack references and direct them to a separate pipeline and cache. Both implementations reduce the bandwidth to the first level data cache, but only by diverting demand to a separate cache. Programs still require the same number of cache accesses and dynamic instructions. The VCA actually eliminates cache accesses and reduces the number of instructions required to execute the program. Both architectures could be combined with the VCA. However, because of the reduction in accesses in the redirected stream, their effectiveness would be greatly diminished.

8 CONCLUSIONS

The virtual context architecture uses a novel mapping scheme to effectively remove the physical register storage requirement of logical registers. This enables the physical register file to hold just the most active subset of logical register values by allowing it to spill and fill registers on demand in hardware rather than in software. The removal of the logical register storage requirement allows architects to choose a physical register file size based on performance considerations alone. This mapping scheme requires no changes to the normal physical register file design or the performance-critical schedule/execute/writeback loop and is compatible with most other techniques for reducing the latency of the register file such as banking and caching. The result is an improvement in performance and reduction in the demands on the memory system while using fewer physical registers than a conventional machine.

Because the VCA maps the traditional logical register namespace onto the larger namespace of virtual memory, it can be leveraged to provide a nearly ideal implementation of register windows by allowing multiple activation records to exist in the physical register file at any given time. Our implementation in an out of order processor shows a decrease in execution time of 8% while reducing data cache accesses by nearly 20% using a conventionally sized physical register

file. The performance advantage of the VCA is enough that it can achieve the same performance with only one cache port as an otherwise similar conventional machine would with two cache ports.

The virtual context architecture is also effective at supporting simultaneous multithreading without a significant increase in physical register file size. Traditional SMTs require the architect to increase the size of the physical register file to accommodate the extra logical registers but because the VCA allows some of the logical registers to be stored in the memory system, this requirement is obviated. The VCA also enables register windows to be easily and efficiently implemented on a SMT core. With two thread workloads, the VCA provides a 10% increase in weighted IPC versus the baseline and a decrease of 15% in cache accesses, and can achieve this with fewer physical registers than logical registers. With four threads, we achieve a 10% increase in performance with a 9% decrease in cache accesses, again with fewer physical registers than necessary to hold the architectural state of all threads.

We plan to continue to improve the architecture with compiler enhancements and additionally plan to begin studying the possibilities for the VCA in an operating system environment. On the compiler side, previously proposed dead-value annotations would be a useful addition that integrates easily into our design. These annotations allow us to avoid spilling dead values to memory and reclaim dead registers preferentially over live but inactive ones. On the operating system side, one could leverage the fact that the VCA allows a thread to only have part of its context in the register file at any given time to do very fast interrupt handling and cheap context switching. If an interrupt comes into the machine, it can immediately start processing that interrupt in an empty context without having to spill all of the registers of the previously active process by allowing the hardware to spill registers only as necessary. An interrupt handler would then only impact performance of the system proportionally to its size. Similarly, a context switch can be achieved by simply changing the PC and base address registers of a process, allowing the hardware to take care of spilling and filling registers.

Overall, the VCA provides much more flexible and efficient usage of register file resources, a critical component of modern microarchitectures while further providing exciting possibilities for future work.

9 FUTURE WORK

There are three general areas for future work. The first area is improving the virtual context architecture implementation. The second area is exploring new uses for the VCA. Finally, there are some potential alternate research areas which are only slightly related to the VCA.

9.1 Improving VCA Implementation

There are two potential areas where the VCA implementation could be improved. The first is to reduce the amount of spill/fill traffic. The second is to simplify or minimize the cost of the implementation.

9.1.1 *Reducing Spill/Fill Traffic*

Spill/Fill traffic can be reduced in one of two ways. The first is to minimize the number of live logical registers. The second way is to try to limit the amount of unnecessary spills that are generated.

Minimizing the logical register usage involves changing the register allocation algorithm used by the compiler. The register allocation scheme should be modified to take the VCA requirements into mind. Specifically, it should try to allocate the minimum number of logical registers for each

function. It could also be modified to minimize the time between the last use of a register and it's subsequent redefinition.

Spills caused by the presence of dead value are unnecessary and result in extra memory traffic. The presence of this values also occupy space that could potentially be used by values that will be accesses later. Thus, removing this values can also decrease fill traffic. These values can be identified by the compiler and/or hardware. Martin et al. [19] and Lo et al. [17] propose software identification of dead register values. The same techniques they use to identify these values and communicate the information to the hardware could be used by the VCA. This would allow the pipeline to free the physical register associated with a dead value as soon as possible, reducing the pressure on the physical register file.

A purely architectural approach is also possible. When a register window is deallocated, all the registers in that window are no longer needed (the argument registers should also no longer be valid, they are caller saved). One possible way to track this is to maintain a circular buffer of bit vectors with one bit for each physical register. A current index is kept specifying which vector is currently active. As an instruction is committed, if the destination register is windowed, it sets the bit in the current vector. Thus, the vector specifies which physical registers are assigned to windowed registers in the current window. When the instruction deallocating the window is committed, the bit vector is ored with the free list vector to make the new free list (all windowed registers from the deallocated window immediately become free). The vector is then cleared. When an instruction allocating a new window is committed, the current index is incremented and the new current bit vector is cleared. Clearing the bit vector on allocations and deallocations naturally handles the circular nature of the buffer and the fact that it's a finite resource. In this way, with n bit vectors, it is possible to maintain the information for the last n windows allocated.

9.1.2 *Minimizing Implementation Cost*

The three main costs associated with the VCA are the rename table, ASTQ and physical register file.

The rename table is modified to be set associative and requires tags. As mentioned in Section 3.2.1the tags can be optimized to reduce their size. The table itself is still large, and the set associative nature complicates the rename logic. The access pattern of the table is not random. The table will generally be accessed using memory addresses from the same window. It seems like some type of caching scheme could be used to minimize the cost of the average type of access. Although a penalty may result when transitioning from one window to the next, the potential savings may be worth it.

The ASTQ itself is not a large structure. However, initial studies on it's size suggest that completely removing the queue and just stalling on a spill or fill may provide adequate performance. This possibility requires further study. The existence of the queue itself (even if a single entry in the stall case) does require the insertion of a priority mux in the issue logic. Another potential implementation is to have the rename stage insert micro ops into the pipeline to handle the spills and fills. While this would eliminate the mux, it would put more pressure on the reorder buffer and instruction queue. At this point, it is unclear what the possible performance cost of this would be.

Finally, although the VCA does not require any changes to the physical register file itself, it does tend to put more pressure on it. This would merit investigating some previously studied techniques on optimizing the physical register file itself. In particular, banking and caching can be easily incorporated into the VCA design. Previous work has also focussed on more aggressively

managing physical registers [1, 20, 37]. This work should be compatible with the VCA and allow a more aggressive reclamation of physical registers (freeing them).

9.2 Exploring New Uses

The VCM has three main advantages over a traditional out of order architecture. First, it completely decouples the physical registers from the logical registers. This allows the compiler and operating system to work with an unlimited set of logical registers and allow the architecture to handle them. Second, it allows for fast context switching. The operating system is only responsible for changing the base pointer, the architecture will as needed move architectural state into and out of the physical registers. Third, the architecture can directly map addresses into physical registers.

9.2.3 *Decoupling The Physical Registers From The Logical Registers*

Decoupling the physical registers from the logical registers can have several potential applications. The normal limitations on the number of active thread contexts is relaxed. This allows the architecture to naturally support multiple contexts simultaneously. Within one thread, each function activation record can be given its own context (register windows). Also, multiple threads can have active contexts in the processor at once (SMT). The initial results in these areas are promising. However, a more aggressive approach is possible in both areas.

Register windows are used to reduce some of the memory traffic imposed by register management. By modifying the compiler to mark loads/stores used for this management, the remaining overhead could be measured. If significant, a new ABI could be developed to try to alleviate these costs. Each function could be allowed to allocate a larger pool of registers and his pool could be used to hold all the local variables, removing any necessary spill or fill traffic. Instead, the memory traffic would solely be dependent on the size of the physical register file. A simple move instruction could be introduced to move a value from the pool into the standard logical registers or from a logical register back into the pool. A similar move could also be used to allow a function to access the registers in the register window of the calling function. This would remove the need for non windowed registers.

By decoupling the number of logical registers and physical registers, the cost of supporting additional thread contexts is greatly reduced. This has the potential of allowing large numbers of simultaneous threads. A limit study could be used to determine the performance improvement achieved by adding additional threads. This could be done in both the ideal physical register file case, then repeated with more practical register file sizes. A scheduling algorithm could also be introduced. The processor could support a very large number of active threads, but an internal hardware scheduling algorithm could be introduced to schedule them. The algorithm would have to take into account both resource utilization, and the cost of register spilling and filling. The study may reveal how close to 100% resource utilization it is possible to get.

9.2.4 *Fast Context Switching*

Fast context switching also has several potential benefits. The performance/simplicity of an operating system can be enhanced to allowing the hardware to manage some of the context switch overhead. Very efficient interrupts are possible by using a separate context to handle the interrupts. In fact, completely different contexts could be used for kernel code, simplifying it and enhancing security. This has the potential for allowing efficient microkernels and virtual machines.

The first step in doing any experiments on this would be to modify the Linux kernel to be VCA aware. This could be a potentially large undertaking depending on how thoroughly the kernel code is going to be optimized. Once the OS was running, a thorough study of context switch and interrupt overhead could be performed. Perhaps the most challenging part of this research would be finding a virtual machine or microkernel that I could use to do research on. Ideally, it would have to be open source and relatively easy to work with. Initial research on the internet have not yielded many promising possibilities.

9.2.5 *Directly Mapping Addresses To Physical Registers*

Directly mapping addresses to physical registers also has several potential applications. In particular, it has the potential of allowing very efficient emulation of other ISAs. This would allow for an efficient implementation of a stack based architecture (Java) without placing any limits on the size of the stack. Systems with vastly different logical register requirements (x86 and Itanium) could coexist on the same system. The backend of the pipeline would implement a microop architecture with enough flexibility to support both systems. Separate decodes could supply the backend of the pipeline with microops that use addresses for sources and destinations. The VCM rename stage and backend could then efficiently implement this. While a potentially very valuable capability, any testing would involve implementing at least one more instruction set (ideally a non RISC one) in M5.

REFERENCES

[1] Haitham Akkary, Ravi Rajwar, and Srikanth T. Srinivasan. Checkpoint processing and recovery: Towards scalable large instruction window processors. In *36th Ann. Int'l Symp. on Microarchitecture*, pages 423–434, December 2003.

[2] Rajeev Balasubramonian, Sandhya Dwarkadas, and David Albonesi. Reducing the complexity of the register file in dynamic superscalar processors. In *34th Ann. Int'l Symp. on Microarchitecture*, pages 237–248, December 2001.

[3] Nathan L. Binkert, Erik G. Hallnor, and Steven K. Reinhardt. Network-oriented full-system simulation using M5. In *Proc. Sixth Workshop on Computer Architecture Evaluation using Commercial Workloads*, February 2003.

[4] Doug Burger, Todd M. Austin, and Steve Bennett. Evaluating future microprocessors: the SimpleScalar tool set. Technical Report 1308, Computer Sciences Department, University of Wisconsin–Madison, July 1996.

[5] Steve Chamberlain, Roland Pesch, Jeff Johnston, and Red Hat Support. The red hat newlib c library, July 2002. Red Hat, Inc.

[6] Sangyeun Cho, Pen-Chung Yew, and Gyungho Lee. Decoupling local variable accesses in a wide-issue superscalar processor. In *Proc. 26th Ann. Int'l Symp. on Computer Architecture*, pages 100–110, May 1999.

[7] Intel Corporation. *Intel IA-64 Architecture Software Developer's Manual*. Santa Clara, CA, 2000.

[8] José-Lorenzo Cruz, Antonio González, Mateo Valero, and Nigel P. Topham. Multiple-banked register file architectures. In *Proc. 27th Ann. Int'l Symp. on Computer Architecture*, pages 316–325, June 2000.

[9] David R. Ditzel and H. R. McLellan. Register allocation for free: The C machine stack cache. In *Proc. Symp. on Architectural Support for Programming Languages and Operating Systems*, pages 48–56, March 1982.

[10] Lieven Eeckhout, Hans Vandierendonck, and Koen De Bosschere. Quantifying the impact of input data sets on program behavior and its applications. *Journal of Instruction-Level Parallelism*, 5, February 2003.

[11] Free Software Foundation. GNU Compiler Collection. http://gcc.gnu.org.

[12] David Anthony Greene. *Design, Implementation, and use of an Experimental Compiler for Computer Architecture Research*. PhD thesis, April 2003.

[13] Linley Gwennap. Digital 21264 sets new standard. *Microprocessor Report*, 10(14):11–16, Oct. 28, 1996.

[14] John L. Henning. SPEC CPU2000: Measuring CPU performance in the new millennium. *IEEE Computer*, 33(7):28–35, July 2000.

[15] Miquel Huguet and Tomás Lang. Architectural support for reduced register saving/restoring in single-window register files. *ACM Trans. Computer Systems*, 9(1):66–97, February 1991.

[16] Hsien-Hsin S. Lee, Mikhail Smelyanskiy, Gary S. Tyson, and Chris J. Newburn. Stack value file: Custom microarchitecture for the stack. In *Proc. 7th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 5–14, January 2001.

[17] J. L. Lo, S. S. Parekh, S. J. Eggers, H. M. Levy, and D. M. Tullsen. Software-directed register deallocation for simultaneous multithreaded processors. *IEEE Trans. Parallel and Distributed Systems*, 10(9):922–933, September 1999.

[18] Deborah T. Marr, Frank Binns, David L. Hill, Glenn Hinton, David A. Koufaty, J. Alan Miller, and Michael Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, 6(1), February 2002.

[19] Milo M. Martin, Amir Roth, and Charles N. Fischer. Exploiting dead value information. In *30th Ann. Int'l Symp. on Microarchitecture*, pages 125–135, December 1997.

[20] Teresa Monreal, Antonio Gonzlez, Mateo Valero, Jos Gonzlez, and Victor Vinals. Delaying physical register allocation through virtual-physical registers. In *32nd Ann. Int'l Symp. on Microarchitecture*, pages 186–192, November 1999.

[21] M. Moudgill, K. Pingali, and S. Vassiliadis. Register renaming and dynamic speculation: an alternative approach. In *Proceedings of MICRO-26*, 1993.

[22] Peter R. Nuth and William J. Dally. The named-state register file: Implementation and performance. In *Proc. 1st Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 4–13, January 1995.

[23] David A. Patterson and Carlo H. Sequin. RISC I: A reduced instruction set VLSI computer. In *Proc. 8th Intl. Symp. Computer Architecture*, volume 32, pages 443–457, Nov. 1981.

[24] Matt Postiff, David Greene, Steven Raasch, and Trevor N. Mudge. Integrating superscalar processor components to implement register caching. In *Proc. 2001 Int'l Conf. on Supercomputing*, pages 348–357, 2001.

[25] Steven E. Raasch and Steven K. Reinhardt. The impact of resource partitioning on smt processors. In *Proc. 12th Ann. Int'l Conf. on Parallel Architectures and Compilation Techniques*, September 2003.

[26] Joshua A. Redstone, Susan J. Eggers, and Henry M. Levy. Mini-threads: Increasing tlp on small-scale smt processors. In *Proc. 9th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 19–30, February 2003.

[27] Yiannakis Sazeides and Toni Juan. How to compare the performance of two SMT microarchitectures. In *Proc. 2001 IEEE Int'l Symp. on Performance Analysis of Systems and Software*, November 2001.

[28] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *Proc. Tenth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*, pages 45–57, October 2002.

[29] Richard L. Sites. How to use 1000 registers. In *Caltech Conference on VLSI*, pages 527–532. Caltech Computer Science Dept., 1979.

[30] Richard L. Sites, editor. *Alpha Architecture Reference Manual*. Digital Press, 3 edition, 1998.

[31] J. E. Smith and G. S. Sohi. The microarchitecture of superscalar processors. In *Proc. IEEE*, volume 83, pages 1609–1624, December 1995.

[32] Allan Snavely and Dean M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading processor. In *Proc. Ninth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*, pages 234–244, November 2000.

[33] Dean Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proc. 23rd Ann. Int'l Symp. on Computer Architecture*, pages 191–202, May 1996.

[34] Carl A. Waldspurger and William E. Weihl. Register relocation: Flexible contexts for multithreading. In *Proc. 20th Ann. Int'l Symp. on Computer Architecture*, pages 120–130, May 1993.

[35] David L. Weaver and Tom Germond, editors. *SPARC Architecture Manual (Version 9)*. PTR Prentice Hall, 1994.

[36] Kenneth C. Yeager. The MIPS R10000 superscalar microprocessor. *IEEE Micro*, 16(2):28–40, April 1996.

[37] Robert Yung and Neil C. Wilhelm. Caching processor general registers. In *Proc. 1995 Int'l Conf. on Computer Design*, pages 307–312, October 1995.

APPENDIX A: MEM-MACHINE SUPPORTED OPERATIONS

jal - jump and link
jail - jump and link indirect
b(cond)* - conditional branch
j - jump
ji - jump indirect
omov - offset move
mov - move
cvt - convert type
add - add
sub - subtract
mul - multiply
div - divide
mod - modulus
sll - shift left logical
srl - shift right logical
sla - shift left arithmetic
sra - shift right arithmetic
neg - negation
and - logical AND
or - logical OR
xor - logical XOR
comp - logical complement
s(cond)* - set conditional

*(cond) : eq(=), ne(!=), lt(<), le(<=), gt(>), ge(>=)

APPENDIX B: MEM-MACHINE ASSEMBLY LANGUAGE

C code:

```
#include <stdio.h>

int main()
{
int i;
  for (i=0;i<10;++i) {
    printf("test = %d\n",i);
  }
  return 0;
}
```

Assembly code:

```
.data
        .align 2
$__mirv_pack.m1.200:
        .ascii "test = %d\n\0"

.text
.ent main
.globl main
main:
$L46:
        // Prologue sizes local=12 call=16 frame size=32
        !mov.uw -4(%sp), %fp
        !mov.uw %fp, %sp
        !sub.uw %sp, %sp, #32
$L47:
        !mov.uw -12(%fp), #0
$L51:
        !bge.w -12(%fp), #10, #$L50
$L49:
        !mov.uw 8(%sp), #$__mirv_pack.m1.200
        !mov.uw 12(%sp), -12(%fp)
        !jal 0(%sp), #printf
        !add.w -12(%fp), -12(%fp), #1
        !j #$L51
$L50:
        !mov.uw 4(%fp), #0
        !j #$L48
$L48:
        !mov.uw %sp, %fp
        !mov.uw %fp, -4(%sp)
        !ji 0(%sp)
.end main
```

‘#’ - 0 level
‘ ‘ - 1 level
‘*’ - 2 level

Indirection Level

%0 - zero register
%sp - stack pointer
%fp - frame pointer

Register Specifier

s - single precision float
d - double precision float
b - byte
h - half word
w - word
l - long word
ub - unsigned byte
uh - unsigned half word
uw - unsigned word
ul - unsigned long word

Type Specifier