A Flexible Compilation Infrastructure for VLIW and SIMD Architectures

by

David Z. Maze

B.S., Electrical Engineering and Computer Science,
Massachusetts Institute of Technology (2000)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2001

Author ....
Department of Electrical Engineering and Computer Science
August 31, 2001

Certified by ....
Saman P. Amarasinghe
Associate Professor
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

A Flexible Compilation Infrastructure for VLIW and SIMD Architectures

by

David Z. Maze

Submitted to the Department of Electrical Engineering and Computer Science
on August 31, 2001, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

Most compiler infrastructures target existing physical architectures. This creates difficulty when trying to perform architectural research: trying to simulate the effects of adding registers to a processor might involve making changes deep inside the compiler infrastructure and writing a dedicated simulator. Making higher-level changes, such as adding a functional unit or extending data paths to handle multiple words in parallel, may be nearly impossible.

This thesis describes a compiler back-end and simulator code to compile and simulate code for a variety of architectures. In-order, single instruction/multiple data (SIMD) superword architectures, both clustered and unclustered very long instruction word (VLIW) architectures, and SIMD/VLIW hybrid architectures can all be targeted. A single control file specifies these high-level architectural details, along with characteristics such as the number of registers and the number of functional units, the operations they can execute, and the latency of these operations.

There are several characteristics of this system that make it more suitable for architecture research than existing systems. Only having to change a single file to change the target architecture makes it possible to compare several distinct architectures directly, as is demonstrated here. Since this work is based on an existing infrastructure, it can take advantage of existing optimization passes and components such as a register allocator. Finally, since the machine models used here are not based on a physical processor, the user has some flexibility to add to or modify the instruction set to simulate additional instructions.

Thesis Supervisor: Saman P. Amarasinghe
Title: Associate Professor

Acknowledgments

Several people deserve thanks for their help with this thesis. Sam Larsen wrote the SUIF 1 front-end pass for detecting superword-level parallelism. Michael Gordon wrote large parts of the simulator code, and Diego Puppin was responsible for the scheduler. Mark Stephenson also provided substantial support in coding and debugging the system. Andromeda Yelton proofread late revisions of the thesis. Finally, this thesis would not have been possible without the help and support of my advisor, Saman Amarasinghe.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Traditional computer architectures have presented a machine model in which one instruction completely executes before the next instruction begins. While advances such as pipelining and superscalar architectures have allowed multiple instructions to be execute in parallel, hardware has been added to maintain the illusion of a single instruction executing to the programmer.

In hopes of reducing the complexity introduced by this hardware, other architectural options have been explored, including very long instruction word (VLIW) processors[5] and the addition of single instruction, multiple data (SIMD) instructions to conventional processors. Both of these options allow the compiler to explicitly specify when multiple instructions should be executed in parallel, removing this burden from the processor itself. These options can also be combined with existing architectures; for example, extensions such as Intel's MMX[13] and Motorola's AltiVec[6] add SIMD operations to the Pentium and PowerPC processors, respectively, and it is possible to extend most standard processor designs into superword machines by widening data paths and duplicating functional units.

Since SIMD and VLIW architectures require explicit scheduling of parallel instructions, good compiler support is required to target these sorts of architectures. However, most research compiler back-ends only target machines whose assembly code mimics an in-order machine. The Machine SUIF system[14] used here has this limitation, for example. Other infrastructures target a specific parallel machine (in the case of Trimaran[1], a virtual machine similar to Intel's IA64 architecture), but this specific targeting makes it difficult to compile for a different style of machine.

Other architectural constraints make it difficult to use the same compiler infrastructure to compile and simulate code for different architectures. While the basic tool chain in every case is similar, extra passes are needed to extract superword-level parallelism[10], or to schedule code for a particular clustered VLIW architecture[11]. Other modifications, such as changing the size of the register file or the number or latency of functional units, are desirable for research but are often difficult to perform without making fundamental changes to the compiler.

1.1 Overview

This thesis introduces extensions to the Machine SUIF back-end[14] to target a variety of architectures, including in-order, VLIW, and SIMD architectures. Details of the architecture, including the size of the register files, the general type of architecture, and the number and latency of included functional units, are specified in a control file that is easily changed.

It is also reasonably straightforward to change the instruction set by modifying the source to the back-end and the simulator. The new opcode needs to be added to the list of opcodes in the source and to a small number of other files, and simulator code for this opcode needs to be added to the simulation printer and the associated library. However, adding instructions does not necessitate changes to other components such as the scheduler, register allocator, or linker.

The code produced by the back-end is of consistent quality in spite of the system's flexibility. The register allocator and calling convention easily accommodate changes in the number of registers; the scheduler used gives reasonable schedules for in-order, unclustered, and clustered VLIW machines, while leaving SIMD information intact. A peep-hole optimizer can improve the generated code, sometimes substantially, on any targeted architecture.

1.2 Related Work

Several other systems are available for architectural research; however, they tend to be lacking in one way or another. There are a fair number of systems to generate parts of the tool chain from an architectural description, but these are not complete systems as they lack integration with a front-end or a simulator.

Trimaran[1] is a research compiler focused on EPIC architectures. Its primary target is a simulated machine called HPL-PD[9], though there is a complicated means for specifying other architectures. Trimaran's main disadvantages are its close ties to HPL-PD and the opacity of the infrastructure. While HPL-PD accommodates VLIW and in-order machines readily, it is difficult to extend the system for SIMD. The available Trimaran documentation also encourages exclusive use of a graphical front-end, and neglects to mention any inner workings of the system.

SimpleScalar[4] is a popular simulation environment for architectural research; its infrastructure has been modified extensively to perform power simulations and more detailed simulations of what happens inside the chip. SimpleScalar's main target is an in-order MIPS-like machine, using a modified version of the GNU C compiler and tool chain. It supports both in-order and out-of-order execution, the latter via an artificial "register update unit." SimpleScalar has a detailed simulator which accounts for caching, branch prediction, and other particulars of the architecture. While SimpleScalar supports modifying the instruction set and machine characteristics, adding VLIW or SIMD code to the system would be difficult at best; these changes would involve making modifications to the GNU compiler and assembler, which one SimpleScalar presentation describes as "very painful"[3].

The Aviv retargetable code generator[7] is built upon the Stanford SUIF infrastructure[15], and targets a variety of embedded systems, primarily VLIW DSPs. Aviv uses an instruction set description language (ISDL) to describe the processor target, and automatically generate a back-end and simulator. For this work, the ISDL is too powerful and flexible: while it is readily possible to simulate any real VLIW processor, it is difficult to work with multiple machine models in parallel. The Aviv system creates separate tool chain and simulator for each target architecture, hiding the details of what actually happens in the back-end. It is also unclear how Aviv would be extended to support SIMD operations.

1.3 Organization

This thesis explores extensions to the Machine SUIF back-end to target both VLIW and SIMD architectures. In Chapter 2, a general machine model capable of being an in-order machine, a simple or clustered VLIW machine, or any of these machines with SIMD extensions is described. Chapter 3 describes a back-end for Machine SUIF that targets this machine model. Chapter 4 discusses the implementation of the simulator and how assembly is converted into a simulated binary. The process for compiling and running simulated code is described in Chapter 5. Finally, results from compiling a suite of test programs comparing the different architectures are presented in Chapter 6.

Chapter 2

Machine Model

This chapter describes the machine model this system targets. The target machine is not tied to any physical hardware; as such, information such as the number of registers in the register file, the latency of the attached functional units, and extensions such as SIMD operations may be specified at compile time. However, the instruction set is heavily influenced by that of real RISC machines, most notably the MIPS processor.

The instruction set is, for the most part, fairly typical of RISC machines, though some additions have been made to accommodate VLIW and SIMD architectures. The instruction set is described in detail in Appendix A.

2.1 Basic Model

The basic model emulates a simple microprocessor. A maximum of one instruction is issued per cycle, though the processor may stall if scheduling constraints are not met. Instructions may begin executing before previous instructions have committed their results to the register file, but only if they do not depend on the results of these instructions.

The processor has three separate register files. The *general-purpose* or *integer* register file holds integer values; the *floating-point* register file holds floating-point values; and the *predicate* register file holds single-bit flags. Most instructions read and write to either the general-purpose or floating-point register file, though every instruction reads a single predicate register.

For simplicity, the processor has a limited number of addressing modes. Most instructions use values in registers directly; only the `LD` and `ST` instructions, their floating-point variants, and the `LDA` (load-address) instruction use memory addresses. Memory addresses are always expressed as an offset relative to a register, such as `4(r1)` to refer to the fourth byte above register `r1`. Many ALU operations can also use a 16-bit signed constant, which must be the second source operand. A complete listing of opcodes and allowed addressing for each is included in Appendix A.

2.1.1 Memory

All addresses in the machine model are 32 bits wide, and it is assumed that any general-purpose register can hold any memory address. This restriction is necessary to simplify the generated code. Program code is not represented as being stored in memory, since the bit-pattern of instructions is not specified. The simulator is responsible for setting up the exact memory environment. Hooks are provided to make it possible for the simulator to simulate an instruction cache, though the simulator currently lacks such a cache.

The current simulator implementation uses a 256 MB address space; this is a compromise between using all of the available memory on the host machine and accommodating the working set of benchmarks. The first kilobyte of memory is reserved; reading from or writing to there is an error, intended to protect against dereferencing null pointers. The next section of memory contains static data, with initial values provided by the compiler. Dynamically allocated memory begins after the static data area and grows upward in memory. The stack begins at the top of memory and grows downwards. This memory layout is illustrated in Figure 2-1.

2.1.2 Calling Convention

All procedure calls use a standard calling convention. This requires cooperation on the part of both the calling procedure and the procedure being called, and establishes

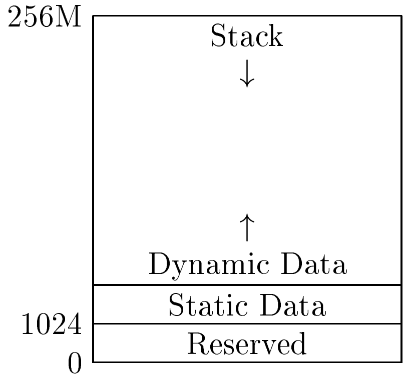

Figure 2-1: Memory Layout

standards for the use of registers and stack space.

With the exception of a small number of reserved registers, all registers are available for general use. The caller is responsible for saving live registers with an odd number, such as `r11` or `f7`. The callee is responsible for saving any register it uses with an even number, such as `r10` or `f8`. This even/odd convention allows for a balance between caller- and callee-saved registers, and works even when the actual number of registers is unknown. The callee is made responsible for saving all of the predicate registers, since there is no way to save individual predicates.

Appendix B contains a complete description of the calling convention.

2.1.3 Predication

Instructions on the simulated machine are *predicated*[12]. Every instruction causes reads from memory or a register file, along with any stalls this may require. Each instruction also reads a single predicate register; if the value of that predicate register is a boolean false, then the result is not written back to the register file or to memory. A complete description of the effects of predication on instructions is given in Table 2.1.

Predication is appearing in a number of real processors, including Intel's IA64 architecture[8]. An effective predicator in the compiler can remove branches in the generated code, which require extra effort for branch predication and prefetching

Instruction	Effects of false predicate
Loads and stores	Cache effects still take place, but no value is actually read or written
Jumps	Jump not taken; destination register (if any) unchanged
Predicate-setting	Destination registers set to false
Inter-cluster sends	No effect; value is still sent
All others	Result not written to destination register

Table 2.1: Effects of Predication

instructions. Adding predication to this architecture allows experimentation with different predication algorithms. In the absence of explicit predication annotations, the compiler infrastructure will add in predicates only as required for conditional branches.

2.1.4 Scheduling

On each cycle, an instruction is read from a virtual memory store. The sources of the instruction are read from the register file, and the result of the instruction is calculated and stored in the target register. Two different numbers determine the amount of simulated time this takes: no other operations may occur until the *delay* of the instruction has passed, and an instruction depending on the result of the calculation will stall until the *latency* of the instruction has passed.

These two numbers are sufficient to model the delays present in most functional units. For example, the user might request a machine without a fully pipelined multiply unit. The delay of multiply instructions would be two cycles; no instruction could be issued in the cycle after a multiply. Additionally, multiplies would have a latency of four cycles, compared with one cycle for simple integer ALU operations. If the user wanted a fully pipelined multiplier instead, the delay for the multiply instruction could easily be changed from two cycles to one by changing the listed timing constraints for the MUL instruction.

To see the effects of scheduling constraints, consider the following code sequence, which calculates $(a * b + c) + (d * e)$:

MUL r10, *a*, *b*
ADD r11, r10, *c*
MUL r12, *d*, *e*
ADD r13, r11, r12

Without scheduling, this code sequence would execute in 10 cycles; see Figure 2-2. The first ADD instruction must wait until the first MUL is complete to use its result, and the second ADD is similarly delayed. A scheduler would notice that the second MUL could be moved before the first ADD, since there are no dependencies between the two instructions. This reduces the time required to 7 cycles, as shown in Figure 2-3. In both of these figures, note that nothing can be issued on the second cycle of a multiply, and that instructions depending on the results of a multiply cannot be scheduled sooner than four cycles after the start of the multiply. The empty boxes have no instruction scheduled for them at all; three fewer cycles are thus wasted in the scheduled code sequence.

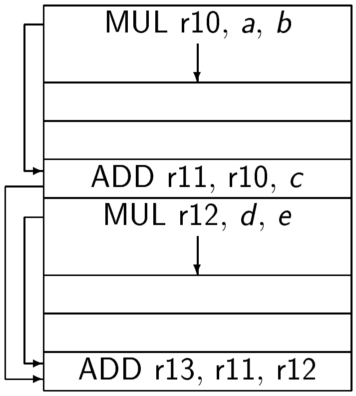

Figure 2-2: Sample Unscheduled Code

2.2 VLIW Machines

This architecture lends itself well to compiling for *very long instruction word* (VLIW) machines. A simple VLIW machine allows the compiler to explicitly schedule more than one operation per long instruction; on a single cycle, the processor might si-

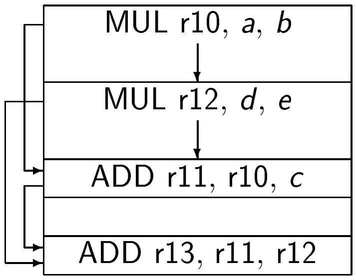

Figure 2-3: Sample Scheduled Code

multaneously schedule a load from memory, an integer addition, and a floating-point multiplication. This style of architecture is very closely related to the *explicitly parallel instruction computing* (EPIC) architecture used in IA64.

A VLIW machine may also be *clustered*. A clustered VLIW machine has some number of identical clusters, each of which contains a register file and some number of functional units. Clustered VLIW architectures are generally intended to increase the number of available functional units without requiring excessive numbers of read and write ports on the register files.

On a clustered VLIW machine, it is often necessary to transfer data between clusters. A pair of instructions, SND and RCV, do this work. The cluster sending a value executes a SND instruction, and the receiving cluster executes a RCV during the same cycle. The transfer mechanism may be implemented as either a simple bus, in which case only one value may be sent per cycle, or as a cross-bar, in which case any number of values may be transferred.

The multiply-and-add code sequence in Section 2.1.4 could be scheduled in six cycles on a sample clustered VLIW machine. This machine might have two clusters with two functional units each. The first functional unit can perform loads, stores, and inter-cluster transfers; the second can perform integer and floating-point arithmetic operations. This scheduling requires that the correct values be in each cluster's register file beforehand. A schedule that accomplishes this is shown in Figure 2-4; the result appears in r13 in the first cluster.

	MUL r10, *a*, *b*
RCV r12, ...	ADD r11, r10, *c*
	ADD r13, r11, r12

	MUL r12, *c*, *d*
SND r12	

Figure 2-4: VLIW Scheduled Code

2.3 SIMD Machines

This architecture can also easily accommodate *single instruction, multiple data* (SIMD) machines, in which each instruction affects multiple data words in parallel. A single "register" in this case contains a *superword*, which in turn contains some number of parallel data *subwords*, typically four. An instruction such as an ADD instruction would result in subword addition in each of four parallel *lanes*, with the results being put in the appropriate slots in the destination registers. Similarly, memory operations are typically wide loads and stores, which load or store four words in parallel to or from sequential memory addresses.

On a SIMD machine, predication is applied to each subword separately. A SLT instruction, for example, does a pairwise comparison of each subword in the source operands and sets each bit in the target predication register accordingly. This turns predicates into masks which specify which subwords in a superword are affected by an operation. A non-superword machine can be viewed as a special case of a SIMD machine with only one lane.

As with the clustered VLIW architecture, SIMD machines require some way of transferring data between separate computational units. There are two specialized instructions used here: PACK moves values from the first subword of separate registers into subwords of a single register, and UNPACK moves values from the subwords of a single register into separate registers. These instructions are necessary to accommodate both scalar and vector operations together.

The code sample being used here is somewhat lacking in superword-level parallelism, but for completeness, Figure 2-5 shows the same multiply-and-add sequence as it might be scheduled for a SIMD machine. The subscripts after the opcodes show which clusters are in use, so an $\mathsf{ADD}_{0,1}$ instruction would add the values in the first and second lanes of the source registers and store the results in the first and second lanes of the target register. For PACK and UNPACK instructions, subscripts after operands indicate which clusters of those operands are used; $\mathsf{PACK}_{0,1}$ z, x_0, y_0 moves the values from the first lane of x and y into the first and second lanes of z.

In the scheduled code, the first two instructions pack the sources of the first two multiplies into registers r14 and r15; the single MUL instruction then executes these two multiplies in parallel. After they complete, the value in the first lane of r17 is added to the value in the first lane of r16, and r0 to the value in the second lane. The results are then unpacked using a UNPACK instruction, and added to get the desired result in the first lane of r13. This code sequence requires nine cycles to execute; longer code sequences would require fewer PACK and UNPACK instructions, reducing the setup requirements.

$\mathsf{PACK}_{0,1}$ r14, a_0, d_0
$\mathsf{PACK}_{0,1}$ r15, b_0, e_0
$\mathsf{MUL}_{0,1}$ r16, r14, r15
$\mathsf{PACK}_{0,1}$ r17, c_0, $\mathsf{r0}_0$
$\mathsf{ADD}_{0,1}$ r18, r16, r17
$\mathsf{UNPACK}_{0,1}$ $\mathsf{r11}_0$, $\mathsf{r12}_0$, r18
ADD_0 r13, r11, r12

Figure 2-5: SIMD Scheduled Code

Chapter 3

Machine SUIF Interface

This chapter describes how the previously described architecture interfaces to the Machine SUIF system[14]. Machine SUIF provides a target-independent infrastructure for reducing a general intermediate format to machine-specific instructions and for performing optimizations on the resulting instruction stream. The passes shown in Figure 3-1 are run as part of the Machine SUIF back-end; the indicated passes use parts of this system, rather than provided Machine SUIF code.

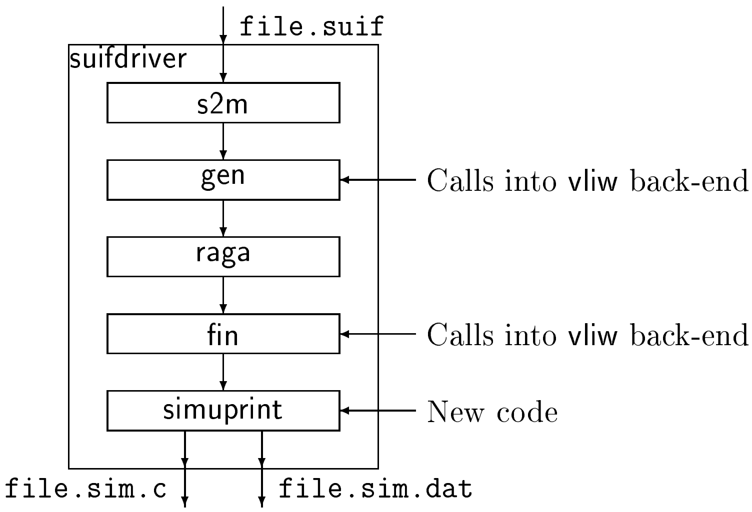

Figure 3-1: Machine SUIF Back-End

There are several reasons for choosing Machine SUIF for this infrastructure. Machine SUIF is already quite flexible; given code from the SUIF 2 front-end, code may be generated for the Intel x86, Alpha, MIPS, and PowerPC architectures using existing back-ends. The Machine SUIF distribution already includes generalized passes for common back-end operations such as register allocation and assembly code printing. Several useful passes, including a pass to detect superword-level parallelism in code[10], already exist in SUIF 1; there is a straightforward path to convert code from SUIF 1 to SUIF 2, and from there to Machine SUIF.

A number of passes are involved in creating an assembly file from a file with the SUIF 2 intermediate format. A pass called s2m converts from the SUIF 2 intermediate format to Machine SUIF, targeting a virtual platform called SUIFvm. The gen pass then converts from SUIFvm to the target assembly, with the name of the target machine taken as a command-line parameter. The named back-end library actually provides a great deal of the code for this pass, describing which native instructions should be generated for a particular SUIFvm instruction. Variables and virtual registers are assigned to physical registers by the register allocator, praga. A code-finalization pass, fin, inserts function prologues and epilogues and assigns memory addresses to any local variable names remaining in the code. After this the code may be scheduled, converted to assembly, or printed as source to a compiled simulator by subsequent passes.

3.1 The vliw Back-End

The architecture described in the previous chapter is implemented as a back-end to Machine SUIF. Machine SUIF's implementation allows for a variety of architectures to be implemented, provided that they implement functions in a structure called a *context* that answers questions for the rest of the system. These answers include information such as which registers use a caller-saved convention, or the string name corresponding to a particular opcode number. There is not native support in Machine SUIF for supporting architectures that execute multiple explicitly scheduled instructions in the same cycle, such as VLIW architectures.

The back-end for this architecture is named vliw; code can be generated from

SUIFvm for it by using the command `do_gen -target_lib vliw`. In addition to the opcodes listed in Appendix A, the back-end defines required opcodes for null instructions, labels in code, and line numbers.

When each file is compiled, the back-end code for the `gen` pass reads in a file called `native-names`. This file contains a list of procedure names, one to a line. If calls to any of these functions are encountered in processing the code, the back-end emits a NATIVE instruction instead of the normal call sequence. These are treated specially by the simulator; see Section 4.1.3 for details.

The back-end uses a simple convention for dealing with predicated instructions: every instruction has a predicate register as its first source operand. The code that converts particular SUIFvm instructions to native instructions inserts a predicate value as needed. If the instruction being translated has an annotation named `predicate`, that annotation is removed and its value is used as the predicate register. This allows a front-end pass to insert necessary predication information. If the scheduling annotations are not present, a default predicate of p1 is used, so that in unannotated instructions always execute.

Predication causes some problems when interfacing with the remainder of Machine SUIF. For example, the Machine SUIF distribution includes a liveness analyzer, but the semantics of predicated code are different from unpredicated code. Consider the following code sample:

```
     SEQ p2, p3, r10, r0
(p2) ADD r10, r0, 1
(p3) ADD r10, r0, 2
```

Register r10 gets the value 1 if it is equal to 0 on entry to the block, and 2 if not. Assuming the register is live on exit from the block, traditional liveness analysis would conclude that the third instruction would always overwrite the definition from the second instruction, and that the second instruction's definition of r10 reaches no uses. In reality, the SEQ instruction sets p2 and p3 to be logical inverses of each other, causing exactly one of the second and third instructions to execute. Therefore, both definitions will reach the end of the block.

The predication convention also causes problems in other parts of Machine SUIF, particularly the register allocator, which make assumptions about the format of instructions in the machine code. Changes made to accommodate the changed instruction format are described in Section 3.4.

3.2 Exported Information

While the Machine SUIF context system provides target-independent presentation of a significant amount of material, there is more variable information that we would like to provide to other passes in a standardized way. This section discusses additional functions provided by the back-end to schedule individual instructions and examine details of the target architecture.

3.2.1 VLIW Scheduling

A VLIW scheduler will schedule each Machine SUIF instruction for a particular cycle within the function, on a particular cluster, and on a particular functional unit. The functions in `vliw/vliwsched.h` provide standard ways to access this information, via get_instr_cycle(), get_instr_cluster(), and set_instr_fu(), along with equivalent set_...() functions.

Internally, scheduling is done with Machine SUIF annotations. Each of these three values is stored as an annotation containing a single long. The annotation keys are `instr_cycle`, `instr_cluster`, and `instr_fu`.

Support is also provided for two additional annotations used by the scheduler. set_new_block() and is_new_block() set and query a `new_block` annotation used at the start of a basic block; set_end_of_bundle() and is_end_of_bundle() control an `end_of_bundle` annotation added by the scheduler to the last instruction scheduled for a particular cycle.

3.2.2 Machine Characteristics

Since this back-end compiles for a family of related architectures, it is useful for other passes to know what particular architecture has been selected. Information on the actual details of the target architecture is made available through functions in `vliw/vliwinfo.h`.

Many passes need to know details of the processor's register structure. gpr_count(), fpr_count(), and pr_count() return the number of general-purpose, floating-point, and predicate registers. gpr_type() and analogues return the default Machine SUIF types for each register class. Machine SUIF requires a unique integer identifier for each register; reg_num_r() returns the number for a general purpose-register (e.g. r0), is_reg_gpr() determines whether an abstract register number corresponds to a general-purpose register, and reg_reg_num() returns the assembly register number corresponding to an abstract register number. Analogous functions exist for floating-point and predicate registers.

There are also functions to describe the higher-level characteristics of the processor. cluster_count() returns the number of clusters on a clustered VLIW machine, with 1 corresponding to the degenerate case of an unclustered machine. lane_count() returns the number of SIMD lanes on the machine, again with 1 corresponding to a non-SIMD machine. fu_count() returns the number of functional units per cluster. instr_delay() and instr_latency() return the timing characteristics of a particular instruction on a particular functional unit, or return the best timing on any functional unit if the functional unit number passed is -1.

3.3 The `vliwinfo` File

The back-end depends on an external file to finish the description of the target architecture. By default, it looks for a file named `vliwinfo` in the current directory, but the user may specify a different file by setting the `VLIWINFO` environment variable to point to the desired file.

The file contains multiple sections. The first section describes general characteristics of the processor; the remainder describe the operations possible in each functional unit and their scheduling constraints. Items in the file are currently very case- and format-sensitive; for example, a single-cluster machine must specify `1 clusters`, all in lowercase, and with a plural "clusters". Blank lines in the file are ignored, as are any lines that have a pound sign (`#`) as their first character.

Several sample files are included in Appendix C, including models for in-order, SIMD, and both clustered and unclustered VLIW architectures.

3.3.1 General Machine Characteristics

The first section of the `vliwinfo` file contains general characteristics of the target machine. Options may appear in any order, though one option from each category should be specified. If multiple options from the same category are specified, the last one in the file is used. Valid options include:

`4 clusters` Indicates the number of clusters in a VLIW machine, or the number of lanes in a SIMD machine.

`32 32-bit GPRs` Indicates the number and width of registers in the general-purpose (integer) register file. Only bit widths of 32 and 64 bits are recognized; at least 10 registers are required. The current simulator implementation only supports 32-bit integers.

`32 64-bit FPRs` Indicates the number and width of registers in the floating-point register file. Only bit widths of 32, 64, and 128 bits are recognized; at least 7 registers are required. The current simulator implementation only supports 64-bit floats, which are used for double-precision values. The simulator also simulates single-precision floating-point, using 32-bit values.

`16 predicate registers` Indicates the number of predicate registers available. At least 4 registers are required.

`4 functional units` Indicates the number of functional units in each cluster.

`Unrestricted loads` or `Restricted loads` Specifies whether load and store instructions can only be executed from the first cluster of a clustered VLIW machine (restricted), or from any cluster (unrestricted).

`Unrestricted transfers` or `Restricted transfers` Specifies whether only one inter-cluster transfer instruction (RCV) may be executed per cycle (restricted), or if any number of transfers may be scheduled (unrestricted). [1]

`VLIW mode` or `SIMD mode` Specifies whether the target machine is a clustered VLIW machine or a SIMD machine. The number of clusters is used as the number of lanes in SIMD mode. An unclustered VLIW machine with SIMD operations can be simulated by requesting SIMD mode here and specifying multiple functional units.

3.3.2 Functional Unit Descriptions

There is one functional unit description section for each functional unit on the machine. These sections list the instructions that it is possible to execute on each functional unit, along with scheduling information for that instruction. The *delay* and *latency* parameters are explained in Section 2.1.4.

A sample section might look like:

```
FU 4:
FMOV 1 1
FADD 2 2
FSUB 2 2
FMUL 2 8
FDIV 2 8
FCMP 1 1
```

This section is for the fourth functional unit. This is a floating-point unit; it can execute FMOV and the six floating-point compare instructions in a single cycle,

[1] Since the scheduler currently inserts SND instructions in the cluster sending data and has most variables live primarily in the first cluster, the effect of restricting transfers is only minimal.

and addition and subtraction in two cycles each. Multiplies and divides have an eight-cycle latency, but an instruction can be issued every two cycles.

In general, the form of an instruction line is

```
opcode delay latency
```

`delay` and `latency` are the delay and latency parameters for the opcode or opcodes described on this line. The `opcode` may be any of the opcodes from Appendix A. It may also be one of the names in Table 3.1, which refer to a set of related instructions. Specifying `ALU 1 1` in the `vliwinfo` file indicates a single-cycle latency and single-cycle delay for all simple integer ALU operations, for example.

Abbrev.	Description	Opcodes
`ALU`	Simple integer ALU operations	MOV LDA ADD SUB AND OR XOR SHL SHR SSR SPR RPR
`FALU`	Simple floating-point operations	FMOV FADD FSUB
`CVT`	Format conversions	CIS CID CSI CDI CSD CDS
`CVTI`	Integer-to-* conversions	CIS CID
`CVTF`	Floating-point-to-* conversions	CSI CDI CSD CDS
`XFR`	Inter-cluster transfers	SND FSND PSND RCV FRCV PRCV PACK UNPACK
`CMP`	Integer comparisons	SLT SLE SEQ SGE SGT SNE
`FCMP`	Floating-point comparisons	FLT FLE FEQ FGE FGT FNE
`UBR`	Unconditional branches	JMP JMPR JSR JSRR NATIVE

Table 3.1: `vliwinfo` Instruction Abbreviations

3.4 Changes to Machine SUIF

For the most part, the architecture could be implemented without making other changes to the Machine SUIF system. Machine SUIF's context system dispatches function calls to the back-end when needed, meaning that most of the details of the target architecture can be described without changing the core system. However, differences in the target architecture from Machine SUIF's expected model necessitated changes to parts of the system.

3.4.1 Generic Front-End Instructions

In addition to other objects generated from normal C code, the SUIF 2 front-end has a GenExpression class for extensions to the infrastructure. The normal SUIF 2 compiler does not use this class, and so the s2m pass to convert SUIF 2 code into Machine SUIF code does not handle these objects effectively. However, both the predication and SLP-detection passes used in the front-end cause GenExpressions to appear in the SUIF 2 code.

Machine SUIF already has an equivalent hook for adding instructions to the SUIFvm code: an ANY instruction in SUIFvm is annotated with the name of an operation, and can have arbitrary sources and destinations. s2m is modified to turn GenExpressions into ANY instructions, copying the operation-name annotation from the front-end object to the SUIFvm instruction. The code-generation pass can then examine the ANY instruction and produce appropriate native code depending on the attached operation name.

3.4.2 Initialization Hooks

Loading the `vliwinfo` file presented a number of problems within the Machine SUIF infrastructure. Ideally, the back-end could load the file once and somehow attach the associated data to the file. This is doable with annotations on SUIF file blocks, but retrieving the data is then difficult.

Machine SUIF does provide a number of initialization functions that get called, and one of these is an initialization function for the back-end library. At that point, the library checks for a `VLIWINFO` environment variable and loads the file named there. If that environment variable is not set, it loads a file named `vliwinfo` in the current directory instead. This approach leads to potential skew problems – for example, it is possible to run code generation with one `vliwinfo` file, and register allocation with another – but it is the best solution possible without making changes to the abstract code-generation pass.

3.4.3 Register Allocator

The Machine SUIF register allocator, raga, does several things beyond simply assigning physical registers to local variable names and virtual registers. If an instruction uses a register that needs to be spilled to the stack, raga inserts the necessary load and store instructions. raga also inserts load and store instructions for definitions and uses of global variables.

These inserted instructions create a number of complications with the simulated architecture. Most notably, the format of the inserted instructions is wrong: raga assumes that load instructions have a single address source and a single register destination, but the architecture requires an additional register source for the predicate register. A modified version of raga, praga (predicated raga), works around these problems. When an instruction is spilled, praga copies the predicate guard from this instruction to the instruction loading or storing the spilled register. This results in both correct predication and the correct instruction format.

praga also takes into account the annotations necessary for SIMD instructions. When spill code is generated, annotations are copied from the original instruction so that live values in the wide register are not overwritten. The simulator code also uses the ability of raga to add annotations to spill code to recognize spill instructions without an explicit SIMD annotation.

3.4.4 Liveness Analysis

Machine SUIF provides a liveness analyzer for virtual registers and variable operands. This is primarily used by the register allocator, though it is available as a general-purpose utility package.

As discussed previously, traditional liveness analyzers can incorrectly determine that certain variables are dead in predicated code. It is possible to determine what definitions reach which uses under various predication conditions. However, the Machine SUIF liveness analyzer is built on top of a bit-vector data-flow analysis library, and performing proper liveness analysis on predicated code requires more than one bit of data per variable.[2] Because of this tight coupling, it becomes very difficult to implement a proper liveness analysis in a way that can be easily used by raga. raga would need to be further modified to use the new liveness analyzer instead of the provided one.

A minor modification to the liveness analyzer lets it produce correct, though conservative, results with the existing bit-vector structure. Knowing that the first source of every instruction is a predicate register, the liveness analyzer can compare it to p1, the default predicate. The existing code path is used in this case. Otherwise, the modified liveness analyzer treats the destination register as an additional source, with the assumption that, if the predicate is false, the destination is assigned to itself. This means that a value continues to be live, even before a predicated definition of it. The resulting live ranges are longer than the actual live ranges of the variables, but since this is a more conservative result it will not cause the register allocator to assign conflicting variables to the same register.

[2]Specifically, it requires one bit of data per variable per predication variable.

Chapter 4

Simulator

Since the target architecture does not correspond to a real, physical processor, and since the system's flexibility allows for several different instantiations of the architecture, source code is ultimately compiled into a simulated binary which can be run natively on the local machine. The simulator code collects statistics on the performance the code would have on the simulated machine, including the number of simulated cycles it takes to run. Running the simulated binary has the same user-visible effects that running a natively-compiled binary of the same source code would have.

The simulator requires both Machine SUIF support and some separately compiled code. A Machine SUIF pass, simuprint, generates a C source file from the intermediate representation, along with a separate file listing every function and global data item. This pass is run for each input file. A linker then collects the global data files and produces a C header file which is included by the simulator source files, along with a memory image and a relocation table. The generated C files are then compiled together and linked against the simulator library. This produces a binary which, when run, loads the initial memory image and simulates the original program on the target architecture.

This approach has both advantages and disadvantages. The implementation of the simulator is fairly straightforward, and functions in the simulator library correspond directly to machine instructions. Compiling the simulator code natively allows somewhat better performance on long simulations as well, and affords the opportunity to avoid implementing system libraries. However, this sort of simulator can be harder to debug due to extra stages between the intermediate representation and the end simulator, and effectively skipping library calls can change the performance characteristics of many benchmarks.

4.1 Implementation

At the end of the compilation sequence, instead of using the Machine SUIF m2a pass to print out assembly code, a pass called simuprint writes out two files: a file containing C source code for the simulated binary, and a file containing global data. The global data file is discussed below in Section 4.2.1.

4.1.1 Emitting Instructions

The C source file includes a header file, `Simunow.h`, included as part of the simulator library. This header file defines a function for each opcode, along with several helper functions. If the source code included ADD r4, r1, r0, for example, the C file would contain a call similar to `__add(4, 1, 0)`. Each function call actually contains a number of additional parameters indicating what processor resources it has been scheduled to use and what address it occupies in instruction memory. Similar function calls are emitted for each instruction executing in parallel on the same cycle, followed by a call to __launch(), which causes outstanding operations to be committed.

There are some variants on the function names. For the ADD opcode, __add() would cover the basic case of a register-plus-register addition. However, __addu() is also defined as an unsigned addition, __addi() is a signed register-plus-immediate addition, and __addiu() is an unsigned register-plus-immediate addition. For floating-point operations, an “s” or “d” is appended to the opcode to select single- or double-precision. Similar conventions are used for other opcodes. The Machine SUIF types of the source and destination operands are examined to determine the correct function to call.

The only case where an assembly operand requires more than one C operand is in the case of a register-indirect memory address. This is used only for specific arguments of LD and ST instructions. In these cases, two C arguments are used to represent the base register and immediate offset of the appropriate source or destination.

Most of the machine opcodes are resolved in this fashion. Some opcodes, such as NULL or LINE, are only used internally by Machine SUIF, and are ignored by simuprint. The LDA opcode, which loads an address into a register, is treated specially: its parameter is either a global symbol or a register-indirect offset, both of which are emitted as simple adds.

4.1.2 Functions

Each function in the original source file is also printed as a C function in the output simulation file. This simplifies calls and returns, which can just be maintained as calls and returns in the C program. However, it does introduce minor complications in actually making the calls and in resolving namespace conflicts.

Because of this setup, the JMP, JSR, JMPR, and JSRR instructions have no direct effect. Instead, they return a boolean flag which indicates whether or not the instruction's predicate allows the branch to be taken. After the call to __launch(), appropriate action is taken based on the opcode: a JMP is generally translated to a C goto, a JSR to a function call, a JMPR to a return, and a JSRR to a function call through a function pointer.

C functions corresponding to simulated functions take no parameters and return nothing. They expect their parameters to come through the correct registers in the simulator, and return a value in r9 or f2 for integer and floating-point values, respectively. Aside from the actual call instruction, the simulated code should exactly obey the calling convention described in Appendix B.

4.1.3 Native Calls

Operating systems typically define a number of functions in external libraries, such as the system C library. While it is possible to reimplement these functions within the simulator, it is much easier to allow the simulator code to call out to system functions directly. It may also be desirable to implement parts of an application in native code for performance reasons. The simulator provides a *native call* mechanism to bypass the normal call scheme for these situations.

The NATIVE opcode requests a native call. For normal calls, the back-end generates a pre-call sequence, the actual call, and a post-call sequence, as described in Appendix B. For native calls, however, this calling convention is completely omitted from the assembly source, and a NATIVE instruction is inserted instead. The simulator code then translates this instruction into a direct call to the target procedure.

The NATIVE instruction has some number of register and immediate sources, and a register destination. Printing the actual call is fairly straightforward: each parameter is read from the appropriate register file and passed on in the C function call, and the return value, if any, is stored to the target register. The type of each operand is inspected to determine whether registers should be pulled from the general-purpose or floating-point register file.

The major complication comes from C's handling of function prototypes. Every value in a register is either a C int or a C double, and so without casts, every parameter to a native function is one of these two types. However, standard header file includes cause definitions to be included for some functions, causing the native C compiler to complain about pointer-integer mismatches. There are even worse problems when calling varargs procedures such as printf(), since in the absence of a prototype, the C compiler will implicitly create a prototype based on the parameters to the first call to the procedure.

To work around this problem, the printer emits a prototype for each function listed in both the external symbol table and the `native-names` annotation to the SUIF file. It does this via a pre-pass over each file that looks at each function in the external symbol table. SUIF 2 stores enough information in the symbol table to be able to reconstruct a C function prototype from the symbol, and this information is carried forward through Machine SUIF. The top of each simulator source file therefore contains prototypes for each function that is called natively, and passed parameters are cast to the correct types.

4.1.4 In-Simulator Functions

The nature of the compiled simulator also makes it possible to implement functions in the simulator library which are available to simulated code. This approach is used for several debugging functions used in simple test cases. It also lets a simple malloc() be written that just allocates space in the dynamic data area, with a trivial free() and corresponding calloc() and realloc() implementations. Implementations of the UNIX open(), close(), read(), and write() procedures are also provided.

For each function provided by the simulator library, the library code provides a function with the correct prefixed name, such as __sim_malloc(). The code printer will call the function with the standard calling conventions, and the exported function is always called with no C parameters. The exported function needs to call some internal simulator functions to retrieve its parameters; a malloc() would call `__getIntReg(0,5)` to get the number of bytes to allocate from r5 in the first cluster. Returning an integer value is done via register r9, so the code would call `__setIntReg(0, 9, value)`, followed by a call to `__launch()` to cause the written value to be committed to the register file.

This approach allows fairly straightforward implementations of wrappers to standard C library functions. For example, Figure 4-1 shows an implementation of close() that would be available to simulated code. This implementation just takes the file descriptor parameter from the simulator state, calls the normal C library close() function, and pushes the return value into r9.

```
void __sim_close()
{
  int simFd = __getIntReg(0, 5);
  int result = close(simFd);
  __setIntReg(0, 9, result);
  __launch();
}
```

Figure 4-1: Simulator Implementation of close()

4.2 Pointers and Memory

The memory model shown in Figure 2-1 assumes that memory starts at address 0 and extends for 256 megabytes above that. The simulator does internally allocate a 256-megabyte block of memory, but memory addresses in the simulated machine are actually native-machine addresses. That is, the first item in the static data area has a simulated address of 1024, but is actually referred to with an address 1024 greater than the address of the start of the allocated block representing the simulated memory.

This creates some problems with global data. Since the memory space is created with malloc(), the initial addresses of global memory items cannot be known, only their offsets relative to the start of the data area. The simulator's data loader therefore needs to load in a relocation table with the simulator addresses of global data items containing other memory addresses. Additionally, the linker emits a C header file containing the addresses of every data item; these addresses are offset by `__static_base_addr`, which contains the actual start address of the simulated memory's static data area.

4.2.1 Global Data

Each SUIF 2 file contains a global symbol table with symbols defined and used in the file. Some of these symbols have associated definitions; the global data associated with these symbols needs to be printed so that the simulator code can access it. The

Machine SUIF system does not provide any sort of abstraction around this global data, so the simulator printing pass needs to access the SUIF 2 data structures directly.

Information about the global data is printed to a separate file from the C source; this data file is read by the linker, described below. For each data item, a record is printed that includes the name of the symbol and its length in bytes. If the symbol table entry includes a definition, the definition is also printed. For function entries, a short record is emitted declaring that a function with that name exists.

Not all of the stored data consists of simple constant expressions. It is possibile to initialize a data structure with pointers to other data items or functions, for example; SUIF 2 represents these with UnaryExpressions which must be processed. References to other global data items are outputted as such to the global data file, leaving the linker responsible for resolving these to actual memory addresses.

4.2.2 Linker

The simulation-printing code described so far has two shortcomings. As noted, the initialized data is printed to a separate file from the actual simulated C code. It also fails to take into account interactions between multiple source files, providing no direct way to call a procedure defined in a separate file. A linker is run after the simuprint pass to assign simulated memory addresses to global data items, provide a memory image for statically initialized data, and create a C header file to provide information about global symbols to the simulated source.

The linker is implemented as a Perl script; the actual implementation is not terribly difficult. Each function and data item is read in and stored in an associative array. Addresses are assigned to each data item. Binary images are created for each data item, and the resulting images are concatenated to form a memory image. The memory image is written to `__sim_globals.dat`, the header file to `__sim_globals.h`, and a file listing simulator addresses of global data items containing other addresses in `__sim_globals.reloc`.

The linker also performs several sanity checks. It is an error for a symbol to be referenced that is not defined by any of the included files, for example. It is also an error for a data item to include more bytes of definition than the declared length of that item. This problem comes up frequently when processing floating-point numbers. It is valid, however, for a data item to be shorter than its declared length; the extra space is padded with zero bytes.

`__sim_globals.dat` contains a hex dump of the static data area of the simulator. This file is somewhat dependent on the host architecture. Integers and addresses are both stored using the host byte order, and floating-point numbers use the host format. As such, there will be problems trying to use the same memory image on separate machines. There may also be issues with compiling on non-32-bit architectures, such as the Alpha.

4.3 An Example

Figure 4-2 contains an example of compiling a simple function call referencing a static string. The pointer `str` is stored as a global variable, and is initialized to a pointer to a static anonymous string. The global data file, `__sim_globals.dat`, contains the pointer `str` with the offset of the string relative to the start of the static data area (0x00000004, stored as four little-endian bytes on an x86 machine). Since this is an address in simulator memory, the relocation file, `__sim_globals.reloc`, contains the offset of that data item; when the global data file is loaded, the address of the base of memory will be added to this word in simulator memory. This address is followed by the text of the string.

The linker also writes `__sim_globals.h`, which contains C preprocessor defines for the addresses of every static variable. As previously mentioned, these values are offset by `__static_base_addr`, which receives the physical address of the start of the simulated static data area. The header file is then included by each of the simulator C source files. In the original call to printString(), the parameter `str` is brought into register r5 by an LD instruction, which becomes a __lw() (load word) call in the simulator C source. The reference to the symbolic name `str` here uses the macro definition to get the actual address in physical memory.

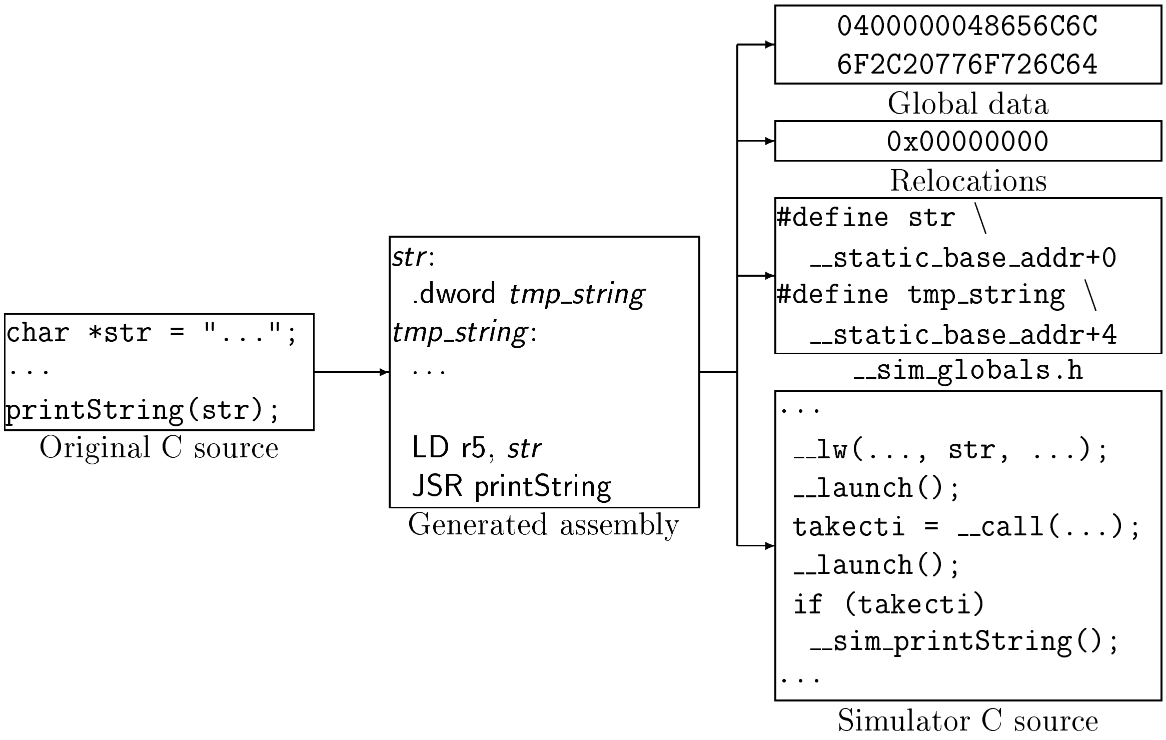

Figure 4-2: Global Data Relocation

Chapter 5

Compilation and Execution

5.1 Compilation

Compilation for the simulated environment uses the SUIF 2 system[2], together with Machine SUIF for the back-end. For SIMD architectures, effective compilation requires optimization passes written in SUIF 1[15]; in this case, a SUIF 1 front-end is used, with a SUIF 1-to-2 conversion pass inserted before the back-end passes.

5.1.1 Simple Compilation

A basic compilation sequence is shown in Figure 5-1. This uses a SUIF 2 front-end (here c2suif), then converts to Machine SUIF with s2m and uses the normal Machine SUIF back-end. The do_lower pass is provided with Machine SUIF; it performs a number of transformations, such as dismantling loops into labels and branches, that are necessary for s2m to properly digest the SUIF 2 code.

5.1.2 Extended Front-End

Instead of just using c2suif, it is possible to use a SUIF 1-based front-end and convert the resulting intermediate code to SUIF 2. The primary advantage to this compilation sequence, shown in Figure 5-2, is that optimization passes already developed for SUIF 1 can be used. In particular, the pass that detects superword-level parallelism and

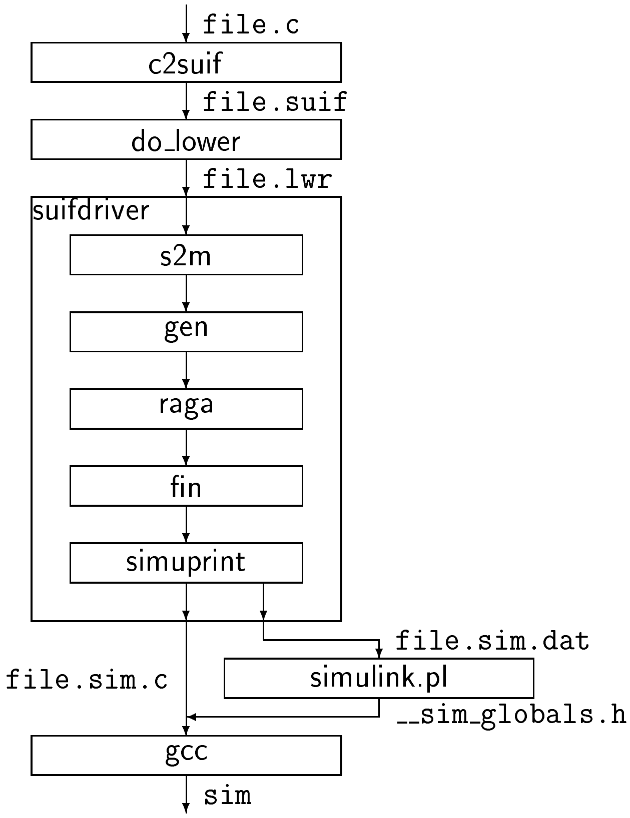

Figure 5-1: Basic Compilation Sequence

prepares the code for SIMD compilation is only available for SUIF 1.

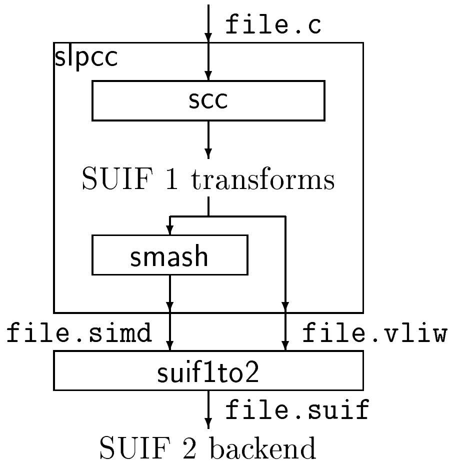

Figure 5-2: Extended Front-End

slpcc (Superword-Level Parallelism C Compiler) uses the SUIF 1 scc front-end to produce a SUIF 1 intermediate file from either C or Fortran intermediate code. It then applies a number of SUIF 1 passes to simplify the intermediate code, as well as performing operations such as loop unrolling that possibly make more superword-level parallelism available. If the target machine is a SIMD architecture, slpcc then runs a pass to reduce parallel, isomorphic instructions into single SIMD instructions[10].

After slpcc is run, the resulting SUIF 1 intermediate file can be converted into SUIF 2 using the suif1to2 pass provided with SUIF 2. Either the `file.simd` or `file.vliw` output files from slpcc can be passed into suif1to2. Since the front-end passes have already dismantled high-level constructs such as while and for loops, it is unnecessary to run do_lower; we instead move directly into the Machine SUIF back-end.

Using this extended front-end introduces a couple of minor problems. SUIF 1 does not have a useful internal representation for the special varargs calls, so calls to va_start() and its relatives do not get properly translated to SUIF 2. Additionally, there is not a SUIF 1 pass to change functions returning a structure value to take a pointer to the return value as a parameter, so these functions cannot be compiled using this front-end.

5.1.3 Extended Back-End

It is also possible to add extra passes to the back-end to accommodate compilation for VLIW machines and add a peep-hole optimizer. This is shown in Figure 5-3.

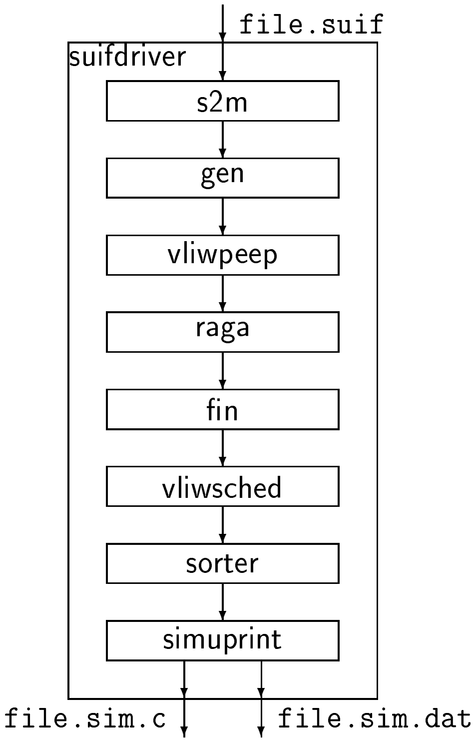

Figure 5-3: Extended Back-End

The scheduler in the vliwsched pass can be run even on SIMD architectures. It looks at each basic block and assigns a cycle number, cluster, and functional unit to each instruction. Machines without multiple functional units are viewed as a degenerate case of a VLIW architecture with only one cluster and one functional unit. By taking into account long-latency instructions such as integer multiplies, the scheduler can yield a performance improvement even on these architectures. A post-pass, sorter, rearranges the instructions to be in order by cycle, cluster, and functional unit to simplify the work of later passes.

5.2 Additional Components

A number of components besides the back-end are required to compile code for this architecture. This section describes these additional components.

5.2.1 Peep-hole Optimizer

The code produced by the back-end is generally of fairly poor quality. Some of this is due to its input; the `s2m` pass, which generates SUIFvm code from a SUIF 2 file, seems fond of inserting `cvt` instructions to convert a value to an equivalent integer type, for example. Additionally, the code generator must make conservative assumptions about the use of values set by comparison instructions: specifically, it must assume that the result of a comparison is a full integer which can be added to other integers in addition to being used as a branch condition.

The peep-hole optimizer, `vliwpeep`, transforms three common code sequences into simpler code. These are:

Move-use coalescing. If the result of a simple move instruction is used as a source in the next instruction, the move can be eliminated and the source replaced with the source of the move.

Def-move coalescing. If the result of an instruction is used as the source of a move in the next instruction, the destination of the first instruction can be replaced with the destination of the move and the move eliminated.

Integer predicate simplification. A comparison instruction always sets one predicate register and clears another. If these two predicate registers are used to set a general-purpose register to zero or one, and then that register is compared for equality to zero, the integer register can be dropped and the predicates set directly on the initial comparison.

In the absence of real liveness analysis, these optimizations only take place when the intermediate value is a virtual register, and when that virtual register only has a single use in the code. Additionally, optimization only happens when instructions have the same predication. Examples of these optimizations are shown in Figure 5-4.

	Original Code	Optimized Code
Move-use coalescing	MOV v0, r5 ADD r10, v0, 1	ADD r10, r5, 1
Def-move coalescing	ADD v1, r12, r11 MOV r9, v1	ADD r9, r12, r11
Integer predicate simplification	SLT v2, v3, r6, 0 (v2) ADD v4, r0, 1 (v3) ADD v4, r0, 0 SNE v5, v6, v4, 0	SLT v5, v6, r6, 0

Figure 5-4: Examples of Peep-hole Optimization

In spite of these limitations, the peep-hole optimizer gets very good results on compiled code. While it had little effect on some code, the peephole optimizer led to as much as a 50% speed improvement on some benchmarks. More details on these optimization results are included in Section 6.4.

5.2.2 Simulation Code Printer

To run simulated code, the back-end prints out a C++ source file based on the Machine SUIF assembly. The simuprint pass does this work. It takes as input finalized, scheduled, register-allocated code, and outputs two files. The `file.sim.cc` output file contains the actual C++ code; `file.sim.dat` contains definitions of global variables that may be used in other files.

For the most part, simuprint's work is fairly straightforward. Each function in the code is printed out as a C++ function in the simulator source. The names of objects are changed somewhat; function names have `__sim_` prepended to them, and static objects have `static` and the name of the enclosing object prepended. A static variable named `bar` in a function `foo()` would be named `foo_static_bar`, for example.

More details of the printer's work are included in Chapter 4.

5.2.3 Simulation Code Linker

The linker creates a memory image from the global data files written out by simuprint, writes a C header file giving addresses to global data items, and performs some consistency checks. It is run after every input source file has passed through simuprint. Details of the linker are included in Section 4.2.2.

5.2.4 Native C Compiler

The actual simulated binary is produced by using the system's native C compiler, such as gcc, to compile the `.sim.c` files produced by simuprint. The compiler needs to be given the path to the simulator library header files, and the binary needs to be linked against `libsim.a`. Running the resulting binary simulates the original compiled program; some statistics are written to a `stats.out` file.

Chapter 6

Results

This infrastructure was used to compare a number of different in-order, SIMD, and VLIW architectures on a set of sample benchmarks. These benchmarks included tests from the SPECfp95 benchmark set, the MediaBench test suite, and a set of multimedia kernels originally used to test the SLP compiler[10].

6.1 Methodology

Each of the benchmarks was compiled and run on a number of separate simulated machines; which machines are described in each section below. More detailed information, including the `vliwinfo` file for some of the machines, is included in Appendix C; a summary of the machine models used is in Table 6.1.

Compilation used the sequence described in Chapter 5, including the extended front- and back-ends described there. All of the passes shown in Figures 5-2 and 5-3 were executed, with the exception that the smash pass for detecting superword-level parallelism was only run for SIMD machines. Code compiled for every machine model was run through the VLIW scheduler for consistency and a modest performance improvement.

Every C library call was dispatched using the native call mechanism, with the exception of malloc(), calloc(), realloc(), free(), open(), close(), read(), and write(), which were implemented in the simulator library. FORTRAN programs were translated to

Description	Section
Figure 6-1: In-order, SIMD, and VLIW	
Baseline model: in-order, 32 registers	C.2
Basic SIMD: in-order, 4 lanes	C.3
VLIW/SIMD composite: int/FP/mem FUs, 4 lanes	C.5
Clustered VLIW: 4 clusters, int/FP/mem FUs	C.4
Large unclustered VLIW: 1 cluster, 4*int/4*FP/1*mem FUs	C.6
Figure 6-2: Clustered VLIW	
Basic VLIW: 1 cluster, int/FP/mem FUs	
Small clustered VLIW: 2 clusters, int/FP/mem FUs	
Clustered VLIW: 4 clusters, int/FP/mem FUs	C.4
Large clustered VLIW: 8 clusters, int/FP/mem FUs	
Figure 6-3: Register File Sizes	
Tiny-register model (baseline with 12 registers)	
Small-register model (baseline with 16 registers)	
Small-register model (baseline with 24 registers)	
Baseline model (32 registers)	C.2
Large-register model (baseline with 64 registers)	
Large-register model (baseline with 128 registers)	
Figure 6-4: Optimization Options	
Baseline model with peephole optimization and scheduling	C.2
Baseline model without peephole optimization	C.2
Baseline model without scheduling	C.2
Baseline model with neither peephole optimization nor scheduling	C.2
Figure 6-5: Memory Latency	
Baseline model with 32 registers and 3-cycle loads	C.2
Baseline model with 32 registers and 10-cycle loads	
Baseline model with 12 registers and 3-cycle loads	
Baseline model with 12 registers and 10-cycle loads	
Figure 6-6: Pipelining Options	
Baseline model (fully pipelined)	C.2
Baseline model with 2-cycle delay for long instructions	
Baseline model with delay = latency	

Table 6.1: Simulated Machine Models

C using the sf2c front-end to SUIF 1, and linked against a modified version of libF77 and libI77 that provides a __sim_main() function rather than main().

The graphs shown below measure the relative number of simulated cycles required to run benchmarks on a variety of architectures. Performance numbers are scaled to some reference model, typically a single-cluster, single-lane machine with 32 registers. The vertical axis shows the number of cycles taken, as a percentage of the number of cycles taken by the baseline model. Shorter bars represent benchmarks requiring fewer cycles to run, and therefore a performance increase over the reference model.

6.2 In-Order, SIMD, VLIW

Since the simulator statistics include the number of simulated cycles it took to run the program, we can compare the raw speed of the various machines running the benchmarks. It is particularly interesting to compare a SIMD machine against a similarly configured clustered VLIW machine. For comparison, simulations were also run on an "ideal" unclustered VLIW machine with the same number of non-memory functional units as the clustered machine, but all in a single cluster. Performance results from such a comparison are included in Figure 6-1.

As expected, the in-order machine had the worst performance of any of the machines, and the ideal VLIW machine tended to do the best. The two SIMD machines did very well, however, and the in-order SIMD machine tended to beat the clustered VLIW machine despite being able to execute fewer parallel operations on a given cycle.

Clustered VLIW machines are easily scalable to be larger or smaller by changing the number of clusters. Performance results for 1, 2, 4, and 8 clusters are shown in Figure 6-2. These models were all simulated with identical `vliwinfo` files like that in Section C.4, except that the number of clusters in the first line of the file was changed.

These results show that performance does not improve with increasing the number of clusters. There are two primary reasons for this. One is implementation difficulties

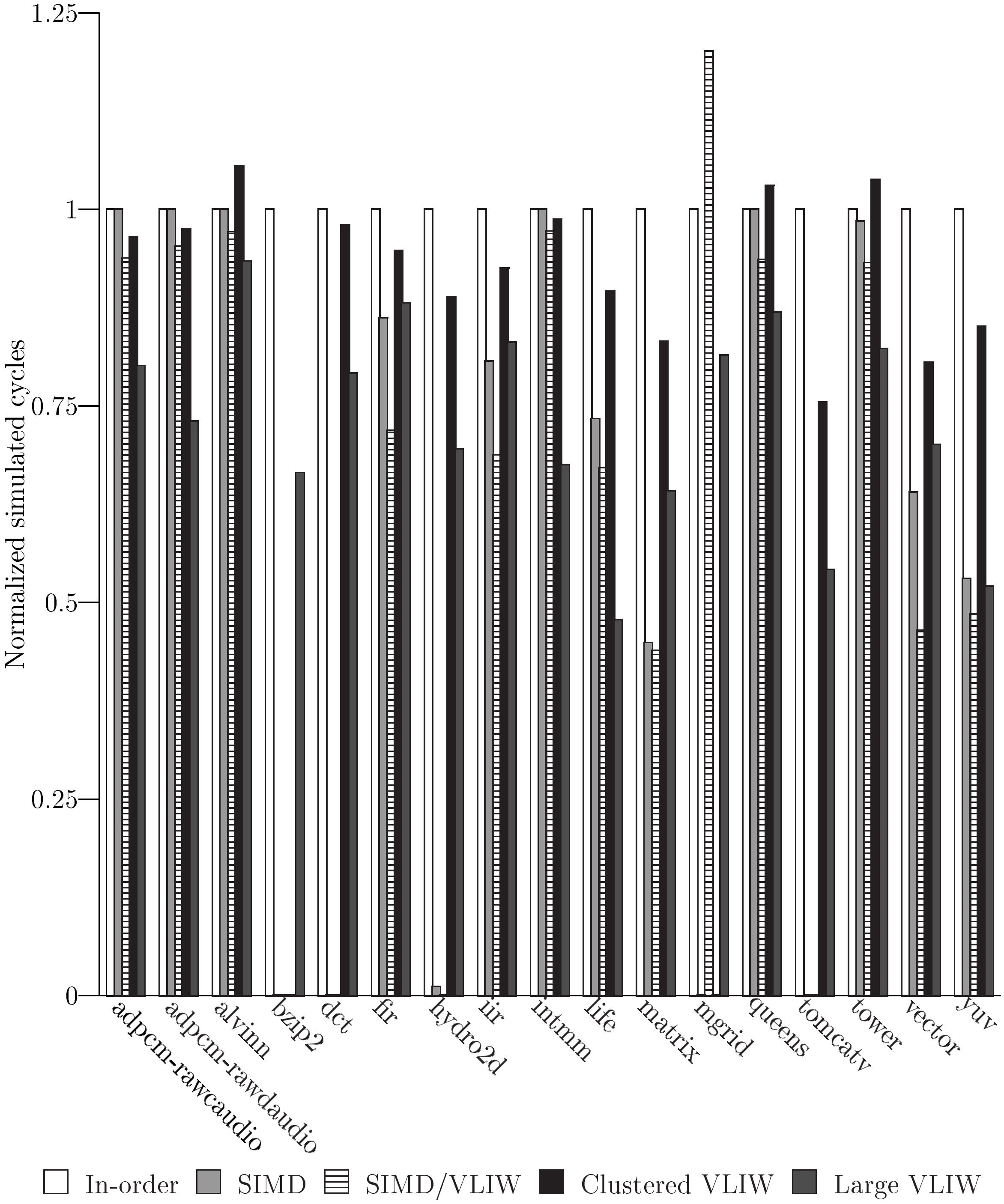

Figure 6-1: Performance of In-Order, SIMD, and VLIW Architectures

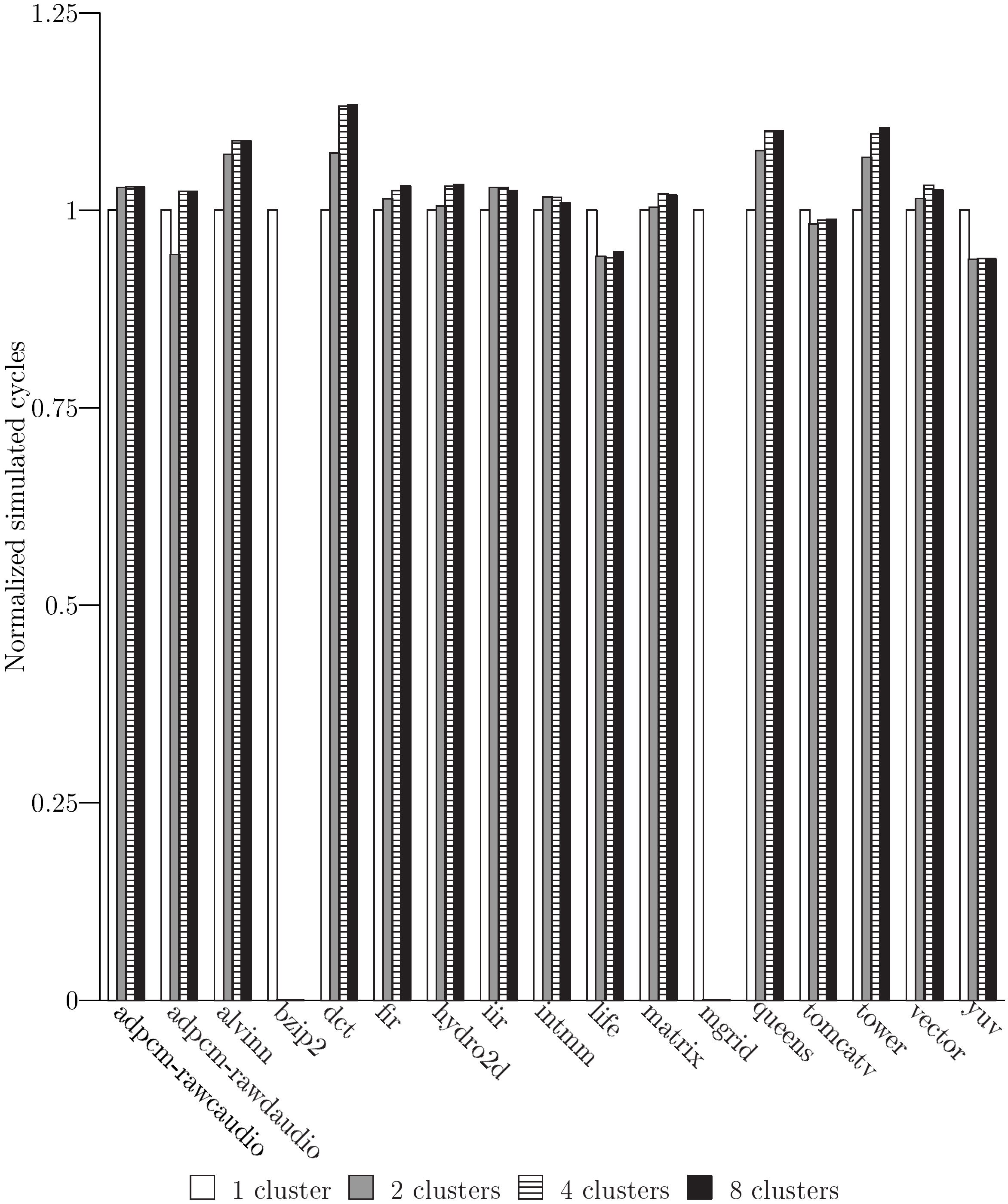

Figure 6-2: Performance of Clustered VLIW Architectures

within the scheduler, which does most of the actual computation within the first cluster. The second is the overhead cost for transferring data between clusters. Since the VLIW architectures require a full cycle to transfer a single value, the time requirement to perform some computation on a different cluster is the latency for the computation itself, plus a cycle for each input and output value. Additionally, since the scheduler keeps each register's "home" in the first cluster, no more than one transfer can be performed per cycle in any case. This results in an artificial slowdown that a better VLIW scheduler would be able to overcome.

6.3 Registers

One critical resource on most processors is the number of registers available in the register file. The number of registers is conventionally a power of 2 on a RISC machine, but the simulated machine models do not have this constraint. Figure 6-3 shows the performance results for simulations on the basic in-order machine described in Section C.2, with register files from 12 to 128 general-purpose and floating-point registers. Note that there are nine reserved registers, and that the register allocator will allocate reserved registers to local variables if it determines that they will not conflict with the registers' intended uses.

Simulated machines with smaller register files have substantially worse performance than machines with 32 integer and floating-point registers. The 12-register machine has only two non-dedicated integer registers, and 4 to 6 registers for general-purpose computation; on some benchmarks, register pressure causes the overall speed to be only half that of a 32-register machine.

Increasing the size of the register file does not seem to have a noticeable effect beyond 24 or 32 registers. There are exceptions to this, but 32 registers, the number present on the MIPS architecture, appears to be a good balance between allowing efficient code and minimizing the required processor resources.

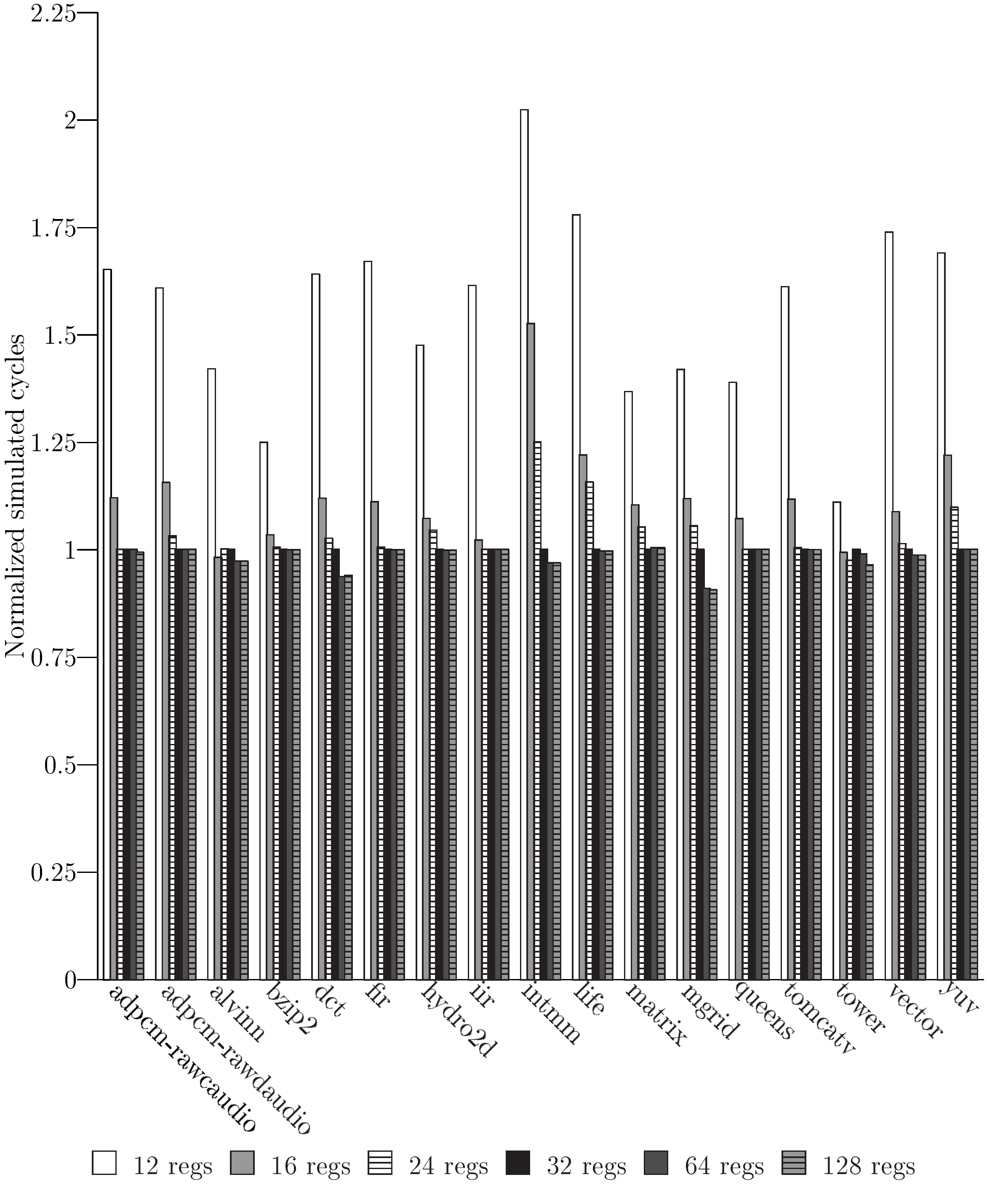

Figure 6-3: Performance with Varying Register File Sizes

6.4 Optimization

There are two major back-end passes which are run to improve the performance of the simulated code: the peep-hole optimizer and the scheduler. Recall that, by ordering instructions to minimize the time spent waiting for results of other instructions, the scheduler should provide a performance improvement even on an in-order machine.

Figure 6-4 shows the performance of the simple simulated in-order machine with peephole optimization and scheduling enabled or disabled. The baseline model runs both passes; other options, unsurprisingly, have worse performance. The results vary by benchmark, but both the simple peephole optimizations described in Section 5.2.1 and the VLIW scheduler can provide performance gains between 0 and 50 percent. On the MediaBench ADPCM audio benchmark, the scheduled code ran almost twice as fast as the unscheduled code. The YUV multimedia kernel also doubled its overall performance with optimization after gains from both the scheduler and the peephole optimizer.

6.5 Memory Latency

The current implementation of the simulator does not have a cache hierarchy. However, it is still possible to experiment by changing the latency of a load instruction. Since the `vliwinfo` file allows the latency of each instruction to be specified, slower memory can be simulated by increasing the latency of LD and FLD instructions. The baseline model in Section C.2 has a load latency of 3 cycles; Figure 6-5 shows the effects of increasing the latency to 10 cycles. For comparison, it also shows the performance of the 16-register machine with 3- and 10-cycle loads, demonstrating the effect that slower spills have on performance.

Most benchmarks do not have a significant slowdown; in most cases, the performance hit is between 5 and 15 percent from tripling the memory latency. This is consistent, since most operations are not, in fact, memory operations. Additionally, the scheduler can often shuffle instructions such that other operations can execute

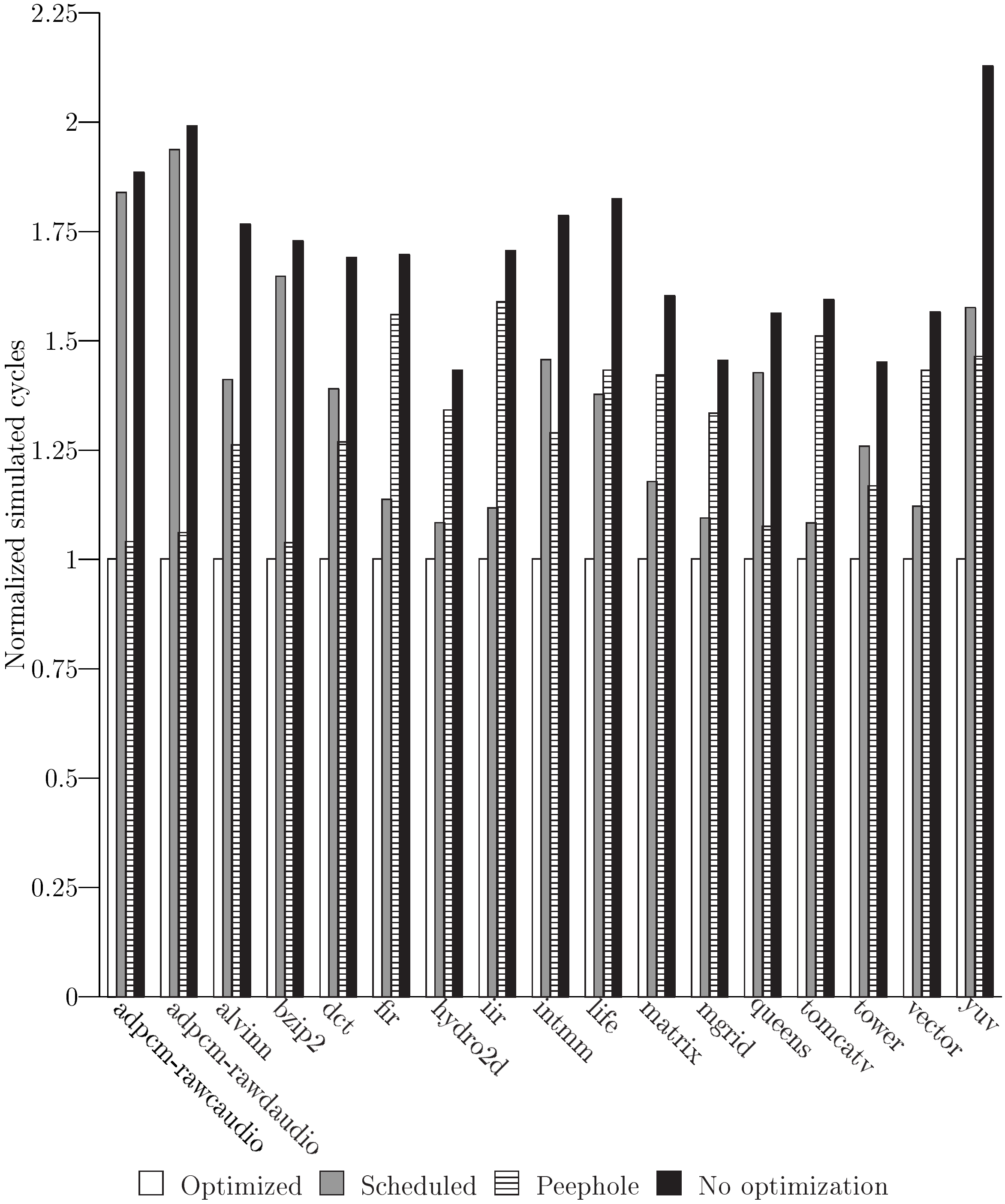

Figure 6-4: Performance with Varying Optimization Passes

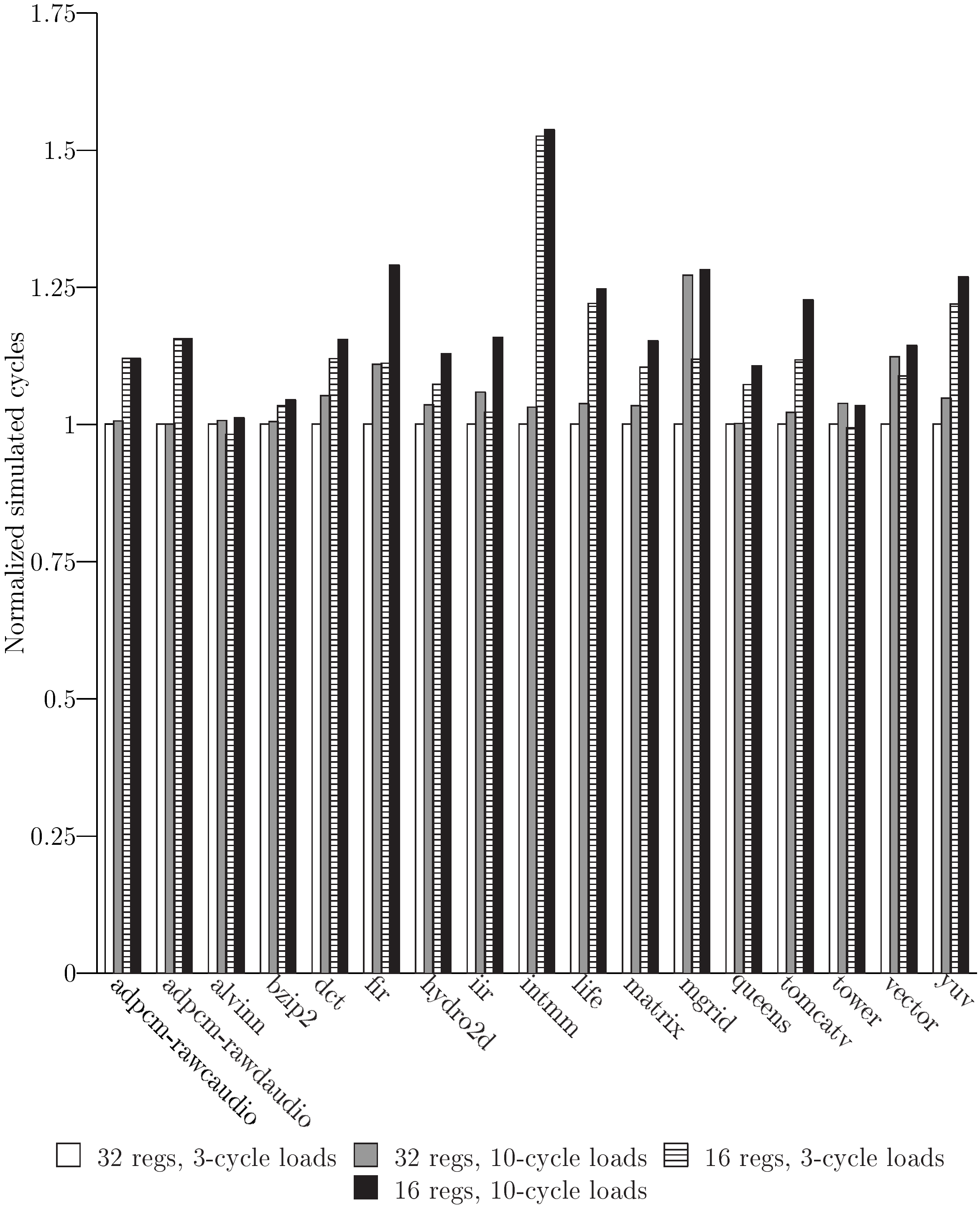

Figure 6-5: Performance with Varying Memory Latency

while code is waiting for a value from memory.

6.6 Pipelining

The machine models used so far have all assumed that all functional units are fully pipelined, that is, one instruction can be issued on each functional unit every cycle. The latency/delay scheduling scheme discussed in Section 2.1.4 makes it possible to simulate machines that are not fully pipelined.

Figure 6-6 shows three pipelining possibilities. The first is the basic, in-order, fully pipelined machine. The second is a partially pipelined machine in which any non-load instruction with a latency longer than one cycle stalls the processor for a second cycle. The third machine model used here is not pipelined at all; every instruction causes the processor to stall until that instruction completes.

6.7 Other Possible Modifications

A number of other modifications are possible to the system. While none of these were attempted for this thesis, they do offer possibilities for future work.

6.7.1 Mixed-latency Functional Units

All of the simulated machine models tested here used constant latencies for operations. If an integer multiply has a latency of 5 cycles on one functional unit, it has that same latency on every functional unit. Nothing intrinsically prohibits having multiple functional units that can run the same instruction, but with different latency and delay characteristics.

More extensive modifications to the system could allow simulation of more complicated architectures. It may be desirable to simulate a functional unit that could execute two additions or a single multiply in a given cycle, for example. The current system has no way to represent this type of functional unit.

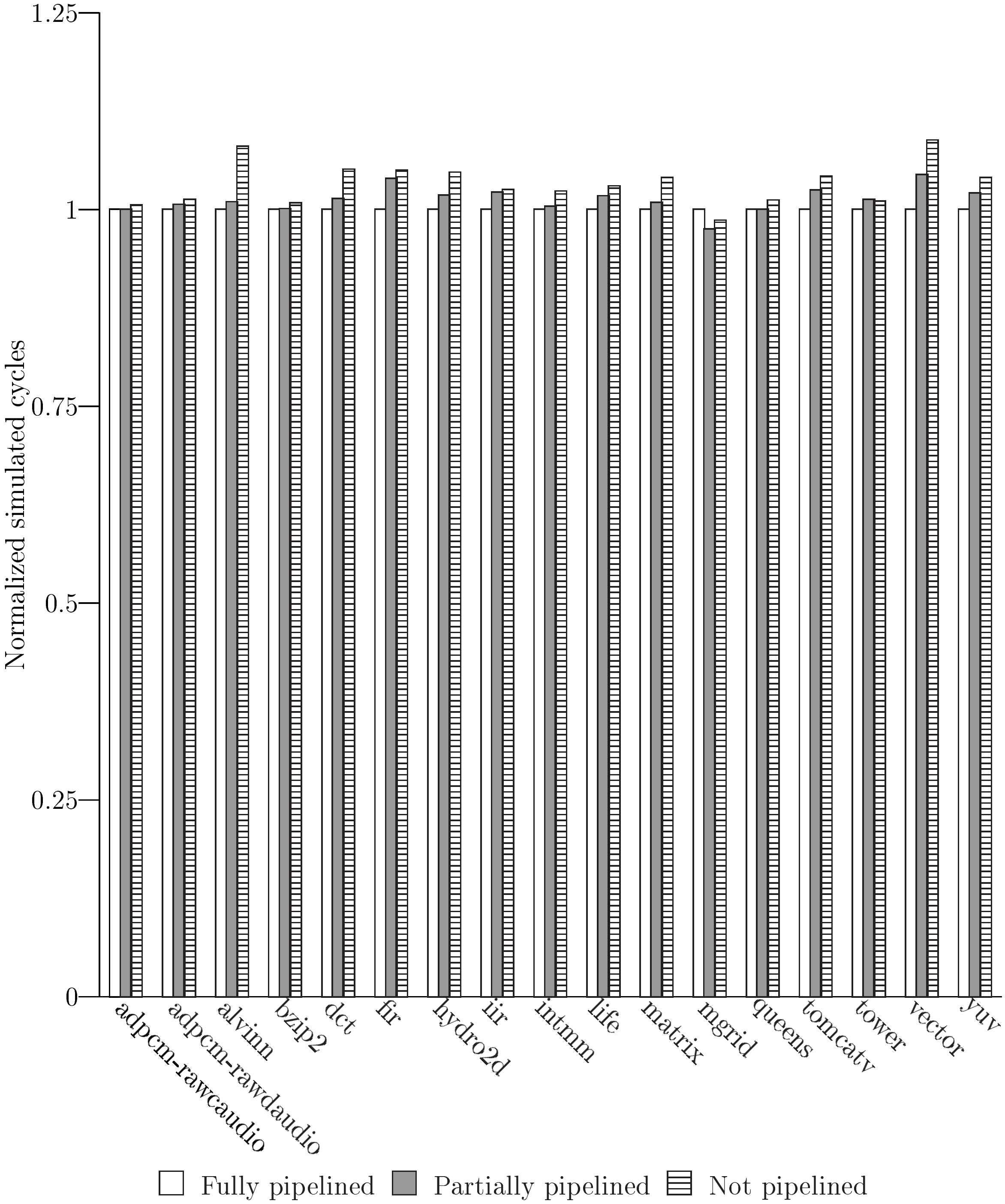

Figure 6-6: Performance with Different Pipeline Options

6.7.2 Additional Instructions

With some effort, instructions can be added to the simulated machine. This first requires adding the opcode for the instruction to the back-end, by modifying `opcodes.h.nw` and `opcodes.cpp` in the vliw back-end source. If the instruction satisfies any of the standard Machine SUIF instruction predicates, appropriate code needs to be added to `instr.cpp`.

A pass then needs to be written to generate the instruction. One option is to create a SUIF 2 GenExpression with a front-end pass, and then add code to `gen.cpp` to translate the resultant SUIFvm ANY instruction to the new opcode. Another is to write a pass that runs after code generation to replace existing instructions with new ones; this approach would work well to add a compare-and-branch instruction, for example.

Finally, the simulation infrastructure needs to be made aware of the instruction. The simuprint pass generates C code for each instruction; code needs to be added here to handle the added instruction. Additionally, code to handle the instruction at simulation time needs to be added to the simulation library, libsim.

Chapter 7

Conclusions

This thesis introduced infrastructure to compile and simulate code on a variety of architectures. By altering a single simple control file, the user can select an in-order, SIMD, or VLIW machine, adjust the number of functional units, registers, and SIMD lanes, and control the latency of individual instructions. Interfacing to Machine SUIF allowed the system to take advantage of pre-existing front- and back-end passes, including an SLP detection pass and a register allocator. Many of these architectural and compiler-flow possibilities were demonstrated, using the statistics generated by the simulator to show the number of cycles the compiled code takes to run.

Where traditional compiler back-ends and simulators target a single, fixed architecture, this thesis has presented a flexible infrastructure capable of targeting a wide variety of simulated machines. This infrastructure made a reasonable comparison between in-order, SIMD, and VLIW architectures possible; it could readily be adopted to add additional instructions to the simulated machine, or to experiment with allocating instructions between functional units. As such, this infrastructure should prove extremely useful for future architectural research.

7.1 Future Work

While many benchmarks could be built with this system, there are still many outstanding issues. For example, the scheduler tends to be extremely slow and run out of memory on large benchmarks, and many benchmarks fail in the conversion from SUIF 1 to SUIF 2. Particularly with SIMD compilation, it is also difficult to verify that the simulated code is correct. Ideally, the system should be able to compile and simulate all of the benchmarks in SPEC95 and MediaBench correctly on every machine model introduced here.

There are also work-arounds in the system that ideally could be avoided. There are two currently supported methods for simulated code calling code outside of the simulator: the native-call mechanism and adding simulator functions to the simulator library. It should be readily possible to compile C and Fortran libraries for any given machine model, and link compiled code against these libraries. Then the native-call mechanism could be eliminated, and a small number of required system calls implemented in the simulator library.

Many of the other possible improvements are algorithmic in nature. The scheduler algorithm could be greatly improved to provide speed increases on clustered VLIW machines. The back-end has predication support, but none of the tests actually used it; a predication pass could be written for the front-end to allow larger basic blocks and better scheduling.

The source code to the infrastructure described here will be made available through the COMMIT (Compilers at MIT) group Web page at `http://www.cag.lcs.mit.edu/commit/`.

Appendix A

Instruction Set

This appendix describes the instruction set used for the simulated machine. Each instruction is described with a line like

OPCODE $R_d \leftarrow R_s \ldots T$ *Description*

The middle section describes the destination and sources of the instruction. These are:

$\emptyset$	No destination
R	Integer register
RI	Integer register or immediate
RA	Address value (register+offset only)
A	Address value (symbol or register+offset)
F	Floating-point register
RF	Integer or floating-point register
FI	Integer or floating-point immediate
P	Predicate register
RFP	Integer, floating-point, or predicate register
T	Target label of a jump or call instruction

Each instruction has an unlisted predicate source; see Section 2.1.3 for details on how this is used.

The assembly notation lists the destination registers in order, followed by the sources and a target, if any. For example:

ADD	$R_d \leftarrow R_1\ RI_2$	ADD r12, r11, 12
FEQ	$P_t\ P_f \leftarrow F_1\ F_2$	FEQ p2, p3, f9, f0
JSR	$R_d \leftarrow T$	JSR r3, *label*

A.1 Standard Instructions

These instructions are useful on all simulated architectures, even on single-issue in-order machines.

ADD $R_d \leftarrow R_1\ RI_2$ *Integer addition*

The sources R_1 and RI_2 are added together and stored in R_d. Sign and overflow are ignored.

AND $R_d \leftarrow R_1\ RI_2$ *Integer bitwise AND*

The bitwise AND of R_1 and RI_2 is calculated and stored in R_d.

CDI $R_d \leftarrow F_s$ *Convert double-precision floating-point to integer*

The value in F_s is interpreted as a double-precision floating-point number and rounded down to the nearest integer, with the result stored in R_d.

CDS $F_d \leftarrow F_s$ *Convert double-precision to single-precision*

The value in F_s is interpreted as a double-precision floating-point number and stored in F_d as a single-precision float.

CID $F_d \leftarrow R_s$ *Convert integer to double-precision floating-point*

The value in R_s is converted to floating-point and stored in F_d as a double-precision float.

CIS $F_d \leftarrow R_s$ *Convert integer to single-precision floating-point*

The value in R_s is converted to floating-point and stored in F_d as a single-precision float.

CSD $F_d \leftarrow F_s$ *Convert single-precision to double-precision*

The value in F_s is interpreted as a single-precision floating-point number and stored in F_d as a double-precision float.

CSI $R_d \leftarrow F_s$ *Convert single-precision floating-point to integer*

The value in F_s is interpreted as a single-precision floating-point number and rounded down to the nearest integer, with the result stored in R_d.

DIV $R_d \leftarrow R_1\ R_2$ *Integer divide*

R_1 is divided by R_2, and the quotient stored in R_d. Sign is ignored.

FADD $F_d \leftarrow F_1\ F_2$ *Floating-point addition*

The sum of the values in F_1 and F_2 is stored in F_d.

FDIV $F_d \leftarrow F_1\ F_2$ *Floating-point division*

F_1 is divided by F_2, and the result is stored in F_d.

FEQ $P_t\ P_f \leftarrow F_1\ F_2$ *Set predicate on floating-point equal*

Sets P_t and clears P_f if $F_1 = F_2$; otherwise, clears P_t and sets P_f.

FGE $P_t\ P_f \leftarrow F_1\ F_2$ *Set predicate on floating-point greater than or equal*

Sets P_t and clears P_f if $F_1 \geq F_2$; otherwise, clears P_t and sets P_f.

FGT $P_t\ P_f \leftarrow F_1\ F_2$ *Set predicate on floating-point greater than*

Sets P_t and clears P_f if $F_1 > F_2$; otherwise, clears P_t and sets P_f.

FLD $F_d \leftarrow RA_s$ *Floating-point load*

F_d receives the floating-point value in memory at RA_s.

FLE $P_t\ P_f \leftarrow F_1\ F_2$ *Set predicate on floating-point less than or equal*

Sets P_t and clears P_f if $F_1 \leq F_2$; otherwise, clears P_t and sets P_f.

FLT $P_t\ P_f \leftarrow F_1\ F_2$ *Set predicate on floating-point less than*

Sets P_t and clears P_f if $F_1 < F_2$; otherwise, clears P_t and sets P_f.

FMOV $F_d \leftarrow F_s$ *Copy a value between registers*

FMOV copies the value from F_s into F_d.

FMUL $F_d \leftarrow F_1\ F_2$ *Floating-point multiplication*

The product of the values in F_1 and F_2 is stored in F_d.

FNE $P_t\ P_f \leftarrow F_1\ F_2$ *Set predicate on floating-point not equal*

Sets P_t and clears P_f if $F_1 \neq F_2$; otherwise, clears P_t and sets P_f.

FST $\emptyset \leftarrow F_1\ RA_2$ *Floating-point store*

The value in F_1 is stored to memory at RA_2.

FSUB $F_d \leftarrow F_1\ F_2$ *Floating-point subtraction*

F_2 is subtracted from F_1, and the result is stored in F_d.

JMP $\emptyset \leftarrow T$ *Jump*

Jumps to T.

JMPR $\emptyset \leftarrow R_t$ *Jump to register*

Jumps to the address in R_t.

JSR $R_d \leftarrow T$ *Jump to subroutine*

Jumps to T, saving the next value of the program counter in R_d.

JSRR $R_d \leftarrow R_t$ *Jump to subroutine via register*

Jumps to the address in R_t, saving the next value of the program counter in R_d.

LD $R_d \leftarrow RA_s$ *Integer load*

R_d receives the integer value in memory at RA_s.

LDA $R_d \leftarrow A_s$ *Load address*

LDA loads an address into a general-purpose register. If A_s is a symbol, the address of the symbol is loaded into R_d; if it is of the form $R_b(I_o)$, the base R_b and offset I_o are added together.

MOD $R_d \leftarrow R_1\ R_2$ *Integer remainder*

R_1 is divided by R_2, and the remainder stored in R_d. Sign is ignored.

MOV $R_d \leftarrow R_s$ *Copy a value between registers*

MOV copies the value from R_s into R_d.

MUL $R_d \leftarrow R_1\ R_2$ *Integer multiply*

MUL multiplies the values in R_1 and R_2, and stores the low-order bits of the product in R_d. Sign and overflow are ignored.

NATIVE $RF_d \leftarrow \ldots T$ *Execute an in-simulator call*

NATIVE causes the simulator to execute a "native" call; instead of simulating machine instructions that make up the call, the simulator calls T directly, with the other sources as parameters. The target may be an integer or floating-point register or absent; in the latter case, the result of the function call, if any, is discarded.

NOP $\emptyset \leftarrow \emptyset$ *Does nothing*

NOP does nothing. It can be used as a placeholder when doing scheduling.

OR $R_d \leftarrow R_1\ RI_2$ *Integer bitwise OR*

The bitwise OR of R_1 and RI_2 is calculated and stored in R_d.

RPR $\emptyset \leftarrow R_d\ I_1$ *Restore predicate registers*

RPR restores predicate registers from a general-purpose register. As many predicate registers as there are bits in a general-purpose register are loaded; the order is unspecified (though the same as used by SPR). The source I_1 indicates which registers are to be restore, in multiples of the integer word width. If there are 64 predicate registers and general-purpose registers are 32 bits wide, then RPR r10, 1 would load registers p32 through p63 from r10.

SEQ $P_t\ P_f \leftarrow R_1\ RI_2$ *Set predicate if equal*

Sets P_t and clears P_f if $R_1 = RI_2$; otherwise, clears P_f and sets P_t.

SGE $P_t\ P_f \leftarrow R_1\ RI_2$ *Set predicate if greater than or equal*

Sets P_t and clears P_f if $R_1 \geq RI_2$; otherwise, clears P_f and sets P_t.

SLT $P_t\ P_f \leftarrow R_1\ RI_2$ *Set predicate if greater*

Sets P_t and clears P_f if $R_1 > RI_2$; otherwise, clears P_f and sets P_t.

SHL $R_d \leftarrow R_1\ RI_2$ *Integer shift left*

The source R_1 is shifted left by the number of bits specified in RI_2, with the result stored in R_d. Extra bits shifted in on the right are set to 0.

SHR $R_d \leftarrow R_1\ RI_2$ *Integer shift right*

The source R_1 is shifted right by the number of bits specified in RI_2, with the result stored in R_d. Extra bits shifted in on the left are set to 0.

SLE $P_t\ P_f \leftarrow R_1\ RI_2$ *Set predicate if less than or equal*

Sets P_t and clears P_f if $R_1 \leq RI_2$; otherwise, clears P_f and sets P_t.

SLT $P_t\ P_f \leftarrow R_1\ RI_2$ *Set predicate if less than*

Sets P_t and clears P_f if $R_1 < RI_2$; otherwise, clears P_f and sets P_t.

SNE $P_t\ P_f \leftarrow R_1\ RI_2$ *Set predicate if not equal*

Sets P_t and clears P_f if $R_1 \neq RI_2$; otherwise, clears P_f and sets P_t.

SPR $R_d \leftarrow I_1$ *Save predicate registers*

SPR saves predicate registers into a general-purpose register. As many predicate registers as there are bits in a general-purpose register are stored; the order is unspecified. The source I_1 indicates which registers are to be saved, in multiples of the integer word width. If there are 64 predicate registers and general-purpose registers are 32 bits wide, then SPR r10, 1 would save registers p32 through p63 into r10.

SSR $R_d \leftarrow R_1\ RI_2$ *Integer shift right, with sign extension*

The source R_1 is shifted right by the number of bits specified in RI_2, with the result stored in R_d. Extra bits shifted in on the left are set to the high bit of R_1.

ST $\emptyset \leftarrow R_1\ RA_2$ *Integer store*

The value in R_1 is stored to memory at RA_2.

SUB $R_d \leftarrow R_1\ RI_2$ *Integer subtraction*

The source RI_2 is subtracted from R_1, and the result is stored in R_d. Sign and overflow are ignored.

XOR $R_d \leftarrow R_1\ RI_2$ *Integer bitwise exclusive OR*

The bitwise exclusive OR of R_1 and RI_2 is calculated and stored in R_d.

A.2 VLIW Extensions

On a VLIW machine, each instruction carries annotations for scheduling information. These annotations indicate the scheduled cycle within the current function, the cluster the instruction runs on, and which functional unit within that cluster is being used. The following additional instructions are used on VLIW machines:

RCV $RFP_d \leftarrow RFP_s\ I_1\ I_2$ *Inter-cluster receive*

RCV copies an instruction between clusters. The value in register RFP_s in cluster I_1 is copied to register RFP_d in cluster I_2. The registers can be of any type, though both must be the same.

SND $\emptyset \leftarrow RFP_s$ *Inter-cluster send*

SND is used as a placeholder indicating that the value in RFP_s is being sent to another cluster. The register operand may be of any type.

A.3 SIMD Extensions

On a SIMD machine, each instruction carries an annotation indicating which lanes are in use. Unannotated instructions are assumed to be in effect on all lanes.

PACK $RF_d \leftarrow RF_1 \ldots$ *SIMD pack*

PACK packs multiple scalar values into a single SIMD long word. There are as many sources as active SIMD lanes; the instruction also carries an annotation indicating which lanes are in use for the source operands. Each value is copied from the specified register and lane to the specified lane in RF_d. The sources and destination can be of either integer or floating-point type, but all must be of the same type.

UNPACK $RF_1 \ldots \leftarrow RF_s$ *SIMD unpack*

UNPACK unpacks a SIMD wide register into a multiple scalar values. There are as many destinations as active SIMD lanes, and an annotation indicating which lanes of the target registers should be used for the destination values.

Appendix B

Register and Calling Conventions

This appendix describes the uses of general-purpose, floating-point, and predicate registers on the simulated architecture, along with a standard calling convention for use by compiled procedures.

B.1 Registers

Some of the registers in the register file are reserved for various operations. These are listed in Table B.1. These reserved registers are required to establish conventions for register usage, and allow a limited number of integer or floating-point function parameters to be passed directly in registers. This is a common technique for avoiding expensive memory operations on function calls.

The stack pointer, `r1`, begins at the top of memory and points to the last word used on the stack; each stack frame is written below the previous one. The frame pointer, `r4`, points to the top of the current stack frame. The stack frame is illustrated in Figure B-1.

B.2 Calling Convention

On a procedure call, the following steps are taken:

Pre-call sequence. The caller reserves space on the stack for caller-saved registers

Register	Description
r0	Always zero when read
r1	Stack pointer
r2	Global data pointer
r3	Return address
r4	Frame pointer
r5	Integer parameter 1
r6	Integer parameter 2
r7	Integer parameter 3
r8	Integer parameter 4
r9	Return value
f0	Always 0.0 when read
f1	Always 1.0 when read
f2	Return value
f3	Floating-point parameter 1
f4	Floating-point parameter 2
f5	Floating-point parameter 3
f6	Floating-point parameter 4
p0	Always false when read
p1	Always true when read

Table B.1: Reserved Registers

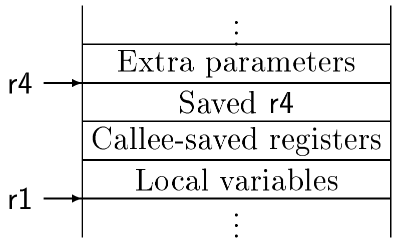

Figure B-1: Stack Frame

and parameters by subtracting an appropriate value from r1. Caller-saved registers are saved on the stack. Parameters are loaded into registers as appropriate; if there are more parameters than available parameter registers, excess parameters are pushed on to the stack, with the earliest parameter being closest to the bottom of the stack.

Call. The caller executes JSR r3, *target*.

Prologue. The callee allocates a stack frame by subtracting from r1. The old value of r4 is saved at the top of the stack frame, and this address is saved in r4 as the frame pointer. Other callee-saved registers are saved in the frame. Space is also allocated for local variables as needed.

Callee body. The procedure may allocate extra stack space if needed by subtracting from r1. The alloca() procedure may do this to allocate temporary stack space; handling of procedures with a variable number of arguments also uses this technique.

Epilogue. Callee-saved registers are restored from the stack. The stack pointer is returned to the current value of the frame pointer, and the frame pointer is returned to its previous value.

Return. If the function returns a value, the value is placed in r9 or f2 as appropriate. The callee then executes JR r3 to return.

Post-call sequence. The caller restores caller-saved registers from the stack, and adds to r1 the size of the saved-register and parameter area, returning the stack pointer to its original value.

Appendix C

Simulated Machine Models

This appendix describes the machine models used in simulation tests. An effort was made to make the simulated machines as similar as possible, though some differences are of course inevitable.

C.1 Generalities

All of the machines share some characteristics. These include:

Register files. All of the machines contain 32 32-bit general-purpose registers, 32 64-bit floating-point registers, and 32 predicate registers. For the clustered VLIW machines, there are this many registers per cluster; for the SIMD machines, each of these registers exist in each lane.

Functional units. No machine can execute a particular opcode more than once in a cycle (per lane on a SIMD machine, per cluster on a VLIW machine). The simulated VLIW machines have a memory/transfer unit, an integer unit, and a floating-point unit, and one instruction can be executed on each of these in parallel.

Scheduling. All of the machines have identical scheduling constraints. Integer ALU instructions have a latency of one cycle, memory operations 3 cycles, multiplies 5, and divides 20. Floating-point arithmetic requires 4 cycles, and divides 12.

Conversions between integer and floating point also require four cycles. Every operation is fully pipelined, with a delay of one cycle.

C.2 In-Order Machine

The basic machine model is a simple in-order machine. It is capable of executing one instruction per cycle.

```
1 clusters
32 32-bit GPRs
32 64-bit FPRs
32 predicate registers
1 functional units
Unrestricted loads
Unrestricted transfers
VLIW mode

FU 1:
LD 1 3
ST 1 1
UBR 1 1
ALU 1 1
CMP 1 1
MUL 1 5
DIV 1 20
MOD 1 20
FLD 1 3
FST 1 1
FMOV 1 1
FADD 1 4
FSUB 1 4
FCMP 1 4
FMUL 1 4
FDIV 1 12
CIF 1 4
CFI 1 4
XFR 1 1
```

C.3 Basic SIMD Machine

The first machine model can be extended by adding SIMD functionality. This also allows one instruction per cycle, but now each instruction can execute on up to four data words in parallel.

```
4 clusters
32 32-bit GPRs
32 64-bit FPRs
32 predicate registers
1 functional units
Unrestricted loads
Unrestricted transfers
SIMD mode

FU 1:
LD 1 3
ST 1 1
UBR 1 1
ALU 1 1
CMP 1 1
MUL 1 5
DIV 1 20
MOD 1 20
FLD 1 3
FST 1 1
FMOV 1 1
FADD 1 4
FSUB 1 4
FCMP 1 4
FMUL 1 4
FDIV 1 12
CIF 1 4
CFI 1 4
XFR 1 1
```

C.4 Basic VLIW Machine

We can also change the first machine model into a clustered VLIW machine. This machine has four clusters, each of which has a memory unit, an integer unit, and a floating-point unit. Loads and stores can be performed from any cluster; an unlimited number of transfer instructions are theoretically possible.

```
4 clusters
32 32-bit GPRs
32 64-bit FPRs
32 predicate registers
3 functional units
Unrestricted loads
Unrestricted transfers
VLIW mode

FU 1:
LD 1 3
ST 1 1
UBR 1 1
FLD 1 3
FST 1 1
XFR 1 1

FU 2:
ALU 1 1
CMP 1 1
MUL 1 5
DIV 1 20
MOD 1 20

FU 3:
FMOV 1 1
FADD 1 4
FSUB 1 4
FCMP 1 4
FMUL 1 4
FDIV 1 12
CIF 1 4
CFI 1 4
```

C.5 SIMD/VLIW Hybrid

We can combine the two machines to produce a VLIW machine with SIMD extensions. This is a machine with a single cluster, that can perform a wide load, four identical integer operations, and four identical floating-point operations per cycle.

```
4 clusters
32 32-bit GPRs
32 64-bit FPRs
32 predicate registers
3 functional units
Unrestricted loads
Unrestricted transfers
SIMD mode

FU 1:
LD 1 3
ST 1 1
UBR 1 1
FLD 1 3
FST 1 1
XFR 1 1

FU 2:
ALU 1 1
CMP 1 1
MUL 1 5
DIV 1 20
MOD 1 20

FU 3:
FMOV 1 1
FADD 1 4
FSUB 1 4
FCMP 1 4
FMUL 1 4
FDIV 1 12
CIF 1 4
CFI 1 4
```

C.6 Ideal VLIW

Both the SIMD machine and the clustered VLIW machine have restrictions that the compiler must work around. Any program that could be scheduled for either of these machines could be equivalently scheduled for a machine that has the same number of functional units and registers, but all in a single cluster. This is an idealized machine; the resources required to interface this large a register file to this many functional units would be impractical to implement on a real processor. However, it does provide a useful upper bound for the performance possible under these models.

```
1 clusters
128 32-bit GPRs
128 64-bit FPRs
32 predicate registers
9 functional units
Unrestricted loads
Unrestricted transfers
VLIW mode

FU 1:
LD 1 3
ST 1 1
UBR 1 1
FLD 1 3
FST 1 1
XFR 1 1

FU 2:
ALU 1 1
CMP 1 1
MUL 1 5
DIV 1 20
MOD 1 20

FU 3:
ALU 1 1
CMP 1 1
MUL 1 5
DIV 1 20
MOD 1 20

FU 4:
ALU 1 1
CMP 1 1
MUL 1 5
DIV 1 20
MOD 1 20
```

```
FU 5:
ALU 1 1
CMP 1 1
MUL 1 5
DIV 1 20
MOD 1 20

FU 6:
FMOV 1 1
FADD 1 4
FSUB 1 4
FCMP 1 4
FMUL 1 4
FDIV 1 12
CIF 1 4
CFI 1 4

FU 7:
FMOV 1 1
FADD 1 4
FSUB 1 4
FCMP 1 4
FMUL 1 4
FDIV 1 12
CIF 1 4
CFI 1 4

FU 8:
FMOV 1 1
FADD 1 4
FSUB 1 4
FCMP 1 4
FMUL 1 4
FDIV 1 12
CIF 1 4
CFI 1 4

FU 9:
FMOV 1 1
FADD 1 4
FSUB 1 4
FCMP 1 4
FMUL 1 4
FDIV 1 12
```

```
CIF 1 4
CFI 1 4
```

Bibliography

[1] The Trimaran compiler research infrastructure for instruction level parallelism. `http://www.trimaran.org/`.

[2] Gerald Aigner, Amer Diwan, David L. Heine, Monica S. Lam, David L. Moore, Brian R. Murphy, and Constantine Sapuntzakis. An overview of the suif2 compiler infrastructure. In *The SUIF 2 Documentation Set*. 2000.

[3] Todd M. Austin. A hacker's guide to the SimpleScalar architectural research tool set, December 1996.

[4] Doug Berger and Todd M. Austin. The simplescalar tool set, version 2.0. Technical Report 1342, University of Wisconsin-Madison Computer Sciences Department, June 1997.

[5] J. A. Fisher. Very long instruction word architectures and the ELI-512. In *Proceedings of the Tenth International Symposium on Cumputer Architecture*, pages 140–150. Computer Society Press, 1983.

[6] Linley Gwennap. AltiVec vectorizes PowerPC. *Microprocessor Report*, 12(6):1,6–9, May 1998.

[7] Silvina Zimi Hanono. *Aviv: A Retargetable Code Generator for Embedded Processors*. PhD thesis, Massachusetts Institute of Technology, June 1999.

[8] J. Huck, D. Morris, J. Ross, A. Knies, H. Mulder, and R. Zahir. Introducing the IA-64 architecture. *IEEE Micro*, 20(5):12–23, September/October 2000.

[9] Vinod Kathail, Michael S. Schlansker, and B. Ramakrishna Rau. Hpl-pd architecture specification: Version 1.1. Technical Report HPL-93-80 (R.1), Hewlett-Packard Laboratories, February 2000.

[10] Sam Larsen. Exploiting superword level paralellism with multimedia instruction sets. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, May 2000.

[11] Rainer Leupers. Instruction scheduling for clustered VLIW DSPs. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, Philadelphia, PA, October 2000.

[12] S. A. Mahlke, R. E. Hank, J. E. McCormick, D. I. August, and W. m. Hwu. A comparison of full and partial predicated execution support for ILP processors. In *Proceedings ISCA '22*, pages 138–149, June 1995.

[13] Alex Peleg and Uri Weiser. MMX technology extension to Intel architecture. *IEEE Micro*, 16(4):42–50, August 1996.

[14] Michael D. Smith and Glenn Holloway. The Machine-SUIF Documentation Set. Technical report, Harvard University, 2000.

[15] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, 29(12):31–37, December 1994.